3-е издание

Охватывает MySQL, Oracle, PostgreSQL и SQL Server

SQL

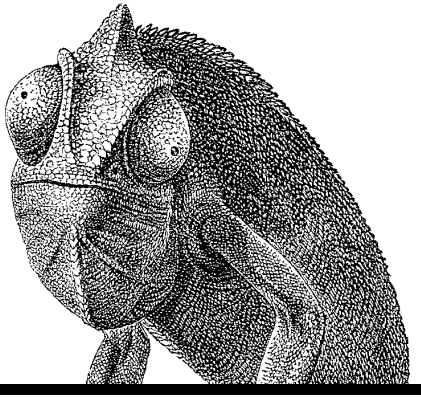
СПРАВОЧНИК

СИМВОЛ®

O'REILLY®

Кевин Е. Кляйн,
Дэниэл Кляйн
и Брэнд Хант

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-165-3, название «SQL. Справочник, 3е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.



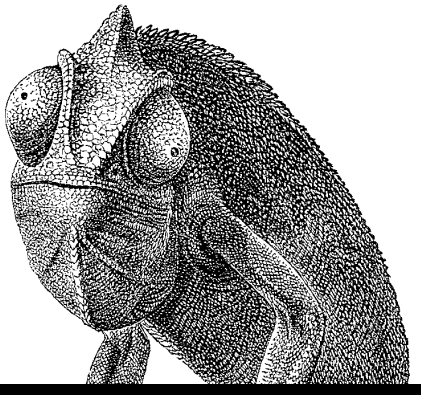
SQL IN A NUTSHELL

A Desktop Quick Reference

Third edition

Kevin E. Kline
with Daniel Klien & Brand Hunt

O'REILLY®



SQL
СПРАВОЧНИК

Третье издание

Кевин Е. Кляйн,
Дэниэл Кляйн и Брэнд Хант

Санкт-Петербург — Москва
2010

Кевин Кляйн, Дэниэл Кляйн и Брэнд Хант

SQL. Справочник, 3-е издание

Перевод Е. Демьянова, А. Слинкина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Научный редактор	*А. Рындин*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Кляйн К., Кляйн Д., Хант Б.
SQL. Справочник, 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2010. – 656 с., ил.
ISBN 978-5-93286-165-3

В третьем издании книги «SQL. Справочник» описываются все операторы SQL согласно последнему стандарту ANSI SQL2003, а также особенности реализации этих операторов в наиболее популярных СУБД: Microsoft SQL Server 2008, Oracle 11g, MySQL 5.1 и PostgreSQL 8.3. Издание содержит описание реляционных моделей данных, объяснение основных концепций реляционных СУБД, полное описание синтаксиса SQL, а также описание специфических функций, характерных для каждой СУБД.

Справочник подготовлен профессиональными администраторами и опытными разработчиками, использующими различные диалекты SQL для поддержки сложных корпоративных приложений. Основная задача издания – служить кроссплатформенным руководством для тех, кто, не будучи экспертами, занимается переносом кода (включая пользовательские приложения) между различными СУБД. Независимо от того, является ли читатель новичком в SQL или имеет значительный опыт его использования, он найдет много полезных советов и приемов в этой лаконичной и удобной для работы книге.

ISBN 978-5-93286-165-3
ISBN 978-0-596-51884-4 (англ)

© Издательство Символ-Плюс, 2010

Authorized translation of the English edition © 2008 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 30.09.2009. Формат 70x100 1/16 . Печать офсетная.
Объем 41 печ. л. Тираж 1500 экз. Заказ N
Отпечатано с диапозитивов в ГУП «Типографии «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 7

1. История и реализации SQL 15

Реляционная модель и ANSI SQL 15
Правила Кодда для реляционных баз данных 16
Правила Кодда в действии: простой пример SELECT 22
История стандарта SQL 24
Новое в SQL2006 24
Новое в SQL2003 (SQL3) 24
Уровни соответствия стандарту 25
Дополнительные пакеты возможностей в стандарте SQL3 26
Классы операторов SQL3 28
Диалекты SQL 29

2. Основные концепции 30

Рассматриваемые СУБД 30
Категории синтаксиса 31
Идентификаторы 31
Литералы 37
Операторы 38
Зарезервированные и ключевые слова 44
SQL2003 и типы данных 44
Типы данных MySQL 50
Типы данных Oracle 55
Типы данных PostgreSQL 58
Типы данных SQL Server 62
Ограничения целостности 67
Область применения 67
Синтаксис 67
Первичные ключи 68
Внешние ключи 69
Уникальные ключи 72
Проверочные ограничения целостности 73

3. Справочник операторов SQL 75

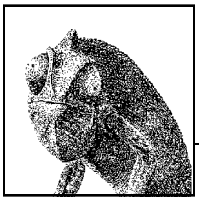
Как читать эту главу? 75
Поддержка SQL платформами 76
Перечень операторов SQL 79

4. Функции SQL 501

Классификация функций 501
Детерминированные и недетерминированные функции 501
Агрегатные и скалярные функции 502
Оконные функции 502
Агрегатные функции в ANSI SQL 502
Оконные функции в ANSI SQL 520
Синтаксис оконных функций в ANSI SQL2003 521
Синтаксис оконных функций в Oracle 521
Синтаксис оконных функций в SQL Server 521
Разбиение (partitioning) 522
Упорядочение (ordering) 522
Группировка и скользящие окна (framing) 523
Перечень оконных функций 524
Скалярные функции в стандарте ANSI SQL 528
Встроенные скалярные функции 529
Функции CASE и CAST 530
Числовые скалярные функции 533
Строковые функции и операторы 544
Платформо-зависимые расширения 550
Функции, поддерживаемые MySQL 551
Функции, поддерживаемые Oracle 570
Функции, поддерживаемые PostgreSQL 596
Функции, поддерживаемые SQL Server 610

А. Ключевые слова: общие и платформо-зависимые 624

Алфавитный указатель 633



Предисловие

С момента появления в 1970-х годах развитие языка SQL (structured query language, структурированный язык запросов) шло в параллели с информационным бумом, и в результате SQL стал самым широко используемым языком управления базами данных. Множество компаний и отдельных разработчиков, включая разработчиков ПО с открытым кодом (*http://www.opensource.org*), создали свои диалекты SQL в соответствии со своими потребностями. В то же время комитеты, занимающиеся стандартами, разрабатывали растущий список стандартных возможностей.

В этом справочнике описываются все операторы SQL согласно последнему стандарту ANSI SQL2003(SQL3), а также особенности реализации этих операторов в различных СУБД. В этой книге вы найдете описание реляционных моделей данных, объяснение основных концепций реляционных СУБД, полное описание синтаксиса SQL.

Но самое главное, особенно для программиста или разработчика (проектировщика ПО), это то, что эта книга является справочником по двум самым популярным коммерческим СУБД (Microsoft SQL Server и Oracle) и двум самым известным СУБД с открытым кодом (MySQL и PostgreSQL). Внимание, уделяемое в этой книге системам с открытым кодом, отражает возрастающую важность таких систем для компьютерного сообщества.

Синтаксис SQL, описанный в этой книге, включает следующие варианты реализации:

- В соответствии со стандартом ANSI SQL2003 (SQL3)
- MySQL версии 5.1
- Oracle Database 11g
- PostgreSQL версии 8.3
- Microsoft SQL Server 2008

Зачем нужна эта книга?

Основным источником информации по реляционным СУБД традиционно является документация и файлы справки, поставляемые разработчиками. Хотя доку-

ментация и является необходимым ресурсом, к которому большинство разработчиков и администраторов обращается в первую очередь, такая документация имеет определенные ограничения:

- Реализация языка SQL описывается без указания того, насколько хорошо эта реализация согласуется с ANSI стандартом
- Описывается только один конкретный продукт, но не дается никакого описания возможных проблем при переводе, миграции или интеграции
- Описание приводится в множестве маленьких разрозненных файлов
- Описание отдельных команд зачастую настолько детально, что сбивает с толку, скрывая простые и очевидные способы применения этих команд в повседневной работе программиста или администратора.

Другими словами, поставляемая с СУБД документация дает исчерпывающее объяснение каждого аспекта конкретной версии продукта. И это нормально: в конце концов, документация и нужна для того, чтобы дать всю информацию о продукте. В документации приводится синтаксис команд (со всеми мыслимыми вариантами), и в общих чертах описывается использование. Тем не менее, если вы работаете с разными СУБД и хотите работать продуктивно, то вы вряд ли будете использовать эти специфические особенности; наоборот, вы будете использовать те возможности, которые необходимы в большинстве реальных ситуаций.

Эта книга начинается там, где заканчивается документация. В книге использован опыт профессиональных администраторов баз данных и разработчиков, использующих различные диалекты SQL в своей каждодневной работе по поддержке сложных корпоративных приложений. Вам предлагается воспользоваться их опытом, изложенным в компактном и легко используемом формате. Независимо от того, являетесь ли вы новичком в SQL или используете его на протяжении нескольких лет, вы найдете множество полезных советов и приемов. А при переходе от одной СУБД к другой важно быть в курсе основных возможных проблем, которые могут возникнуть, если вы не будете осведомлены и осторожны.

Для кого предназначена эта книга?

Этот справочник будет полезен разным читателям. Это и программисты, которым нужен ясный и удобный справочник по SQL; и разработчики, занимающиеся миграцией с одного диалекта SQL на другой; и администраторы баз данных, которым нужно выполнять тысячи команд для поддержки корпоративных информационных систем в рабочем состоянии, а также управлять объектами баз данных, такими как таблицы, индексы, представления.

Эта книга является справочником, а не учебником. Мы, к примеру, не объясняем концепцию элементарного цикла; опытному программисту это известно, и ему «мясо» подавай. Поэтому мы объясняем, как должен работать курсор по стандарту ANSI, и как курсор работает на описываемых платформах, особенности курсоров на каждой платформе, а также основные потенциальные проблемы при использовании курсоров и методы решения этих проблем.

Хотя эта книга не учебник по SQL или проектированию баз данных, мы даем некоторую вводную информацию и надеемся, что она окажется вам полезной. Главы 1 и 2 дают краткое введение в SQL, описывая истоки возникновения этого

языка, основные структуры и базовые операции. Если вы новичок в SQL, то эти главы помогут вам начать работать.

Как организована эта книга

Книга содержит четыре главы и приложение:

Глава 1. *История и реализации SQL*

Обсуждаются реляционная модель данных, описываются текущий и предыдущие стандарты SQL, дается введение в реализации SQL, описываемые в этой книге.

Глава 2. *Основные концепции*

Описываются концепции, необходимые для понимания реляционных баз данных и SQL команд.

Глава 3. *Справочник операторов SQL*

Приводится алфавитный справочник операторов SQL. В этой главе описывается последний стандарт ANSI (SQL3) и детали реализации каждой команды в MySQL, Oracle, PostgreSQL и SQL Server.

Глава 4. *Функции SQL*

Описываются все функции SQL по стандарту ANSI SQL3, а также особенности реализации этих функций производителями. Также приводятся все специфические функции, реализованные в каждой платформе.

Приложение. *Ключевые слова: общие и платформо-зависимые*

Приводится список ключевых слов, зарезервированных стандартом SQL3 и фирмами-производителями. Не используйте слова из этого списка для именования объектов и переменных.

Как использовать эту книгу

Книга в первую очередь является справочником команд. Следовательно, вы будете использовать ее для получения информации по командам и функциям SQL. Однако при наличии описания по стандарту и по четырем платформам описание каждой команды может быть очень большим.

Для того чтобы избежать излишнего многословия при описании, мы сравниваем реализацию каждой платформы со стандартом SQL3. Если реализация согласуется со стандартом, мы не повторяем описание еще раз.

Обобщенные и переносимые между платформами примеры приводятся в описании каждой команды по стандарту SQL3. Так как стандарт развивается быстрее большинства реализаций, то для тех команд, которые не поддерживаются ни одной из рассматриваемых в книге платформ, примеры не приводятся. С другой стороны, приводятся примеры для пояснения многочисленных расширений и улучшений команд, реализованных на каждой конкретной платформе.

Мы понимаем, что при таком подходе может потребоваться неоднократно перепрыгивать от описания платформенной реализации команды обратно к описанию

команды в стандарте SQL3. Однако мы верим, что это все же лучше, чем забивать книгу сотнями страниц повторяющейся информации.

Ресурсы

На следующих веб-сайтах можно найти дополнительную информацию о платформах, рассматриваемых в этой книге:

MySQL

Корпоративным ресурсом по MySQL является *http://www.mysql.com*, еще один хороший сайт *http://dev.mysql.com/doc/refman/5.1/en/*. Хорошим ресурсом для разработчиков с множеством полезных примеров является Devshed.com Ищите информацию по MySQL по ссылке *http://www.devshed.com/c/b/MySQL*.

PostgreSQL

Домашняя страница этой СУБД с открытым кодом расположена на *http://www.postgresql.org*. Помимо огромного количества информации для скачивания на сайте поддерживается рассылка для пользователей PostgreSQL. Другой достойный внимания сайт, предлагающий также поддержку коммерческим пользователям, – *http://www.pgsql.com*.

Oracle

Портал Oracle расположен на *http://www.oracle.com*. Огромное количество информации для постоянных пользователей Oracle находится по адресу *http://www.oracle.com/technology/*. Вся документация доступна по ссылке *http://www.oracle.com/technology/documentation/index.html*. Для полезной информации об Oracle от независимых источников посетите сайт Independent Oracle User Group *http://www.ioug.org*.

SQL Server

Официальный сайт Microsoft SQL Server *http://www.microsoft.com/sql/*. Другой хороший ресурс расположен на сайте Профессиональной ассоциации пользователей SQL Server (PASS) *http://www.sqlpass.org*.

Изменения в третьем издании

Основной причиной для выпуска нового издания книги о технологии обычно является прогресс этой технологии. С момента выпуска второго издания этой книги был создан новый стандарт ANSI, а большинство производителей СУБД выпустили как минимум по одному новому релизу. В результате наши читатели получат свежую информацию о последних версиях языка SQL, представленных на рынке.

Вот основные изменения в третьем издании:

Уменьшенный охват

Читателям второго издания пришлось по душе представленное в нем описание всех основных СУБД. Но поддержка такого большого количества информации в актуальном состоянии оказалась сложным делом. Поэтому по результатам большого опроса читателей мы решили исключить из этого изда-

ния обзор двух наименее популярных платформ: Sybase Adaptive Server и IBM DB2 UDB.

Улучшенная организация

При подготовке второго издания была проделана громадная работа по описанию всего, что читатель может захотеть узнать о командах и функциях SQL в основных СУБД. Но необходимую информацию не всегда было легко найти. Мы улучшили указатели, содержание и колонтитулы страниц, чтобы поиск был быстрее и эффективнее.

Больше примеров

Примеров слишком много не бывает. Мы пополнили и так немалое число базовых примеров, добавив примеры кода, поясняющие уникальные и мощные возможности стандартного SQL и расширений, предлагаемых платформами по отдельности.

Обозначения, принятые в данной книге

В этой книге мы придерживаемся следующих обозначений:

`Моноширинный шрифт`

Применяется для описания синтаксиса, фрагментов кода и примеров.

`Моноширинный шрифт с курсивом`

Используется для указания переменных в коде, вместо имен которых пользователь должен ввести собственные значения.

`Жирный моноширинный шрифт`

Используется для выделения отдельных фрагментов кода.

Курсив

Используется при введении новых терминов, для привлечения внимания, для указания команд или вводимых пользователем имен файлов и папок, а также для указания переменных в тексте.

Жирный шрифт

Используется для имен объектов базы данных, таких как таблицы, столбцы или хранимые процедуры.

КУРСИВ В ВЕРХНЕМ РЕГИСТРЕ

Используется для ключевых слов SQL.

Так оформляется дополнительная информация (например, ссылка на материалы по теме или более подробное объяснение).

Такой текст содержит предупреждение или предостережение.

Использование примеров кода

Эта книга должна служить подспорьем в вашей работе. В целом вы можете свободно использовать приведенный на ее страницах код в своих приложениях или документации. Если объем заимствованного вами кода невелик, нет необходимости обращаться к нам за разрешением. К примеру, вставка в вашу программу нескольких строчек кода из данной книги разрешения не требует. В то же время продажа или иное распространение CD-дисков с записью примеров, взятых из книг издательства O'Reilly, требует разрешения. Для цитирования материалов этой книги при ответе на вопрос специального разрешения не нужно, но при включении в составляемую вами документацию значительного объема кода из данной книги получение нашего разрешения потребуется.

При использовании наших материалов мы не требуем обязательной ссылки на источник, однако будем вам за нее благодарны. Ссылка на источник, как правило, включает в себя название, имя автора, издательство и ISBN книги, например: «*SQL in a Nutshell*, Third Edition, by Kevin E. Kline with Daniel Kline and Brand Hunt. Copyright 2009 O'Reilly Media, Inc., 978-0-596-51884-4».

Если вы чувствуете, что при использовании кода из книги вы вышли за рамки свободного распространения, оговоренные выше, пожалуйста, свяжитесь с нами по адресу *permissions@oreilly.com*.

Нам интересно ваше мнение

Мы, конечно, тестировали и выверяли информацию, приведенную в данной книге, максимально внимательно, но, возможно, вы обнаружите, что в какие-то особенности были внесены изменения (или мы что-то пропустили!). Мы будем вам признательны за подобные сведения, особенно, если это поможет сделать книгу лучше. Пожалуйста, присылайте свои вопросы и комментарии по поводу этой книги издателю по следующему адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (для звонков из США или Канады)
(707) 829-0515 (для международных или местных звонков)
(707) 829-0104 (факс)

На сайте издательства есть посвященная этой книге страница, где приводятся примеры, список опечаток и другие дополнительные сведения. Адрес страницы:

www.oreilly.com/catalog/9780596518844/

Пожалуйста, укажите нам на наши опечатки и синтаксические ошибки. (Вы можете представить, насколько трудно откорректировать книгу, описывающую стандарт ANSI и четыре разные реализации.) Адрес электронной почты для комментариев и вопросов технического характера, связанных с этой книгой:

booksquestions@oreilly.com

Чтобы получить информацию о наших книгах, конференциях, ресурсных центрах и сети O'Reilly Network, посетите наш веб-сайт, расположенный по адресу:

www.oreilly.com

Safari® Books Online

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

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

Благодарности

Мы хотели бы поблагодарить нескольких человек из издательства O'Reilly Media. В первую очередь, мы испытываем огромную благодарность к Джулии Стил (Julie Steel), редактору третьего издания. Джулия не давала нам расслабиться, и благодаря ей мы закончили то, что начали. Благодаря ее мягкому и предупредительному стилю с Джулией было очень приятно работать. Спасибо тебе за все, что ты для нас сделала!

Также мы благодарим наших замечательных технических рецензентов: Питера Гулутзана (Peter Gulutzan), который отвечал за стандарт SQL и работал с нами еще со второго издания; Томаса Локхарта (Thomas Lockhart), специалиста по PostgreSQL; Рональда Брэдфорда (Ronald Bradford) – Oracle/MySQL и Ричарда Соннена (Richard Sonnen) – Oracle. Сердечное спасибо! Вы внесли существенный вклад в точность, удобочитаемость и ценность этой книги. Без вас наши разделы о расширениях языка были бы неполными. В дополнение снимаем шляпу перед Питером Гулутзаном и Труди Пельтцер (Trudy Pelzer) за их книгу «SQL-99 Complete, Really!», которая помогла нам разобраться в стандарте SQL3.

Благодарности Кевина Кляйна

В создании толстой книги, которую вы держите в руках, нам помогали многие люди. Мы выражаем признательность тем, кто помог этой книге стать реальностью.

Во-первых, большое спасибо двум моим фантастическим соавторам – Дэну и Брэнду. Вы удивительные ребята, и с вами было приятно работать. Крепко обнимаю Джулию Стил, нашего редактора из O'Reilly Media.

Большое спасибо моим коллегам из Quest Software за поддержку и ободрение. Кристиан Хаскер (Christian Hasker), Энди Грант (Andy Grant), Хизер Эйкман (Heather Eichman), Дэвид Гугик (David Gugick), Билли Босворс (Billy Bosworth), Дуглас Кристалл (Douglas Chrystall), Дэвид Свонсон (David Swanson), Джейсон

Холл (Jason Hall), Эриэл Вэйл (Ariel Weil) и многие другие мои друзья из Quest Software, – спасибо, что сделали эти последние шесть лет в Quest Software такими яркими!

Посвящаю эту книгу своим любимым Дилану, Эмили, Анне и Кэти. Вы были моей надеждой, дыханием и светом в моменты, когда казалось, что в мире уже не оставалось ни надежды, ни дыхания, ни света. Спасибо, что любите меня так сильно и бескорыстно. И наконец, спасибо моей драгоценнейшей Рэйчел – твоя любовь сохранила мое сердце и веру.

Благодарности Дэниэла Кляйна

Я хотел бы поблагодарить моего брата Кевина за его постоянную готовность работать вместе со мной; моих коллег из Университета Анкоридж штата Аляска за их пожелания; и читателей первых двух изданий за их честные отзывы и здравую критику. Мы также получили огромное количество отзывов от переводчиков первых двух изданий, и за это им также большое спасибо.

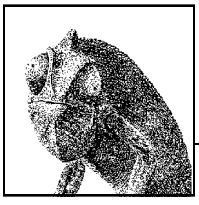
Благодарности Брэнда Ханта

Спасибо моей жене Мишель: без твоей поддержки и любви я бы не был частью этого проекта. Я ценю каждый момент нашей совместной жизни, и особенно ценю, что ты прощала мне постоянное ночное клацанье по клавиатуре, мешавшее тебе спать.

Спасибо моим родителям Рексу и Джэки, тем людям, в наибольшей степени благодаря которым мне удается доводить до конца свои начинания, особенно те, которые требуют многократных усилий (например, написание книги!).

Спасибо членам нашей команды – Кевину, Дэниэлу и Джонатану – за то, что дали мне участвовать в этом проекте и потратили силы на обучение человека, которому впервые довелось писать книгу для O'Reilly. Ваш профессионализм, этика рабочих отношений и умение превратить самую нудную задачу в удовольствие так восхищают, что я намерен украсть эти умения и владеть ими в дальнейшем!

Спасибо моим друзьям и коллегам из Rogue Wave Software, ProWorks, NewCode Technology и Systems Research and Development за предоставление мне песочницы, в которой я мог оттачивать свои навыки SQL, баз данных, бизнеса, разработки программного обеспечения, письма и дружбы. Это Гус Уотрес (Gus Waters), Грэг Коупер (Greg Koerper), Марк Мэнли (Marc Manley), Вэнди Минн (Wendi Minne), Эрин Фоли (Erin Foley), Элэйн Кулл (Elaine Cull), Рэндал Робинсон (Randall Robinson), Дэйв Риттер (Dave Ritter), Эдин Дзулиц (Edin Zulic), Дэвид Нур (David Noor), Джим Шур (Jim Shur), Крис Мосбрукер (Chris Mosbrucker), Дэн Робин (Dan Robin), Майк Фокс (Mike Faux), Джейсон Протеро (Jason Prothero), Тим Романовски (Tim Romanowski), Энди Мосбрукер (Andy Mosbrucker), Джефф Джонас (Jeff Jonas), Джефф Бутчер (Jeff Butcher), Чарли Барбур (Charlie Barbour), Стив Данхэм (Steve Dunham), Брайэн Мэйси (Brian Macy) и Зив Мейлер (Ze'ev Mehler).



1 История и реализации SQL

В начале 1970-х работы сотрудника IBM профессора Э. Ф. Кодда послужили основой при разработке продукта для работы с реляционными данными, получившего название SEQUEL (Structured English Query Language). SEQUEL позднее превратился в SQL (Structured Query Language).

IBM, равно как и другие производители СУБД, был очень заинтересован в стандартизации методов доступа и манипуляции данными в реляционных базах данных. Хотя в IBM была разработана теория реляционных баз данных, компания Oracle первой вывела технологию на рынок. Со временем SQL стал достаточно популярным, чтобы привлечь к себе внимание Американского национального института по стандартам (ANSI), выпустившего стандарты SQL в 1986, 1989, 1992, 1999, 2003 и 2006 годах. В этом издании используется стандарт 2003 года, так как стандарт SQL2006 описывает элементы SQL, не входящие в поле рассмотрения данной книги. (По существу, SQL2006 описывает использование XML в SQL.)

С 1986 года было создано много различных языков, позволявших программистам и разработчикам работать с реляционными данными. Но лишь немногие из них были столь же просты для изучения и стали настолько общеприняты, как SQL. Программисты и администраторы получили возможность изучения одного языка, который с небольшими поправками применим к широкому спектру платформ СУБД, приложений и продуктов.

В этом справочнике приводятся описания синтаксиса пяти реализаций SQL2003:

- Синтаксис по стандарту ANSI SQL
- MySQL версии 5.1
- Oracle Database 11g
- PosgreSQL версии 8.3
- Microsoft SQL Server 2008

Реляционная модель и ANSI SQL

Реляционные системы управления базами данных, такие как описываемые в этой книге, являются двигателем множества информационных систем в мире, в особенности веб-приложений и распределенных клиент-серверных вычислительных

систем. Реляционные СУБД позволяют множеству пользователей быстро и одновременно получать доступ, создавать, редактировать данные, не влияя на работу других пользователей. Реляционные СУБД позволяют разработчикам создавать полезные приложения для доступа к данным, а администраторам дают возможность развивать, защищать, оптимизировать информационные ресурсы.

Реляционная СУБД – это система, пользователи которой получают доступ к данным, представленным в виде набора связанных таблиц. Данные хранятся в *таблицах*, состоящих из *строк* и *столбцов*. Таблицы, хранящие независимые данные, могут быть *связаны* между собой, если в них есть столбцы, значения в которых уникально идентифицируют строки. Наборы таких столбцов называют ключами. Кодд впервые описал реляционную теорию в своей исторической работе «Реляционная модель данных для больших разделяемых банков данных», опубликованной в журнале Communications of the ACM (Association for Computing Machinery) в 1970 году. Согласно новой реляционной модели данных Кодда данные должны быть *структурированными* (в таблицы из строк и столбцов), *управляемыми* с помощью операторов, таких как выборка, проекция, объединение, и *целостными* в результате применения правил контроля целостности, таких как первичные и внешние ключи. Кодд также изложил правила, которые должны применяться при проектировании реляционных баз данных. Процедура применения этих правил получила название *нормализации*.

Правила Кодда для реляционных баз данных

Кодд применил строгие математические теории (в основном, теорию множеств) к управлению данными и составил список критериев, которому должна удовлетворять база данных, чтобы считаться реляционной. Основной концепцией реляционной теории является хранение данных в таблицах. Эта концепция сейчас настолько привычна, что кажется тривиальной; однако еще не так давно создание системы, поддерживающей реляционную модель, считалось сложной задачей с малополезным результатом.

Рассмотрим *двенадцать принципов реляционных баз данных*:

1. Логически информация должна быть представлена в виде таблиц.
2. Каждый элемент данных должен быть доступен по комбинации имени таблицы, значения первичного ключа и названия столбца.
3. Значения NULL должны однозначно трактоваться как «отсутствующая информация», а не как пустые строки, пробелы или нули.
4. Метаданные (данные, описывающие базу данных) должны храниться в базе данных – так же, как и обычные данные.
5. Один и тот же язык должен использоваться для создания данных, представлений, ограничений целостности, авторизации, транзакций и работы с данными.
6. Представления должны отражать изменения данных в таблицах, на основе которых они созданы (и наоборот).
7. Должно быть достаточно одной команды для выполнения любой из следующих операций: извлечение данных, вставка данных, изменения данных и удаление данных.

8. Пакетные и пользовательские команды должны быть логически отделены от структур физического хранения и методов доступа.
9. Пакетные и пользовательские команды должны иметь возможность изменения модели данных без пересоздания этих данных или приложений, использующих базу данных.
10. Ограничения целостности должны храниться в метаданных, а не в приложениях.
11. Язык манипуляции данными не должен зависеть от того, являются ли данные централизованными или распределенными.
12. И пакетная, и построчная обработка должны подчиняться одним и тем же ограничениям и правилам целостности.

Эти принципы продолжают оставаться «лакмусовой бумажкой» для определения реляционной СУБД: платформа, не удовлетворяющая всем критериям, не может считаться реляционной. Эти правила относятся не к процессу разработки приложений, а относятся только к самому «движку», к СУБД. На сегодняшний момент большинство коммерческих СУБД проходят тест Кодда. Среди СУБД, рассматриваемых в третьем издании этого справочника, только MySQL не удовлетворял всем принципам, и то только в релизе, который предшествовал рассматриваемому здесь.

Понимание принципов Кодда помогает программистам и разработчикам в правильном проектировании и создании реляционных баз данных. В следующих разделах мы покажем, как SQL удовлетворяет этим принципам.

Структуры данных (правила 1, 2 и 8)

Правила Кодда 1 и 2 формулируются следующим образом: «логически информация должна быть представлена в виде таблиц» и «каждый элемент данных должен быть доступен по комбинации имени таблицы, значения первичного ключа и названия столбца». Таким образом, при создании таблицы не требуется указание того, как базе данных взаимодействовать с лежащими в основе физическими структурами данных. Более того, SQL логически изолирует процесс доступа к данным и физическое хранение данных, как того требует правило 8: «пакетные и пользовательские команды должны быть логически отделены от структур физического хранения и методов доступа».

В реляционной модели данные представляются в виде двухмерной *таблицы*, описывающей отдельную сущность (например, расходы компании). Теоретики называют таблицы *сущностями*. Таблицы состоят из *строк* или *записей* (ученые называют их *кортежами*) и *столбцов* (теоретики называют их *атрибутами*, так как столбец описывает какой-либо атрибут сущности). Пересечение записи и столбца образует *значение*. Один или несколько столбцов, чьи значения позволяют идентифицировать запись, могут выступать в качестве *первичного ключа*. В наши дни это кажется элементарным, но когда эти концепции были предложены, они явились действительно инновационными.

SQL3 определяет сложную иерархическую структуру объектов помимо простых таблиц, являющихся базовыми структурами данных. Реляционный подход состоит в работе с данными на уровне таблиц, а не на уровне отдельных строк. За ориентацией на обработку таблиц стоит работа с множествами. Как следствие,

почти все SQL-команды эффективнее работают с множествами строк, а не с отдельными строками. Другими словами, для эффективного использования SQL необходимо мыслить в терминах наборов данных, а не в терминах отдельных строк.

На рис. 1.1 представлена терминология SQL3, используемая для описания иерархической структуры реляционной базы данных: *кластер* содержит множество каталогов; *каталог* содержит множество схем; *схема* содержит множество объектов, таких как таблицы или представления; *таблицы* состоят из строк и столбцов.

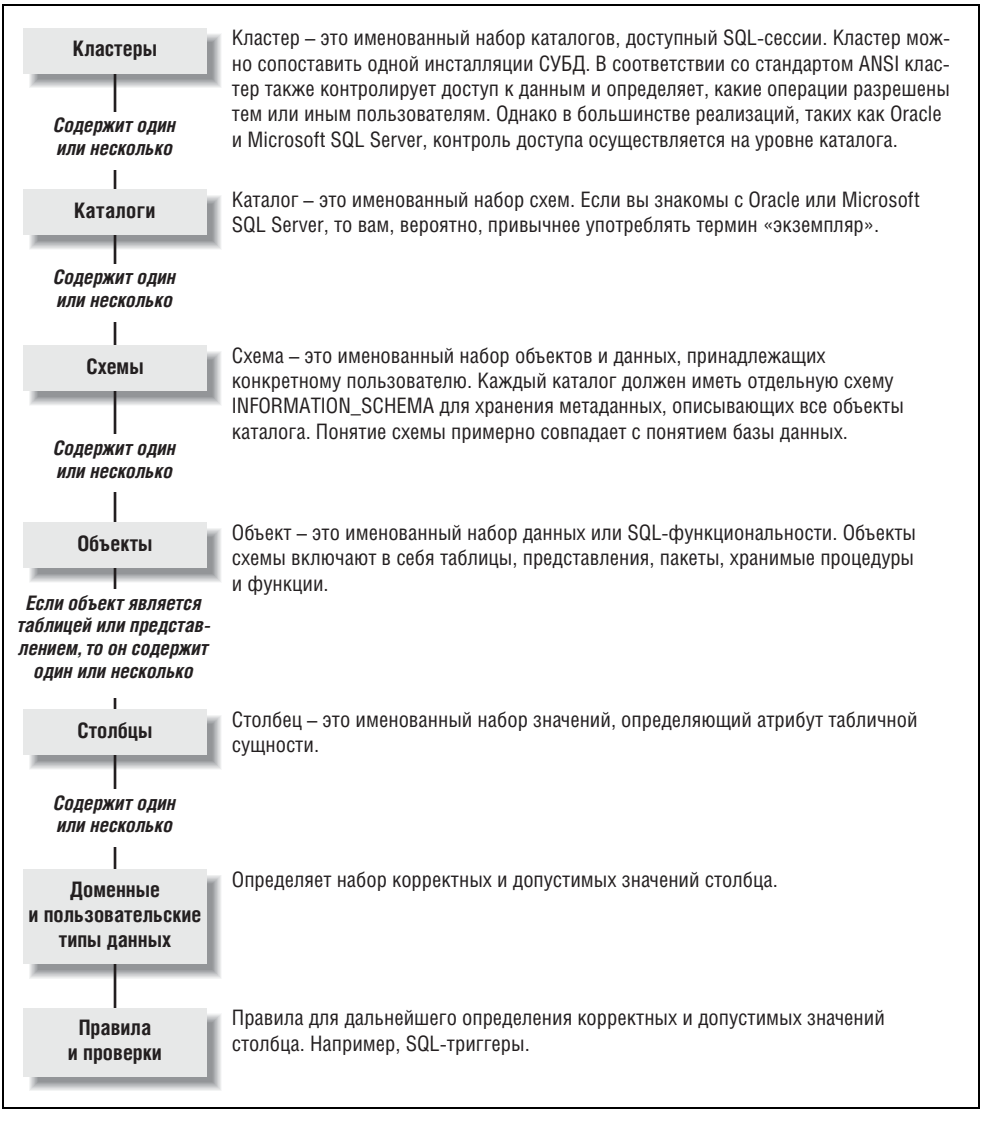


Рис. 1.1. *Иерархия данных в SQL3*

Например, в таблице **Business_Expense** столбец с названием **Expense_Date** мог бы означать дату, в которую были понесены расходы. Каждая запись в таблице описывает отдельную сущность, в нашем случае – все, что описывает расходы (когда произведен, на какую сумму, кем, для чего и т. д.) Каждый атрибут затрат – другими словами, каждый столбец – должен быть *атомарным*, что означает, что столбец записи должен содержать одно и только одно значение. Если таблица создана так, что на пересечении строки и столбца может содержаться несколько значений, то это нарушает одно из базовых положений SQL-проектирования. (Некоторые СУБД, рассматриваемые в этой книге, позволяют хранить в столбце больше одного значения с помощью типов *VARRAY* и *TABLE*.)

Для значений столбцов определяются правила поведения. В первую очередь, все значения столбца должны относиться к одному и тому же *домену* или, по-другому, к одному *типу данных*. К примеру, если столбец **Expense_Date** имеет тип *DATE*, то значение *ELMER* в этот столбец сохранить нельзя, так как это значение является не датой, а строкой, а в столбце **Expense_Date** допустимы только даты. Кроме того, SQL3 позволяет контролировать значения столбцов с помощью *ограничений* (обсуждаются детально в главе 2) и *проверок* (assertion). SQL-ограничение может, к примеру, проверять, что в столбце **Expense_Date** содержатся даты не более чем годичной давности. Доступ к данным всех пользователей и процессов контролируется с помощью *AuthorizationID* или *user*. Доступ или модификация определенных наборов данных может разрешаться или запрещаться на уровне отдельных пользователей.

В реляционных базах также используются понятия *кодировки* и *схем упорядочивания*. Кодировка – это «символы» или «алфавит», используемые «языком» данных. Например, кодировка для американского английского не содержит буквы *ñ* из испанской кодировки. Схема упорядочивания – это правила сортировки для кодировки. Схема упорядочивания определяет, как при манипуляции будут отсортированы данные. Например, для символов американского английского могут быть применены сортировки *алфавитная*, *регистронезависимая* или *алфавитная*, *регистрозависимая*.

Стандарт ANSI не описывает, как сортировки должны выполняться, но говорит о необходимости поддержки основных схем упорядочивания отдельного языка.

При написании SQL-кода для конкретной платформы СУБД вы должны понимать, какая используется схема упорядочивания, потому что это оказывает большое влияние на поведение запросов, в особенности, на фразы *WHERE* и *ORDER BY*. Например, запрос, использующий бинарную схему упорядочивания, вернет записи в другом порядке, нежели запрос, сортирующий данные по алфавиту.

Неопределенные значения (правило 3)

В большинстве СУБД любой тип данных может иметь неопределенное значение (NULL). Неопытные SQL-программисты и разработчики часто принимают NULL за ноль или пробел. Но NULL – это ни то ни другое. В SQL3 NULL буквально означает, что значение неизвестно или не определено. (Вопрос о том, считать ли NULL все-таки неопределенным или же неизвестным значением, является пред-

метом постоянных научных дебатов). Такое разделение позволяет проектировщику базы данных отличать данные, сознательно введенные как ноль, и данные, значение которых не зафиксировано в системе или намеренно проставлено как неопределенное. Как иллюстрацию семантического различия, представим систему работы с платежами. Если в качестве суммы платежа стоит NULL, то это не значит, что товар был куплен бесплатно. Это означает, что количество товара неизвестно или еще не определено.

Существует много отличий в том, как каждая СУБД работает с неопределенными значениями. И это является одним из главных источников проблем при переносе кода между платформами. Например, пустая строка в Oracle представляется значением NULL. Все остальные СУБД, рассматриваемые в этой книге, позволяют явно сохранить пустую строку в столбце с типом *CHAR* или *VARCHAR*.

Побочным эффектом такой неопределенной природы значения NULL является то, что NULL не используется в вычислениях и сравнениях. Вот несколько коротких, но очень важных правил из стандарта ANSI, которые необходимо помнить при работе с неопределенными значениями в выражениях SQL:

- Значение NULL нельзя вставить в столбец, определенный как NOT NULL.
- Значения NULL не равны между собой. Ожидать, что два значения NULL окажутся равны при сравнении двух столбцов – это очень частая ошибка. (Правильным способом определения NULL в логическом выражении или во фразе *WHERE* является использование специальных фраз «IS NULL» и «NOT IS NULL»).
- Значения NULL игнорируются при вычислении по столбцу агрегированных значений, таких как *AVG*, *SUM* или *MAX*.
- Если столбец, содержащий NULL, используется в запросе во фразе *GROUP BY*, то результат запроса содержит одну строку для всех значений NULL. Стандарт ANSI предписывает, что все найденные неопределенные значения должны попасть в одну группу.
- Фразы *DISTINCT* и *ORDER BY*, как и *GROUP BY*, также не различают значения NULL друг от друга. При сортировке с помощью фразу *ORDER BY* производители СУБД могут устанавливать произвольным образом порядок сортировки по умолчанию для NULL.

Метаданные (правила 4 и 10)

Четвертое из правил Кодда для реляционных баз данных требует, чтобы описание базы данных хранилось в таких же таблицах, что и обычные данные. Данные, описывающие саму базу данных, называются *метаданными*. Например, каждый раз, когда вы создаете в базе данных таблицу или представление, создаются и сохраняются записи, описывающие эти новые таблицы. Дополнительные записи требуются для описания столбцов, ключей или ограничений таблицы. Эта технология реализована в большинстве коммерческих и открытых реляционных СУБД. Например, в SQL Server есть так называемые системные таблицы, в которых отслеживается информация о базах данных и всех их объектах. Кроме того, SQL Server использует «системные базы данных» для хранения инфор-

мации о сервере, на котором установлено и сконфигурировано программное обеспечение СУБД.

Язык (правила 5 и 11)

Правила Кодда *не требуют*, чтобы при работе с базами данных использовался именно язык SQL. Эти правила, в частности 5 и 11, только определяют, как должен вести себя язык при работе с реляционными базами данных. Одно время SQL состязался с другими языками (такими как RDO и Fox/PRO), которые также удовлетворяли критериям реляционности, и SQL победил, по трем причинам. Во-первых, SQL относительно простой, интуитивно понятный и похожий на естественный английский, язык, но при этом охватывает большинство аспектов манипуляции данными. Во-вторых, SQL достаточно высокоуровневый язык. Программист или администратор баз данных не должен тратить время и силы на сохранение данных в нужные регистры оперативной памяти или на кэширование данных на диск; все эти задачи СУБД решает автоматически. Наконец, SQL не принадлежит какому-то одному производителю и поэтому успешно был адаптирован к множеству различных платформ.

Представления (правило 6)

Представление – это виртуальная таблица, не хранящая данные физически, а создаваемая на лету из SQL-запроса каждый раз, когда происходит обращение к представлению. Представления позволяют создавать несколько разных образов одних и тех же исходных данных для разных пользователей без необходимости модифицировать способ хранения данных.

Некоторые производители СУБД поддерживают объекты, называемые материализованными представлениями. Но не попадитесь на схожесть названий: материализованные представления не подчиняются тем же правилам, что и стандартные ANSI-представления.

Операции над множествами (правила 7 и 12)

Многие языки для манипуляции данными, такие как многоуважаемый Xbase, выполняют операции совсем не так, как SQL. В этих языках необходимо явно указывать, что делать с данными, – с каждой строкой. Программа в цикле перебирает все строки, применяя логику обработки к каждой строке, и поэтому такой стиль программирования часто называется *построчной обработкой,* или *процедурным программированием*.

Программы на SQL логически оперируют *множествами* данных. Теория множеств применяется почти во всех операторах SQL, включая операторы *SELECT*, *INSERT*, *UPDATE* и *DELETE*. Данные выбираются из множества, называемого таблицей. В отличие от построчной обработки, при работе с множествами программист говорит, что ему требуется получить, а не как обрабатывать каждый кусочек данных. Иногда работу с множествами называют *декларативной обработкой*, потому что программист декларирует желаемый результат (например, «Мне нужны все сотрудники из южного региона с зарплатой более $70 000 в год»), а не описывает всю процедуру извлечения данных.

Теория множеств была разработана математиком Георгом Кантором в конце XIX века. В то время она была воспринята достаточно противоречиво. Сегодня теория множеств настолько плотно вошла в нашу жизнь, что преподается в школе. Простыми и общеизвестными применениями теории множеств являются карточные каталоги, десятичная система классификации Дьюи, алфавитные телефонные справочники.

Связь теории множеств и реляционных баз данных детально описывается в следующих разделах.

Правила Кодда в действии: простой пример SELECT

До сих пор в этой главе мы рассматривали отдельные аспекты теории реляционных баз данных, определенных Коддом и реализованных в стандарте ANSI SQL. В этом разделе приводится общий обзор наиболее важного SQL-оператора, *SELECT*, и основных его применений – а именно, выполнение реляционных операций проекции, выборки и соединения.

Проекция

Выбираются конкретные столбцы данных

Выборка

Выбираются конкретные строки

Соединение

Одним результирующим множеством возвращаются строки столбцы из нескольких таблиц

Хотя на первый взгляд может показаться, что оператор *SELECT* выполняет только реляционную операцию выборки, но на самом деле он выполняет все три операции.

Следующий оператор осуществляет проекцию, выбирая имя и фамилию автора из таблицы **authors**:

```
SELECT au_fname, au_lname, state
FROM   authors
```

Результат этого оператора *SELECT* представляется в виде другой таблицы данных:

```
au_fname          au_lname            state
----------------- ------------------- -----
Johnson           White               CA
Marjorie          Green               CA
Cheryl            Carson              CA
Michael           O'Leary             CA
Meander           Smith               KS
Morningstar       Greene              TN
Reginald          Blotchet-Halls      OR
Innes             del Castillo        MI
```

Результат выполнения запроса иногда называют результирующим множеством (result set), рабочей таблицей (work table) или производной таблицей (derived table) в отличие от базовой таблицы, по отношению к которой выполняется оператор *SELECT*.

Важно заметить, что в операторе *SELECT* фраза *SELECT* (т. е. ключевое слово *SELECT* со списком извлекаемых столбцов и выражений) соответствует реляционной операции проекции. Выборка – операция извлечения конкретных строк – реализуется фразой *WHERE* оператора *SELECT*. *WHERE* отфильтровывает ненужные строки результата, оставляя только необходимые. Дополним предыдущий пример условием отбора только авторов не из Калифорнии:

```
SELECT au_fname, au_lname, state
FROM   authors
WHERE  state <> 'CA'
```

В то время как предыдущий запрос возвращал всех авторов, этот запрос возвращает намного меньшее подмножество строк:

```
au_fname        au_lname            state
--------------- ------------------- -----
Meander         Smith               KS
Morningstar     Greene              TN
Reginald        Blotchet-Halls      OR
Innes           del Castillo        MI
```

Комбинируя операции проекции и выборки в одном запросе, вы можете с помощью SQL выбрать только те строки и столбцы, которые вам нужны.

Последняя реляционная операция, которую мы обсудим в этом разделе, – это соединение. Соединение связывает две таблицы и возвращает результирующее множество, состоящее из данных из обеих таблиц.

Разные разработчики СУБД позволяют соединить в одном запросе разное число таблиц. Oracle не накладывает ограничений на число соединяемых таблиц, SQL Server позволяет соединять не больше 256 таблиц.

Описанный в стандарте ANSI способ соединения таблиц заключается в использовании фразы *JOIN* оператора *SELECT*. Более старый способ, известный как тэта-соединения, для задания соединения использует фразу *WHERE*. Следующий пример демонстрирует оба подхода. Каждый оператор выбирает информацию о сотрудниках из таблицы **employee** и название должности из таблицы **jobs**. Первый оператор *SELECT* использует новый синтаксис соединения при помощи *JOIN*, а второй использует тэта-соединение:

```
-- ANSI style
SELECT a.au_fname, a.au_lname, t.title_id
FROM   authors AS a
JOIN   titleauthor AS t ON a.au_id = t.au_id
WHERE  a.state <> 'CA'

-- Theta style
SELECT a.au_fname, a.au_lname, t.title_id
FROM   authors AS a,
```

```
        titleauthor AS t
WHERE   a.au_id = t.au_id
  AND   a.state <> 'CA'
```

Для дополнительной информации о соединениях читайте раздел «Фраза JOIN» в главе 3.

История стандарта SQL

В ответ на быстрый рост числа различных SQL-диалектов ANSI в 1986 опубликовал первый стандарт SQL для внесения большей согласованности в действия разработчиков СУБД. Затем, в 1989 году, последовал второй, широко принятый стандарт. Стандарт SQL был также одобрен Международной организацией по стандартам (ISO). ANSI в 1992 году выпустил следующую версию, известную как SQL92 или SQL2, а затем еще одну в 1999, называемую SQL99 или SQL3. Версия, выпущенная в 2003 году, также называется SQL3 (или SQL2003). В этой книге под термином SQL3 мы подразумеваем именно стандарт 2003 года.

В каждой новой версии стандарта ANSI добавляет в язык новые команды и возможности. К примеру, в SQL99 были добавлены возможности работы с объектными типами данных.

Новое в SQL2006

Комитет ANSI, отвечающий за SQL, выпустил новую версию стандарта в 2006 году, при этом в нем сохранились и были расширены все основные усовершенствования стандарта SQL3. Стандарт SQL2006 можно назвать эволюционным по отношению к стандарту SQL3, он не внес существенных изменений в команды и функции языка. Вместо этого стандарт SQL2006 описал новую функциональную область применения стандарта SQL. Если говорить коротко, то SQL2006 описывает взаимодействие SQL и XML. Например, стандарт SQL2006 описывает, как в реляционную базу импортировать и хранить XML данные, как манипулировать этими данными, и как их представлять в виде XML или обычных SQL данных в XML обертке. Стандарт SQL2006 предоставляет возможности интеграции SQL кода с кодом XQuery, языком запросов к XML, стандартизируемым консорциумом W3C. Так как XML и XQuery являются вполне самостоятельными дисциплинами, они не входят в рамки этой книги и не рассматриваются в ней.

Новое в SQL2003 (SQL3)

SQL99 состоял из двух частей – *Foundation:1999* и *Bindings:1999*. Секция Foundation в SQL3 содержит обе части из SQL99, а также новую часть под названием *Schemata*.

Базовые требования в SQL3 не изменились по сравнению с базовыми требованиями SQL99, поэтому СУБД, удовлетворяющие SQL99, автоматически удовлетворяют SQL3. Хотя базовая часть SQL3 не содержит дополнений (за исключением нескольких новых зарезервированных слов), некоторые отдельные операторы были изменены или дополнены. Так как эти изменения описываются в синтаксисе конкретных операторов в главе 3, то здесь мы говорить о них не будем.

Некоторые элементы базовой части SQL99 в SQL3 были удалены:

- Типы данных *BIT* и *BIT VARYING*
- Фраза *UNION JOIN*
- Оператор *UPDATE ... SET ROW*

Некоторое число других, достаточно экзотических, возможностей было добавлено, удалено или переименовано. Многие новые возможности SQL3 интересны пока только с академической точки зрения, так как пока они не поддерживаются ни одной платформой. Тем не менее, мы не можем пройти мимо следующих возможностей:

Элементарные OLAP-функции

SQL3 вводит дополнительную поддержку OLAP (Online Analytical Processing, аналитическая обработка в реальном времени), включая оконные функции для широко используемых вычислений, таких как скользящее среднее или нарастающий итог. Оконные функции вычисляют агрегированные значения по окнам данных: *ROW_NUMBER*, *RANK*, *DENSE_RANK*, *PERCENT_RANK* и *CUME_DIST*. OLAP-функции подробно описаны в части T611 стандарта. Некоторые платформы СУБД уже имеют поддержку OLAP функций. За подробностями обращайтесь к главе 4.

Выборки (sampling)

SQL3 добавляет во фразу *FROM* фразу *TABLESAMPLE*. Эта возможность будет полезной для статистических запросов к большим базам данных, таким как хранилища данных.

Улучшенные числовые функции

SQL3 вводит большое количество новых числовых функций. В этом случае стандарт пытался успеть за развитием рынка, так как некоторые платформы СУБД уже поддерживают эти функции. Детали в главе 4.

Уровни соответствия стандарту

В стандарте SQL92 впервые было введено понятие уровня соответствия стандарту. Было введено три уровня: *начальный*, *средний* и *полный*. Разработчикам СУБД требовалось достичь как минимум начального уровня соответствия для того, чтобы говорить о совместимости своей СУБД с ANSI SQL. Американский национальный институт по стандартам и технологиям (NIST) добавил *переходный* уровень между начальным и средним, таким образом, уровни соответствия по NIST были: начальный, переходный, средний и полный. Стандарт ANSI имел только три уровня: начальный, средний и полный. Каждый уровень соответствия является надмножеством нижестоящего уровня, т. е. каждый уровень содержит все требования нижестоящего уровня.

Позже стандарт SQL99 изменил уровни соответствия, убрав понятия начального, среднего и полного уровня. В SQL99 говорится о необходимости выполнения разработчиками СУБД всех базовых требований стандарта (Core SQL99) для заявления, что продукт разработчика готов к использованию с SQL99. Core SQL99 включает в себя все требования начального уровня соответствия из SQL92, требования из других уровней соответствия SQL92 и некоторые совершенно новые

требования. При желании разработчик может реализовать дополнительные пакеты возможностей, описанные в SQL99.

Дополнительные пакеты возможностей в стандарте SQL3

Мало кто из разработчиков на текущий момент полностью удовлетворяет или даже превосходит требования стандарта. Базовые требования стандарта это как ограничение скорости на шоссе: кто-то едет чуть быстрее, кто-то чуть медленнее и почти никто не едет точно с допустимой скоростью. Так и реализация стандарта разными разработчиками может отличаться.

Два комитета – один в ANSI, другой в ISO, включающие представителей практически всех разработчиков, – составили описание дополнительных возможностей, о котором рассказывается в этом разделе. В такой объединенной и отчасти политизированной обстановке разработчики пришли к соглашению относительно того, какие возможности и особенности реализации должны войти в стандарт.

Нововведения в стандарте ANSI часто появляются из уже существующих продуктов или являются результатом новых исследований и разработок в научном сообществе. Поэтому поддержка функциональности разработчиками может быть неоднородной. Относительно новым добавлением в стандарт SQL3 является SQL/XML (значительно расширенным в SQL2006). Части стандарта SQL99 остались в SQL3 практически без изменений, хотя иногда немного менялись названия или порядок следования.

Девять дополнительных пакетов возможностей, представляющих различные наборы команд, являются опциональными для реализации. Какие-то элементы стандарта могут поддерживаться различными пакетами, в то же время есть возможности, не поддерживаемые ни одним пакетом. Вот список этих пакетов и их функциональностей:

Часть 1. SQL/Framework

Содержит общие определения и концепции, используемые в стандарте. Определяет структуру стандарта и как соотносятся друг с другом различные части. В этой части также описываются установленные комитетом уровни соответствия стандарту.

Часть 2. SQL/Foundation

Содержит базовые требования (расширенный раздел из SQL99 Core). Самая большая и наиболее важная часть стандарта.

Часть 3. SQL/CLI (Call-Level Interface)

Определяет интерфейс уровня вызовов для динамического использования SQL из внешних приложений. Также включает спецификацию более чем 60 процедур и функций, что способствует написанию действительно переносимых приложений-оберток.

Часть 4. SQL/PSM (Persistent Stored Modules)

Стандартизирует процедурные конструкции языка, похожие на некоторые диалекты SQL, такие как PL/SQL или Transact-SQL.

Часть 9. SQL/MED (Management of External Data)

Определяет порядок работы с данными, расположенными вне базы данных, с помощью указателей и интерфейсов-оберток

Часть 10. SQL/OBJ (Object Language Binding)

Описывает использование SQL из программ, написанных на Java. Тесно связано с JDBC, но имеет несколько преимуществ. Кроме того, серьезно отличается от описанного в предыдущем стандарте способа использования SQL из другого языка.

Часть 11. SQL/Schemata

Определяет 85 представлений (на 3 больше, чем в SQL99), используемых для хранения метаданных базы данных и хранящихся в специальной схеме с названием *INFORMATION_SCHEMA*. Несколько представлений изменены по сравнению с SQL99.

Часть 12. SQL/JRT (Java Routines and Types)

Определяет функции и типы языка Java. Теперь поддерживаются такие возможности Java, как статические методы и классы.

Часть 14. SQL/XML

Вводит новый тип с названием XML, четыре новых оператора (*XMLPARSE*, *XMLSERIALIZE*, *XMLROOT* и *XMLCONCAT*), несколько новых функций (описанных в главе 4) и новый предикат *IS DOCUMENT*. Также включает правила для отображения элементов SQL **(идентификаторов, схем и объектов)** на элементы XML.

Обратите внимание, что частей с номерами 5, 6, 7 и 8 в стандарте нет.

Важно понимать, что разработчик может заявлять об ANSI-совместимости своей СУБД, удовлетворяя только базовым требованиям, поэтому нужно читать то, что написано «мелким шрифтом», – подробный перечень поддерживаемых возможностей. Зная, какие возможности описываются в девяти дополнительных пакетах стандарта, вы можете понять возможности конкретной СУБД, а также особенности переноса SQL-кода на другую платформу.

Стандарт ANSI, описывающий извлечение, манипуляцию и управление данными с помощью команд SELECT, JOIN, ALTER TABLE, DROP и других, формализует поведение команд и синтаксические структуры для множества платформ. Этот стандарт стал еще более важным с появлением СУБД с открытым исходным кодом (таких как MySQL или PostgerSQL), которые становятся достаточно популярными и разрабатываются не крупными компаниями, а виртуальными командами разработчиков.

В этом справочнике описывается реализация SQL четырьмя популярными реляционными СУБД. Эти платформы не поддерживают стандарт SQL3 полностью; фактически разработчики постоянно играют с комитетом по стандартам в догонялки. Часто, как только разработчики приближаются к стандарту, комитет обновляет, уточняет или еще как-то меняет стандарт. С другой стороны, разработчики часто реализуют в своих СУБД возможности, еще не описанные в стандарте, но значительно увеличивающие эффективность работы пользователей.

Классы операторов SQL3

Способ разбиения операторов SQL на классы существенно различается в SQL92 и SQL3. Однако старые термины продолжают использоваться, поэтому читателю необходимо иметь о них представление. SQL92 делит операторы на три большие категории:

Язык манипулирования данными (Data Manipulation Language, DML)

Содержит команды для манипулирования с данными, такие как *SELECT*, *INSERT*, *UPDATE* и *DELETE*.

Язык определения данных (Data Definition Language, DDL)

Содержит команды для работы с объектами базы данных, такие как *CREATE* и *DROP*.

Язык управления данными (Data Control Language, DCL)

Содержит команды для управления привилегиями, такие как *GRANT* и *REVOKE*.

SQL3 использует 7 категорий, называемых теперь *классами*, в качестве общего подхода к делению на типы всех команд SQL. Эти «классы» операторов немного отличаются от категорий операторов в SQL92, так как более аккуратно и логично определяют перечень операторов, попадающих в каждый класс, а также позволяют дальнейшее расширение возможностей и списка классов. Также в новые классы попадают некоторые операторы, которые в SQL92 не вписывались достаточно точно ни в одну из категорий.

В табл. 1.1 приводится перечень классов операторов SQL3, и для каждого класса приводятся примеры операторов, рассматриваемых позднее более подробно. В данный момент достаточно только запомнить названия классов.

Таблица 1.1. Классы операторов SQL3

Класс	Описание	Примеры команд
Операторы подключения	Создают и закрывают клиентское подключение	*CONNECT*, *DISCONNECT*
Операторы управления выполнением	Управляют выполнением набора SQL операторов	*CALL*, *RETURN*
Операторы работы с данными	Извлекают и изменяют данные	*SELECT*, *INSERT*, *UPDATE*, *DELETE*
Диагностические операторы	Обеспечивают диагностической информацией, сообщают об исключениях и ошибках	*GET DAGNOSTINCS*
Операторы управления схемой	Управляют объектами в схеме и внутри схемы базы данных	*ALTER*, *CREATE*, *DROP*
Операторы управления сессиями	Управляют поведением по умолчанию и другими параметрами сессий	*SET CONSTRAINT*
Операторы управления транзакциями	Начинают и заканчивают транзакции	*COMMIT*, *ROLLBACK*

Постоянно работающим с SQL необходимо быть в курсе как старых (SQL92), так и новых (SQL3) классов, так как оба варианта еще используются при обращении к SQL-операторам.

Диалекты SQL

Постоянно эволюционирующий SQL породил множество диалектов, используемых на разнообразных платформах. Причиной появления диалектов является то, что пользователям СУБД необходимы новые функции раньше, чем ANSI выпустит новый стандарт. Иногда научное и исследовательское сообщество добавляет в стандарт новые возможности под давлением со стороны конкурирующих на рынке технологий. Например, многие разработчики расширяют возможности программирования в своих СУБД при помощи Java (как в случае с DB2, Oracle и Sybase) или VBScript (в случае Microsoft). В будущем разработчики будут использовать эти языки совместно с SQL для разработки SQL-программ.

Многие диалекты содержат условные операторы (такие как *IF...THEN*), возможности управления потоком выполнения (циклы *WHILE*), переменные и возможности обработки ошибок. Так как всех этих возможностей не было в стандарте на момент возникновения потребности в них со стороны пользователей, то разработчики предложили свои собственные команды и синтаксис. В реальности, у некоторых разработчиков из ранних 80-х есть различия в синтаксисе даже элементарных команд (например, *SELECT*), так как их реализации предшествовали стандарту. ANSI в данное время уточняет стандарт для устранения этих несоответствий.

В некоторых диалектах представлена функциональность, делающая их более близкими к языкам процедурного программирования. К примеру, процедурные возможности этих диалектов SQL могут содержать команды обработки ошибок, команды управления потоком выполнения, условные команды, переменные, массивы и многое другое. В этой книге мы все эти возможности будем называть диалектами, хотя они, по сути, являются процедурными расширениями SQL. Пакет SQL/PSM из стандарта описывает различные возможности, связанные с хранимыми процедурами, и содержит многие расширения, предлагаемые диалектами.

Вот несколько распространенных диалектов:

PL/SQL

Используется в Oracle. Сокращение обозначает Procedural Language/SQL. Диалект во многом похож на язык программирования Ada.

Transact-SQL

Используется в Microsoft SQL Server и Sybase Adaptive Server. С тех пор как в начале 1990-х Microsoft и Sybase начали развивать разные платформы, их реализации Transact-SQL также начали отличаться.

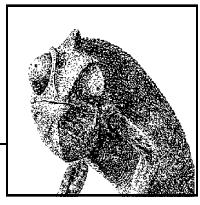
PL/pgSQL

Расширение SQL, реализованное в PostgreSQL. Сокращение расшифровывается как Procedural Language/PostgreSQL.

Пользователи, планирующие плотно работать с одной СУБД, должны изучить все особенности диалекта, используемого этой платформой.

2

Основные концепции



SQL является простым и интуитивно понятным способом взаимодействия с базой данных. Хотя в стандарте SQL2003 не дается определение «базы данных», стандарт определяет основные функции и концепции для создания, извлечения, изменения и удаления данных. Важно знать синтаксис, используемый в стандарте, а также специфические особенности конкретных платформ. В этой главе рассматриваются необходимые основы синтаксиса SQL.

Рассматриваемые СУБД

В этом издании справочника описывается стандарт SQL и его реализация несколькими ведущими поставщиками реляционных СУБД:

MySQL

MySQL – популярная СУБД с открытым кодом, которая известна своей молниеносной производительностью. СУБД работает на множестве операционных систем, включая большинство Linux-систем. По причине основного акцента на скорости работы она поддерживает набор возможностей уже, чем у конкурентов. В этой книге рассматривается MySQL версии 5.1.

Oracle

Oracle является лидирующей СУБД на рынке. Работает практически на любых операционных системах и серверах. Благодаря своей масштабируемой и надежной архитектуре эта СУБД стала выбором многих пользователей. В этом издании мы рассматриваем Oracle 11g.

PostgreSQL

PostgreSQL является системой с самыми богатыми возможностями среди всех СУБД с открытым кодом. В то время как MySQL известен высокой производительностью, PostgreSQL снискал славу за прекрасную поддержку стандарта ANSI, надежную работу с транзакциями, богатый набор поддерживаемых типов данных и объектов. В дополнение к богатым возможностям PostgreSQL также поддерживает широкий список операционных систем и аппаратных платформ. В этой книге рассматривается PostgreSQL 8.2.1.

SQL Server

Microsoft SQL Server является популярной СУБД, доступной только под операционную систему Windows. Особенностями СУБД являются простота использования, широчайший набор возможностей, низкая стоимость и высокая производительность. В этой книге мы рассматриваем Microsoft SQL Server 2008.

Категории синтаксиса

Чтобы начать использовать SQL, необходимо понять, как пишутся операторы. Синтаксис SQL разбивается на четыре основных категории. Категории перечисляются в следующем списке, а затем детально рассматриваются в дальнейших разделах:

Идентификаторы

Пользовательское или системное имя объекта базы данных, например самой базы данных, таблицы, ограничения, столбца, представления и т. п.

Литералы

Любое пользовательское или системное значение, не являющееся идентификатором или ключевым словом. Литералы могут быть строковыми (*«привет»*), числовыми (*1234*), датами (*1 января 1980*), или булевыми (*TRUE*).

Операторы

Обозначения, описывающие действия, которые необходимо выполнить над одним или несколькими выражениями. Операторы, как правило, используются в командах INSERT, DELETE, SELECT и UPDATE. Также операторы часто используются при создании объектов в базе данных.

Зарезервированные и ключевые слова

Имеют особое значение для синтаксического анализатора SQL. *Ключевые слова*, такие как *SELECT*, *GRANT*, *DELETE* или *CREATE*, не могут использоваться в СУБД в качестве идентификаторов. Обычно они являются командами или SQL-операторами. *Зарезервированные слова* – это слова, которые могут в будущем стать ключевыми словами. В дальнейшем при разговоре о любой из этих двух концепций мы будем употреблять только термин *ключевое слово*. Вы можете обойти ограничение на запрет использования ключевых слов в качестве идентификаторов при помощи идентификаторов, взятых в кавычки (обсуждаемых далее). Но так поступать не рекомендуется.

Идентификаторы

Помните, что реляционные СУБД построены на базе теории множеств. В терминах ANSI *кластеры* содержат множества каталогов, *каталоги* содержат множества схем, *схемы* содержат множества объектов и так далее. Большинство СУБД используют похожие термины: *экземпляры* СУБД содержат одну или несколько баз данных, *базы данных* содержат одну или несколько схем, *схемы* содержат одну или несколько таблиц, представлений, хранимых процедур или привилегий доступа к объектам. На каждом уровне этой иерархии элементам требуются уникальные названия (идентификаторы), чтобы к этим элементам можно было

обращаться из программ или системных процессов. Это значит, что каждый объект (независимо от того, является ли он базой данных, таблицей, представлением, столбцом, индексом, ключом, триггером, хранимой процедурой или ограничением) должен быть идентифицирован. При запуске команды на создание объекта вы должны указать идентификатор (т. е. имя) этого нового объекта.

Опытные программисты при выборе идентификаторов руководствуются двумя наборами правил:

Правила именования

Логические правила, используемые проектировщиками баз данных на практике. Постоянное следование этим правилам приводит к более ясной и управляемой структуре базы данных. Это не столько требования SQL, сколько результат жизненного опыта программистов.

Ограничения на идентификаторы

Ограничения, наложенные стандартом SQL и различными реализациями. Например, это могут быть ограничения на допустимую длину идентификатора. Подробнее ограничения каждой СУБД рассматриваются далее в этом разделе.

Правила именования

Правила именования описывают базовые соображения, которыми необходимо руководствоваться при выборе идентификатора объекта. В этом разделе мы приводим правила именования, созданные на базе нашего многолетнего опыта. Стандарт SQL не накладывает никаких ограничений, кроме ограничений на уникальность и длину идентификатора и на набор допустимых к использованию символов. Но вам желательно придерживаться также следующих правил:

Выбирайте осмысленные и содержательные названия

Не называйте таблицу **XP05**; вместо этого используйте название **Expences_2005** для указания того, что в таблице хранятся расходы за 2005 год. Помните, что, возможно, спустя долгое время после вашего ухода другие люди будут пользоваться этой базой данных и этой таблицей, и введенные вами имена объектов должны быть понятными с первого взгляда. Все СУБД имеют ограничения на длину идентификатора, но обычно идентификатор можно сделать достаточно длинным и понятным каждому.

Используйте везде один и тот же регистр символов

Для именования всех объектов используйте либо только верхний регистр символов, либо только нижний. Некоторые СУБД чувствительны к регистру символов, и идентификаторы со смешанными регистрами символов могут впоследствии стать источником проблем.

Будьте последовательны в применении аббревиатур

Однажды выбрав сокращение, используйте его повсеместно. Например, однажды употребив сокращение *EMP* вместо *EMPLOYEE*, используйте его везде; не нужно в одном месте употреблять *EMP*, а в другом *EMPLOYEE*.

Для лучшей читаемости используйте подчеркивания

Название столбца *UPPERCASEWITHUNDERSCORES* читается не так легко, как *UPPERCASE_WITH_UNDERSCORES*.

Не используйте в идентификаторах название компании или продукта

Компании приобретаются, а продукты меняют названия. Эти вещи слишком изменчивы, чтобы быть включенными в названия объектов баз данных.

Не используйте слишком очевидные суффиксы и префиксы

К примеру, не используйте префикс **DB_** для названия базы данных, и не начинайте имя каждого представления с **V_**. Простой запрос к системным таблицам базы данных ответит администратору или программисту на вопрос, какой тип объекта представляется идентификатором.

Не используйте для названия объекта всю доступную длину

Если СУБД позволяет задействовать в имени таблицы до 32 символов, то постарайтесь оставить хотя бы несколько свободных символов про запас. Некоторые СУБД добавляют префиксы или суффиксы к имени таблицы, когда оперируют временными наборами данных.

Не используйте идентификаторы, заключенные в кавычки

Идентификаторы объектов можно заключать в двойные кавычки (в ANSI это называется *идентификаторами с разделителями*). Такие идентификаторы являются регистрозависимыми. Использование кавычек может привести к созданию идентификаторов, которые окажутся сложными в использовании или впоследствии породят проблемы. Например, кавычки позволят вставить в идентификатор пробелы, специальные символы, символы разных регистров и даже управляющие символы, но многие инструменты сторонних производителей (и даже инструменты самих разработчиков СУБД) не могут корректно работать с такими именами объектов. Поэтому не следует использовать идентификаторы, заключенные в кавычки.

В некоторых СУБД для идентификаторов с разделителями можно использовать не только кавычки. Например, в SQL Server для тех же целей можно также использовать квадратные скобки (*[ ]*).

Приведем основные преимущества следования правилам именования. Во-первых, ваш SQL-код становится в некоторой мере самодокументирующимся, потому что используемые имена являются осмысленными и понятными другим пользователям. Во-вторых, ваш SQL-код и базу данных с однотипно названными объектами будет легче поддерживать – особенно тем людям, которые будут делать это после вас. В-третьих, следование правилам именования повышает производительность. Если базу данных вдруг придется переводить на другую платформу, то согласованные и осмысленные названия сэкономят силы и время. Потратив в начале работы несколько минут на размышления о выборе имен, вы избежите проблем в будущем.

Ограничения на идентификаторы

Ограничения на идентификаторы – это ограничения, накладываемые конкретной платформой СУБД. Эти ограничения касаются только обычных идентификаторов, а не заключенных в кавычки. Ограничения стандарта SQL2003 отличаются от тех, что реализованы разработчиками. В табл. 2.1 проводится сравнение

ограничений, накладываемых стандартом и конкретными реализациями СУБД, описываемых в книге.

Таблица 2.1. Ограничения на идентификаторы объектов (за исключением идентификаторов, заключенных в двойные кавычки)

Характеристика	Платформа	Описание
Длина идентификатора	SQL3	128 символов
	MySQL	64 символа, псевдонимы до 128 символов
	Oracle	30 байт (число символов зависит от кодовой страницы), имена баз данных – до 8 байт, имена db-link – до 128 байт
	PostgreSQL	63 символа (свойство *NAMEDATALEN*-1)
	SQL Server	128 символов, временные таблицы – до 116 символов
Идентификатор может содержать	SQL3	Любую цифру, букву и подчеркивания (_)
	MySQL	Любую цифру, букву, символ. Не может состоять только из цифр.
	Oracle	Любую цифру, букву, подчеркивание, символ решетку (#), знак доллара ($), хотя два последних символа использовать не рекомендуется. Имя db-link может также содержать точку.
	PostgreSQL	Любую цифру, букву и подчеркивания (_)
	SQL Server	Любую цифру, букву, символ at (@), символ решетку (#) и знак доллара ($).
Идентификатор должен начинаться	SQL3	С буквы.
	MySQL	С буквы или цифры. Не может состоять только из цифр.
	Oracle	С буквы.
	PostgreSQL	С буквы или подчеркивания (_).
	SQL Server	С буквы, подчеркивания (_), символа at (@), символа решетки (#).
Идентификатор не может содержать	SQL3	Пробелов и специальных символов.
	MySQL	Точку (.), косую черту (/), и символы ASCII(0) и ASCII(255). Одинарные (‘ ’) и двойные (" ") кавычки допустимы только в идентификаторах с ограничителями, идентификатор не может заканчиваться на пробел.
	Oracle	Пробелы, двойные кавычки (" ") и специальные символы.
	PostgreSQL	Двойные кавычки (" ").
	SQL Server	Пробелы и специальные символы.

Характеристика	Платформа	Описание
Поддержка идентификаторов с разделителями	SQL3	Есть.
	MySQL	Есть.
	Oracle	Есть.
	PostgreSQL	Есть.
	SQL Server	Есть.
Разделитель, используемый с идентификаторами	SQL3	Двойные кавычки (" ").
	MySQL	Одинарные кавычки (‘ ’) или двойные кавычки (" ") в режиме ANSI совместимости.
	Oracle	Двойные кавычки (" ").
	PostgreSQL	Двойные кавычки (" ").
	SQL Server	Двойные кавычки (" ") или квадратные скобки ([]); скобки предпочтительнее.
В качестве идентификатора допустимо ключевое слово	SQL3	Нет, если только идентификатор не заключен в кавычки.
	MySQL	Нет, если только идентификатор не заключен в кавычки.
	Oracle	Нет, если только идентификатор не заключен в кавычки.
	PostgreSQL	Нет, если только идентификатор не заключен в кавычки.
	SQL Server	Нет, если только идентификатор не заключен в кавычки.
Схема адресации	SQL3	*Каталог.Схема.Объект*
	MySQL	*База данных.Объект*
	Oracle	*Схема.Объект*
	PostgreSQL	*База данных.Схема.Объект*
	SQL Server	*Сервер.База данных.Схема.Объект*
Идентификатор должен быть уникальным	SQL3	Да.
	MySQL	Да.
	Oracle	Да.
	PostgreSQL	Да.
	SQL Server	Да.
Регистрозависимость	SQL3	Нет.
	MySQL	Только если регистрозависима операционная система (например, MacOS или Unix). Триггеры, группы журнальных файлов и табличные пространства всегда регистрозависимы.

Таблица 2.1 (продолжение)

Характеристика	Платформа	Описание
	Oracle	По умолчанию нет, но можно поменять.
	PostgreSQL	Нет.
	SQL Server	По умолчанию нет, но можно поменять.
Другие ограничения	SQL3	Нет.
	MySQL	Не может состоять только из цифр.
	Oracle	Идентификаторы db-link не могут быть длиннее 128 байт и заключаться в кавычки.
	PostgreSQL	Нет.
	SQL Server	Для идентификаторов с разделителями Microsoft предпочитает использовать квадратные скобки, а не двойные кавычки.

Идентификаторы должны быть уникальны в пределах своей области видимости. Если вспомнить рассмотренную ранее иерархию объектов базы данных, то имена баз данных должны быть уникальны в пределах экземпляра СУБД, в то время как имена таблиц, представлений, функций, триггеров и хранимых процедур должны быть уникальны в пределах конкретной схемы. С другой стороны, таблица и хранимая процедура *могут* иметь одно и то же имя, так как являются объектами разного типа. Имена столбцов, ключей и индексов должны быть уникальны в пределах таблицы или представления и так далее. За более подробной информацией обратитесь к документации по вашей СУБД – некоторые платформы требуют уникальности идентификаторов в определенных ситуациях, другие – не требуют. Например, в Oracle требуется уникальность имен индексов в пределах базы данных, в то время как другие СУБД (например, SQL Server) требуют уникальности имени индекса только в пределах таблицы, по которой этот индекс построен.

Помните, что идентификаторы, заключенные в кавычки, могут нарушать некоторые из описанных ранее правил. Например, такие идентификаторы являются чувствительными к регистру – идентификатор «foo» не равен идентификатору «FOO». Более того, с кавычками в качестве идентификаторов можно использовать зарезервированные слова или запрещенные для использования буквы и символы. К примеру, использование знака процента (%), как правило, запрещено. Тем не менее, в случае особой необходимости, всегда можно использовать знак процента, заключив идентификатор в двойные кавычки. То есть, чтобы назвать таблицу **expense%%ratios**, надо заключить название в двойные кавычки: «**expense%%rates**». Еще раз напомним, что в SQL3 такие идентификаторы называются идентификаторами с разделителями.

Если объект создан с именем, заключенным в кавычки, то мы рекомендуем всегда обращаться к этому объекту с помощью идентификатора с разделителями.

Литералы

В SQL литералом называется любое явно заданное числовое значение, символьная строка, временное значение (дата или время) или булево значение, не являющееся идентификатором или ключевым словом. Реляционные СУБД допускают множество различных литералов в SQL программах. Литералы допустимы для большинства числовых, символьных, булевых и временных типов данных. К примеру, в SQL Server числовые типы данных включают (в числе прочих) типы *INTEGER*, *REAL* и *MONEY*. *Числовые литералы* могут выглядеть следующим образом:

```
30
-117
+883.3338
-6.66
$70000
2E5
7E-3
```

Как показывает этот пример, в SQL Server допускается использование знаковых и беззнаковых чисел в обычной и экспоненциальной записи. А так как в SQL Server есть тип данных *MONEY*, то можно даже использовать знак доллара (*$*). SQL Server *не* допускает использования любых других символов в числовых литералах (кроме 0 1 2 3 4 5 6 7 8 9 + - $. E e), поэтому не пытайтесь использовать запятую (запятая используется в некоторых европейских странах для отделения десятичных разрядов от целой части). Большинство СУБД интерпретируют запятую в числовых литералах как *разделитель списка значений*. Таким образом, литерал вида 3,000 вероятнее всего будет воспринят как два отдельных значения: 3 и 000.

Булевы, символьные и временные литералы могут выглядеть следующим образом:

```
TRUE
'Hello world!'
'OCT-28-1966 22:14:30:00'
```

Символьные литералы всегда должны быть заключены в одинарные кавычки (‘ ’). Это стандартный ограничитель для символьных литералов. В символьных литералах можно использовать не только буквы алфавита. На самом деле, в литерале можно использовать любой символ из таблицы символов. Все это примеры символьных литералов:

```
'1998'
'70,000 + 14000'
'There once was a man from Nantucket,'
'Oct 28, 1966'
```

и все эти значения совместимы с типом данных *CHAR*. Не путайте символьный литерал ‘1998’ с числовым литералом 1998. Так как символьные литералы связаны с символьным типом данных, то неправильно использовать их в арифметических выражениях без явного преобразования типов данных. Некоторые (но не

все) СУБД выполняют автоматическое приведение типов данных к DATE или NUMBER для символьных литералов, содержащих цифры.

Удваивая ограничитель, вы можете при необходимости представить одинарную кавычку в символьном литерале. То есть вы можете использовать две одинарные кавычки каждый раз, когда вам требуется одинарная кавычка в строке. Вот пример для SQL Server, демонстрирующий описанную идею:

```
SELECT 'So he said ''Who''s Le Petomaine?'''
```

Результат выполнения запроса следующий:

```
-----------------
So he said 'Who's Le Petomaine?'
```

Операторы

Оператором называется символ, указывающий какой действие нужно выполнить с одним или несколькими *выражениями*. Операторы чаще всего используются в командах *DELETE*, *INSERT*, *SELECT* и *UPDATE*, но часто встречаются также при создании объектов базы данных, таких как хранимые процедуры, функции, триггеры и представления.

Обычно операторы делят на следующие категории:

Арифметические операторы

Поддерживаются всеми СУБД.

Операторы присваивания

Поддерживаются всеми СУБД.

Битовые операторы

Поддерживаются MySQL и SQL Server.

Операторы сравнения

Поддерживаются всеми СУБД.

Логические операторы

Поддерживаются всеми СУБД.

Унарные операторы

Поддерживаются MySQL, Oracle и SQL Server.

Арифметические операторы

Арифметические операторы выполняют математические действия над двумя числовыми выражениями. В табл. 2.2 представлен список арифметических операторов.

В MySQL, Oracle и SQL Server операторы + и - могут использоваться для выполнения арифметических операций с датами и временем. На различных платформах есть свои, уникальные методы работы с датами.

Таблица 2.2. Арифметические операторы

Арифметический оператор	Действие
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток при целочисленном делении (только в SQL Server)

Операторы присваивания

За исключением Oracle, использующего :=, *оператор присваивания* (=) присваивает значение переменной или псевдониму столбца. Во всех СУБД, рассматриваемых в этой книге, ключевое слово *AS* используется для присваивания псевдонима таблице или столбцу.

Битовые операторы

И в Microsoft SQL Server, и в MySQL имеются *битовые операторы* (табл. 2.3) для битовых манипуляций над целочисленными выражениями. Битовые операторы работают с типами данных *BINARY*, *BIT*, *INT*, *SMALLINT*, *TINYINT* и *VARBINARY*. PostgreSQL поддерживает типы данных *BIT* и *BIT VARYING*; а также битовые операторы *AND*, *OR*, *XOR*, конкатенацию, *NOT* и сдвиги влево и вправо.

Таблица 2.3. Битовые операторы

Битовый оператор	Значение
&	Битовое *И* (два операнда)
|	Битовое *ИЛИ* (два операнда)
^	Битовое исключающее *ИЛИ* (два операнда)

Операторы сравнения

С помощью *операторов сравнения* проверяется равенство или неравенство двух выражений. Результатом операторов сравнения является булево выражение: *TRUE*, *FALSE* или *UNKNOWN*. Обратите внимание, что согласно стандарту ANSI оператор сравнения возвращает NULL, если один из операндов также равен NULL. К примеру, выражение *23+NULL* равно NULL, так же как и выражение *23 февраля 2002 + NULL*. В табл. 2.4 приведен список операторов сравнения.

Булевы операторы сравнения используются чаще всего во фразе *WHERE* для отбора строк, удовлетворяющих условиям поиска. В следующем примере используется оператор «больше или равно»:

```
SELECT *
   FROM Products
   WHERE ProductID >= 347
```

Таблица 2.4. Операторы сравнения

Оператор сравнения	Значение
=	Равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
<>	Не равно
!=	Не равно (не по стандарту ANSI)
!<	Не меньше (не по стандарту ANSI)
!>	Не больше (не по стандарту ANSI)

Логические операторы

Логические операторы обычно используются во фразе *WHERE* для проверки истинности различных условий. Они возвращают либо булево значение *TRUE*, либо *FALSE*. В табл. 2.5 приводится список булевых операторов. Следует помнить, что не все СУБД поддерживают полный перечень этих операторов.

Таблица 2.5. Логические операторы

Логический оператор	Значение
ALL	TRUE, если истинно каждое условие из множества
AND	TRUE, если оба булевых выражения истинны
ANY	TRUE, если истинно хотя бы одно условие из множества
BETWEEN	TRUE, если операнд попадает в заданный интервал
EXISTS	TRUE, если подзапрос возвращает хотя бы одну строку
IN	TRUE, если операнд равен хотя бы одному выражению из списка или строке из результата подзапроса
LIKE	TRUE, если операнд удовлетворяет шаблону
NOT	Заменяет булево значение на противоположное
OR	TRUE, если хотя бы одно из двух булевых выражений истинно
SOME	TRUE, если истинны какие-либо условия из множества

Унарные операторы

Унарные операторы используются с одним операндом, являющимся выражением любого числового типа. Исключением является битовый унарный оператор (~), используемый только с целочисленными выражениями (табл. 2.6).

Таблица 2.6. Унарные операторы

Унарный оператор	Значение
+	Положительное числовое значение
-	Отрицательное числовое значение
~	Битовое отрицание (за исключением Oracle)

Приоритет операторов

Иногда выражения, состоящие из операторов, имеют очень сложную структуру. Если выражение состоит из нескольких операторов, то порядок выполнения операторов определяется *приоритетом операторов*. Порядок выполнения действий может влиять на конечный результат.

Операторы имеют разный приоритет. Оператор с более высоким приоритетом выполняется раньше, чем оператор с более низким приоритетом. В следующем списке операторы перечислены в порядке убывания их приоритета:

- *( )* (выражение в скобках)
- +, -, ~ (унарные операторы)
- *, /, % (арифметические операторы)
- +, - (арифметические операторы)
- =, >, <, >=, <= , <>, *!=*, *!>*, *!<* (операторы сравнения)
- ^ (битовое исключающее ИЛИ), *&* (битовое И), | (битовое ИЛИ)
- *NOT*
- *AND*
- *ALL*, *ANY*, *BETWEEN*, *IN*, *LIKE*, *OR*, *SOME*
- = (присвоение переменной)

Если операторы имеют одинаковый приоритет, то они выполняются слева направо. Для изменения предопределенного порядка выполнения операторов в выражении можно использовать скобки. Сперва вычисляется выражение в скобках, а затем за скобками.

Например, следующие выражения в Oracle вернут разные результаты:

```
SELECT 2 * 4 + 5 FROM dual
-- Вычисляется 8 + 5, в результате получается 13.

SELECT 2 * (4 + 5) FROM dual
-- Вычисляется 2 * 9, в результате получается 18.
```

Если в выражении используются вложенные скобки, то сначала вычисляется выражение в наиболее глубоко вложенных скобках.

В следующем примере используются вложенные скобки, наиболее глубоко вложено выражение 5 – 3. Результат этого выражения равен 2. Затем применяется оператор сложения (+) предыдущего результата и 4, получается 6. Наконец, 6 умножается на 2, получается 12:

```
SELECT 2 * (4 + (5 - 3) ) FROM dual
-- Вычисление вложенных скобок дает результат 2 * (4 + 2),
```

```
-- затем выражением приводится 2 * 6, в результате получается 12
RETURN
```

Мы рекомендуем пользоваться скобками для явного указания порядка выполнения операторов в сложных выражениях.

Системные разделители и операторы

Строковые разделители обозначают границы строки символов. *Системные разделители* – это символы из набора символов, имеющие определенное значение для сервера СУБД. Разделители – это символы, используемые для определения порядка или иерархии процессов или элементов списка. Операторы – это разделители, используемые для определения значений в операторах сравнения, включая символы, обычно используемые для арифметических или логических операций. В табл. 2.7 перечислены разделители и операторы, допустимые в SQL.

Таблица 2.7. Разделители и операторы SQL

Символ	Использование	Пример

+	Оператор сложения; в SQL Server используется также как оператор конкатенации	Для всех платформ:
		`SELECT MAX(emp_id) + 1`
		`  FROM employee`

-	Оператор вычитания; также используется для указания диапазона в ограничениях *CHECK*	Как оператор вычитания:
		`SELECT MIN(emp_id) - 1`
		`  FROM employee`
		Как оператор диапазона в ограничении *CHECK*:
		`ALTER TABLE authors`
		`  ADD CONSTRAINT authors_zip_num`
		`  CHECK (zip LIKE '%[0-9]%')`

*	Оператор умножения	`SELECT salary * 0.05 AS 'bonus'` `  FROM employee;`
/	Оператор деления	`SELECT salary / 12 AS 'monthly'` `  FROM employee;`

=	Оператор равенства	`SELECT *`
		`  FROM employee`
		`  WHERE lname = 'Fudd'`

<>	Оператор неравенства (*!=* – это нестандартный эквивалент на некоторых платформах)	Для всех платформ:
		`SELECT *`
		`  FROM employee`
		`  WHERE lname <> 'Fudd'`

<	Оператор «меньше»	`SELECT lname, emp_id, (salary * 0.05) AS bonus`
		`  FROM employee`
		`  WHERE (salary * 0.05) <= 10000`
		`  AND exempt_status < 3`

<=	Оператор «меньше или равно»	

>	Оператор «больше»	`SELECT lname, emp_id, (salary * 0.025) AS bonus`
		`  FROM employee`
		`  WHERE (salary * 0.025) > 10000`
		`  AND exempt_status >= 4`

>=	Оператор «больше или равно»	

Символ	Использование	Пример

()	Используется в выражениях и вызовах функций, для указания порядка выполнения операторов и для указания границ подзапроса	Выражение:
		`SELECT (salary / 12) AS monthly`
		`  FROM employee`
		`  WHERE exempt_status >= 4`
		Вызов функции:
		`SELECT SUM(travel_expenses)`
		`  FROM "expense%%ratios"`
		Порядок вычисления:
		`SELECT (salary / 12) AS monthly,`
		`  ((salary / 12) / 2) AS biweekly`
		`  FROM employee`
		`  WHERE exempt_status >= 4`
		Подзапрос:
		`SELECT *`
		`  FROM stores`
		`  WHERE stor_id IN`
		`  (SELECT stor_id FROM sales`
		`  WHERE ord_date > '01-JAN-2004')`

%	Метасимвол для обозначения произвольной строки	`SELECT *`
		`  FROM employee`
		`  WHERE lname LIKE 'Fud%'`

,	Разделитель списка элементов	`SELECT lname,`
		`  fname,`
		`  ssn,`
		`  hire_date`
		`  FROM employee`
		`  WHERE lname = 'Fudd'`

.	Разделитель в идентификаторе	`SELECT *`
		`  FROM scott.employee`
		`  WHERE lname LIKE 'Fud%'`

‘	Разделитель строкового литерала	`SELECT *`
		`  FROM employee`
		`  WHERE lname LIKE 'FUD%'`
		`  OR fname = 'ELMER'`

“	Разделитель для идентификатора, заключенного в кавычки	`SELECT expense_date,`
		`  SUM(travel_expense)`
		`  FROM "expense%%ratios"`
		`  WHERE expense_date`
		`  BETWEEN '01-JAN-2004'`
		`  AND '01-APR-2004'`

--	Разделитель однострочного комментария (два дефиса и следующий за ними пробел)	`-- Finds all employees like Fudd,Fudge,`
		`and Fudston`
		`SELECT *`
		`  FROM employee`
		`  WHERE lname LIKE 'Fud%'`

/*	Начало многострочного комментария	`/* Finds all employees like Fudd,Fudge,`
		`and Fudston */`
		`SELECT *`
		`  FROM employee`
		`  WHERE lname LIKE 'Fud%'`

*/	Окончание многострочного комментария	

Зарезервированные и ключевые слова

Как и отдельные символы, целые слова или фразы также могут иметь определенное значение в SQL. *Ключевые слова SQL* – это слова, имеющие особое значение для СУБД. Если говорить в общем, то чаще всего ключевые слова используются в SQL-выражениях и не должны использоваться ни для каких других целей. *Зарезервированные слова* не имеют специального значения в данный момент, но вероятно будут использоваться в будущих релизах. В принципе, в большинстве СУБД эти слова *можно* использовать в качестве идентификаторов, но делать этого не следует. К примеру, слово SELECT является ключевым и не должно использоваться в качестве имени таблицы. Чтобы подчеркнуть тот факт, что ключевые слова не следует использовать в качестве идентификатора, SQL стандарт называет их «незарезервированными ключевыми словами».

Хорошим тоном является отказ от использования в качестве идентификаторов слов, являющихся ключевыми хотя бы на одной из широко распространенных платформ, так как приложения часто переводятся с одной платформы на другую.

Ключевые и зарезервированные слова используются не только в SQL-операторах, а могут быть связаны с самой технологией СУБД. Например, ключевое слово *CASCADE* используется, чтобы разрешить таким операциям, как update или delete, каскадное распространение на подчиненные таблицы. Программисты и разработчики должны быть знакомы с опубликованными списками зарезервированных и ключевых слов, чтобы не использовать их в качестве идентификаторов, так как это рано или поздно может привести к проблемам.

Стандарт SQL3 определяет свой собственный список ключевых и зарезервированных слов. Но любая СУБД имеют свои списки, так как в этих платформах используются расширения SQL. Ключевые слова стандарта SQL, а также всех рассматриваемых в справочнике платформ приводятся в приложении.

SQL2003 и типы данных

Таблица состоит из одного или нескольких столбцов. Каждый столбец должен иметь определенный тип данных, обеспечивающий общую классификацию данных, хранящихся в столбце. В реальных приложениях типы данных делают работу с данными эффективной и обеспечивают некоторый контроль над тем, как создаются таблицы и как в этих таблицах хранятся данные. Явное указание типов данных позволяет писать более понятные запросы и улучшает контроль целостности данных.

Тонким моментом стандарта SQL2003 является то, что его типы данных не всегда имеют одинаковую реализацию на различных платформах. Хотя платформы и имеют типы данных, совпадающие со стандартными, это не всегда *действительно* те же типы, что и в стандарте: например, тип данных *BIT* в MySQL на самом деле является ни чем иным, как *CHAR(1)*. Как бы то ни было, типы данных каждой платформы близки к стандарту настолько, чтобы их было легко идентифицировать и использовать.

Официальные типы данных SQL2003 (в отличие от типов данных конкретных платформ) делятся на общие категории, описанные в табл. 2.8. Обратите внимание, что стандарт содержит несколько редко используемых типов данных (*ARRAY*, *MULTISET*, *REF* и *ROW*), которые приводятся только в таблице и больше не рассматриваются нигде в книге.

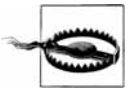
Таблица 2.8. Типы данных SQL2003

Категория	Примеры типов данных (и сокращенные названия)	Описание
BINARY	*BINARY LARGE OBJECT (BLOB)*	Этот тип хранит массив двоичных данных. Значение хранится без указания кодовой страницы и ограничения на длину.
BOOLEAN	*BOOLEAN*	Этот тип данных используется для хранения булевых значений (*TRUE* или *FALSE*)
CHARACTER	*CHAR* *CHARACTER VARYING (VARCHAR)*	Эти типы данных хранят произвольные наборы символов конкретной кодовой страницы. Тип *VARCHAR* допускает хранение значений переменной длины, а тип *CHAR* – только фиксированной. Также в значениях типа *VARCHAR* автоматически удаляются пробелы в конце строки, а в *CHAR*, напротив, все оставшееся место заполняется пробелами.
	NATIONAL CHARACTER (NCHAR), NATIONAL VARYING CHARACTER (NCHAR VARYING)	Национальные символьные типы данных используются для поддержки специальной кодовой страницы, определяемой реализацией.
	CHARACTER LARGE OBJECT (CLOB)	Типы данных *CHARACTER LARGE OBJECT* и *BINARY LARGE OBJECT (BLOB)* относятся к типу *LARGE OBJECT*.
	NATIONAL CHARACTER LARGE OBJECT (NCLOB)	То же, что и *CHARACTER LARGE OBJECT*, но с поддержкой кодовой страницы, определяемой реализацией.
DATALINK	*DATALINK*	Описывает указатель на файл или другой внешний источник данных, не являющийся частью базы данных.
INTERVAL	*INTEVAL*	Описывает набор значений времени или временной интервал.
COLLECTION	*ARRAY* *MULTISET*	*ARRAY* был введен в SQL99, а *MULTISET* был добавлен в SQL2003. *ARRAY* – это упорядоченная коллекция элементов, имеющая ограниченную длину. *MULTISET* – неупорядоченная коллекция, не имеющая ограничений на размер. Элементы *MULTISET* и *ARRAY* должны быть стандартного типа.

Таблица 2.8 (продолжение)

Категория	Примеры типов данных (и сокращенные названия)	Описание
NUMERIC	*INTEGER(INT)* *SMALLINT* *BIGINT* *NUMERIC(p,s)* *DEC[IMAL](p,s)* *FLOAT(p,s)* *REAL* *DOUBLE PRECISION*	Эти типы данных хранят точные (целые и десятичные) или приблизительные (с плавающей точкой) числовые значения. *INT*, *BIGINT* и *SMALLINT* хранят точные числовые значения с фиксированным масштабом и точностью. *NUMERIC* и *DEC* хранят точные числовые значения с настраиваемыми масштабом и точностью. *FLOAT* хранит приблизительное числовое значение с настраиваемой точностью, а точность *REAL* и *DOUBLE PRECISION* фиксирована. Вы можете указать точность (precision) и масштаб (scale) типов *DECIMAL*, *FLOAT* и *NUMERIC* для указания общего числа хранимых цифр и десятичных знаков. *INT*, *SMALLINT*, *DEC* иногда называют *точными типами данных*, а *FLOAT*, *REAL* и *DOUBLE PRECISION* – *приблизительными*.
TEMPORAL	*DATE* *TIME* *TIME WITH TIMEZONE* *TIMESTAMP* *TIMESTAMP WITH TIMEZONE*	Эти типы данных используются для хранения значений времени. *TIME* и *DATE* говорят сами за себя. Типы данных с суффиксом *WITH TIMEZONE* содержат указание на часовой пояс. Тип *TIMESTAMP* используется для хранения более точного момента времени. Эти типы данных часто называют *временными*.
XML	*XML*	Хранит XML данные и может использоваться везде, где может использоваться любой другой стандартный тип данных (например, для столбца таблицы, поля в записи и т. д.). Операции со значениями типа XML предполагают наличие древовидной внутренней структуры данных. Внутренняя структура основана на XML Information Set Recommendation (Infoset), и использует новый внутренний информационный элемент с названием *корневой информационный элемент XML*.

Не каждая платформа поддерживает все типы данных ANSI SQL. В табл. 2.9 последовательно сравниваются типы данных в пяти реализациях. Внимательно читайте сноски, потому что иногда платформой поддерживается тип с определенным именем, но при этом реализация отличается от требований стандарта или от реализации другими разработчиками.



Хотя платформы и поддерживают типы данных с одинаковыми названиями, детали их реализации могут отличаться. Раздел, следующий за таблицей, содержит детальное описание типов данных каждой платформы.

Таблица 2.9. Сравнение типов данных разных платформ

Тип данных вендора	MySQL	Oracle	PostgreSQL	SQL Server	Тип данных SQL2003
BFILE		Да			Нет аналога
BIGINT	Да		Да	Да	*BIGINT*
BINARY	Да			Да	*BLOB*
BINARY_FLOAT		Да			*FLOAT*
BINARY_DOUBLE		Да			*DOUBLE PRECISION*
BIT	Да		Да	Да	Нет аналога
BIT VARAYING, *VARBIT*			Да		Нет аналога
BLOB	Да	Да			*BLOB*
BOOL, *BOOLEAN*	Да		Да		*BOOLEAN*
BOX			Да		Нет аналога
BYTEA			Да		*BLOB*
CHAR, *CHARACTER*	Да	Да	Да	Да	*CHARACTER*
CHAR FOR BIT DATA					Нет аналога
CIDR			Да		Нет аналога
CIRCLE			Да		Нет аналога
CLOB		Да			*CLOB*
CURSOR				Да	Нет аналога
DATALINK					*DATALINK*
DATE	Да	Да	Да	Да	*DATE*
DATETIME	Да			Да	*TIMESTAMP*
DATETIMEOFFSET				Да	*TIMESTAMP*
DATETIME2				Да	*TIMESTAMP WITH TIMEZONE*
DBCLOB					*NCLOB*
DEC, *DECIMAL*	Да	Да	Да	Да	*DECIMAL*
DOUBLE, *DOUBLE PRECISION*	Да	Да	Да[a]	Да	*FLOAT*
ENUM	Да		Да		Нет аналога
FLOAT	Да	Да	Да	Да	*DOUBLE PRECISION*

[a] Синоним для *FLOAT*

Таблица 2.9 (продолжение)

Тип данных вендора	MySQL	Oracle	PostgreSQL	SQL Server	Тип данных SQL2003
FLOAT4			Да[a]		*FLOAT(p)*
FLOAT8			Да[b]		*FLOAT(p)*
GRAPHIC					*BLOB*
GEOGRAPHY				Да	Нет аналога
GEOMETRY				Да	Нет аналога
HYERARCHYID				Да	Нет аналога
IMAGE				Да	Нет аналога
INET			Да		Нет аналога
INT, *INTEGER*	Да	Да	Да	Да	*INTEGER*
INT2			Да		*SMALLINT*
INT4			Да		*INT*, *INTEGER*
INTERVAL			Да		*INTERVAL*
INTERVAL DAY TO SECOND		Да	Да		*INTEVAL DAY TO SECOND*
INTERVAL YEAR TO MONTH		Да	Да		*INTERVAL YEAR TO MONTH*
LINE			Да		Нет аналога
LONG		Да			Нет аналога
LONG VARCHAR					Нет аналога
LONGBLOB	Да				*BLOB*
LONG RAW		Да			*BLOB*
LONG VARGRAPHIC					Нет аналога
LONGTEXT	Да				Нет аналога
LSEG			Да		Нет аналога
MACADDR			Да		Нет аналога
MEDIUMBLOB	Да				Нет аналога
MEDIUMINT	Да				*INT*
MEDIUMTEXT	Да				Нет аналога
MONEY			Да[c]	Да	Нет аналога

[a] Синоним для *REAL*

[b] Синоним для *DOUBLE PRECISION*

[c] Синоним для *DECIMAL(9,2)*

Тип данных вендора	MySQL	Oracle	Postgre-SQL	SQL Server	Тип данных SQL2003
NATIONAL CHARACTER VARYING, *NATIONAL CHAR VARYING*, *NCHAR VARYING*, *NVARCHAR*	Да	Да	Да	Да	*NATIONAL CHARACTER VARYING*
NCHAR, *NATIONAL CHAR*, *NATIONAL CHARACTER*	Да	Да	Да	Да	*NATIONAL CHARACTER*
NCLOB		Да			*NCLOB*
NTEXT, *NATIONAL TEXT*				Да	*NCLOB*
NVARCHAR2(n)		Да			Нет аналога
NUMBER	Да	Да	Да[a]	Да	Нет аналога
NUMERIC				Да	*NUMERIC*
OID			Да		Нет аналога
PATH			Да		Нет аналога
POINT			Да		Нет аналога
POLYGON			Да		Нет аналога
RAW		Да			Нет аналога
REAL	Да	Да	Да	Да	*REAL*
ROWID		Да			Нет аналога
ROWVERSION				Да	Нет аналога
SERIAL, *SERIAL4*	Да[b]		Да		Нет аналога
SERIAL8, *BIGSERIAL*			Да		Нет аналога
SET	Да				Нет аналога
SMALLDATETIME				Да	Нет аналога
SMALLINT	Да	Да	Да	Да	*SMALLINT*
SMALLMONEY				Да	Нет аналога
SQL_VARIANT				Да	Нет аналога
TABLE				Да	Нет аналога
TEXT	Да		Да	Да	Нет аналога
TIME	Да		Да	Да	*TIME*
TIMESPAN					*INTERVAL*
TIMESTAMP	Да	Да	Да	Да[c]	*TIMESTAMP*

[a] Синоним для *DECIMAL*

[b] Синоним для *BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE*

[c] Реализован не как временной тип

Таблица 2.9 (продолжение)

Тип данных вендора	MySQL	Oracle	PostgreSQL	SQL Server	Тип данных SQL2003
TIMESTAMP WITH TIMEZONE, *TIMESTAMPZ*			Да		*TIMESTAMP WITH TIMEZONE*
TIMETZ			Да		*TIMESTAMP WITH TIMEZONE*
TINYBLOB	Да			Да	Нет аналога
TINYINT	Да			Да	Нет аналога
TINYTEXT	Да				Нет аналога
UNIQUEIDENTIFIER				Да	Нет аналога
UROWID		Да			Нет аналога
VARBINARY	Да			Да	*BLOB*
VARCHAR, *CHAR VARYING*, *CHARACTER VARYING*	Да	Да[a]	Да	Да	*CHARACTER VARYING(n)*
VARCHAR2		Да			*CHARACTER VARYING*
VARCHAR FOR BIT DATA					*BIT VARYING*
VARGRAPHIC					*NCHAR VARYING*
YEAR	Да				*TINYINT*
XML			Да	Да	*XML*
XMLTYPE		Да			*XML*

[a] В Oracle рекомендуется использовать *VARCHAR2*

В следующих разделах для каждой платформы перечислены все специфические типы данных, не входящие в SQL2003, указана категория типа данных (если возможно) и приведены необходимые детали.

Типы данных MySQL

MySQL версии 5.1 имеет поддержку пространственных данных, но не как типов данных. Пространственные данные представлены в виде набора классов Геометрической Модели OpenGIS (OpenGIS Geometry Model), поддерживаемой движками MyISAM, InnoDB, NDB и ARCHIVE. Только MyISAM поддерживает и пространственные, и непространственные индексы, остальные движки поддерживают только непространственные индексы.

Числовые типы данных MySQL поддерживают следующие дополнительные атрибуты:

UNSIGNED

Предполагается использование неотрицательных числовых значений (нуля или положительных). Для типов с фиксированной запятой, таких как *DECI-*

MAL и *NUMERIC*, место, используемое обычно для хранения знака числа, может быть использовано для хранения части значения, что немного увеличивает диапазон допустимых значения для типа. (Не существует опции *SIGNED*.)

ZEROFILL

Используется для форматирования при выводе числа в виде строки для указания того, что строку нужно дополнять до максимальной длины нулями, а не пробелами. Атрибут *ZEROFILL* также неявно устанавливает атрибут *UNSIGNED*.

MySQL ограничивает 255 символами максимальную длину при выводе числа в виде строки. Значения большего размера хранятся корректно, но обрезаются до 255 знаков при выводе. Числовые типы с плавающей запятой могут иметь до 30 знаков после запятой.

В следующем списке перечислены все типы данных, поддерживаемые в MySQL. Список включает большую часть из типов SQL2003, несколько дополнительных типов для хранения наборов значений и типы данных для больших двоичных объектов (*BLOB*). Типы данных, расширяющие стандартные типы, включают *TEXT*, *ENUM*, *SET* и *MEDIUMINT*. Специальные атрибуты, выходящие за рамки стандарта ANSI, включают *AUTO_INCREMENT*, *BINARY*, *NULL*, *UNSIGNED* и *ZEROFILL*. Поддерживаются следующие типы:

BIGINT[(n)] [UNSIGNED] [ZEROFILL] (SQL2003 тип: BIGINT)

Хранит знаковые и беззнаковые целые значения. Диапазон знаковых значений от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807. Диапазон беззнаковых значений от 0 до 18 446 744 073 709 551 615. Операции с очень большими (63 бита) числами типа *BIGINT* могут быть неточными из-за погрешностей округления.

BINARY[(n)] (SQL2003 тип: BLOB)

Хранит двоичные массивы байтов (опционально можно указать длину *n*). В остальном похож на тип данных *CHAR*.

BIT, BOOL (SQL2003 тип: отсутствует)

Синоним для *TINYINT*.

BLOB (SQL2003 тип: BLOB)

Хранит до 65535 символов. Возможность индексирования столбцов типа *BLOB* присутствует только в версии 3.23.2 или старше (такой возможности нет больше ни в одной из рассматриваемых платформ). В MySQL тип *BLOB* функционально эквивалентен типу *VARCHAR BINARY* (рассматриваемому позднее) с верхней границей по умолчанию. Значения *BLOB* чувствительны к регистру при сравнениях. *BLOB* отличается от *VARCHAR BINARY* запрещенной возможностью устанавливать значения по умолчанию с помощью атрибута *DEFAULT*. Также столбцы типа *BLOB* нельзя использовать во фразах *GROUP BY* и *ORDER BY*. В зависимости от используемого движка хранения значения *BLOB* могут иногда храниться вне таблиц, в то время как все остальные типы данных (кроме *TEXT*) всегда хранятся в структуре файла, соответствующей таблице.

CHAR(n) [BINARY], CHARACTER(n) [BINARY] (SQL2003 тип:CHARACTER(n))

Содержит символьную строку с фиксированной длиной от 1 до 255 символов. При хранении значения дополняются до указанной длины пробелами, но при извлечении пробелы удаляются, как в типе *VARCHAR* по стандарту SQL2003. Опция *BINARY* позволяет использовать двоичный, а не алфавитный регистронезависимый поиск.

DATE (SQL2003 тип: DATE)

Хранит даты в диапазоне от 01-01-1000 до 31-12-9999 (в качестве разделителя используются кавычки). При отображении MySQL по умолчанию использует формат ГГГГ-ММ-ДД (YYYY-MM-DD), хотя пользователь может указать другой формат.

DATETIME (SQL2003 тип: TIMESTAMP)

Хранит значение даты и времени в диапазоне от 01-01-1000 00:00:00 до 31-12-9999 23:59:59.

DECIMAL[(p[,s])] [ZEROFILL] (SQL2003 тип: DECIMAL(PRECISION,SCALE))

Хранит точные числовые значения в формате строки, на каждую цифру отводится один символ. Точность по умолчанию равна 10, масштаб по умолчанию равен 0.

DOUBLE[(p,s)] [ZEROFILL], DOUBLE PRECISION[(p,s)] [ZEROFILL] (SQL2003 тип: DOUBLE PRECISION)

Хранит числовые значения двойной точности и идентичен типу *FLOAT* во всем, за исключением диапазона: от -1.7976931348623157E+308 до -2.2250738585072014E–308, 0, от 2.2250738585072014E–308 до 1.7976931348623157E+308.

ENUM("значение1", "значение2",..."значениеN") [CHARACTER SET кодовая_страница] [COLLATE схема_упорядочения] (SQL2003 тип: отсутствует)

Содержит значение из задаваемого списка допустимых значений. (Допустимые значения задаются как строки, но хранятся как целые числа). Также может содержать *NULL* или пустую строку в качестве ошибочного значения. Может хранить до 65535 значений.

FLOAT[(p[,s])] [ZEROFILL] (SQL2003 тип: FLOAT(P))

Хранит числовые значения с плавающей запятой в диапазоне от -3.402823466E+38 до -1.175494351E–38 и от 1.175494351E–38 до 3.402823466E+38. *FLOAT* с точностью <=24 или без указания точности называется значением с *одинарной точностью*. В противном случае, с *двойной точностью*. Если указывается только точность, то она может варьироваться от 0 до 53. Если же указываются и точность, и масштаб, то точность может быть до 255, а масштаб может быть до 253. Все вычисления с *FLOAT* выполняются в MySQL с двойной точностью, и могут, так как *FLOAT* – неточный тип, приводить к ошибкам округления.

INT[EGER][(n)] [UNSIGNED] [ZEROFILL] [AUTO_INCREMENT] (SQL2003 тип: INTEGER)

Хранит знаковые или беззнаковые целые значения. При использовании механизма хранения ISAM диапазон знаковых значений от -2 147 483 648 до 2 147 483 647, диапазон беззнаковых значений от 0 до 4 294 967 295. Для других механизмов хранения диапазоны немного отличаются. Опция *AUTO_INCREMENT* доступна для всех целочисленных типов и используется для генерации уникальных идентификаторов вставляемых строк. (Для дополнительной информации по *AUTO_INCREMENT* обратитесь к разделу «Оператор *CREATE/ALTER DATABASE*» в главе 3.)

LONGBLOB (SQL2003 тип данных: BLOB)

Хранит данные *BLOB* длиной до 4 294 967 295 байт. Помните, что в некоторых клиент-серверных протоколах такой размер данных может не поддерживаться.

LONGTEXT [CHARACTER SET кодовая_страница] [COLLATE схема_упорядочения] (SQL2003 тип данных: CLOB)

Хранит данные типа *TEXT* длиной до 4 294 967 295 символов (или меньше при многобайтной кодовой странице). Помните, что в некоторых клиент-серверных протоколах такой размер данных может не поддерживаться.

MEDIUMBLOB (SQL2003 тип данных: отсутствует)

Хранит данные *BLOB* длиной до 16 777 215 байт. Первые три байта используются для хранения общей длины значения.

MEDIUMINT[(n)] [UNSIGNED] [ZEROFILL] (SQL2003 тип: отсутствует)

Хранит знаковые или беззнаковые целые значения. Диапазон знаковых значений от -8 388 608 до 8 388 608. Диапазон беззнаковых значений от 0 до 16 777 215.

MEDIUMTEXT [CHARACTER SET кодовая_страница] [COLLATE схема_упорядочения] (SQL2003 тип: отсутствует)

Хранит данные типа *TEXT* длиной до 16 777 215 символов (или меньше при многобайтной кодовой странице). Первые три байта используются для хранения общей длины значения.

NCHAR(n) [BINARY], [NATIONAL] CHAR(n) [BINARY] (SQL2003 тип: NCHAR(n))

Синонимы для *CHAR*. Типы *NCHAR* поддерживают *UNICODE* начиная с версии MySQL 4.1.

NUMERIC(p,s) (SQL2003 тип: DECIMAL(p,s))

Синоним для *DECIMAL*.

NVARCHAR(n) [BINARY], [NATIONAL] VARCHAR(n) [BINARY], NATIONAL CHARACTER VARYING(n) [BINARY] (SQL2003 тип данных: NCHAR VARYING)

Синонимы для *VARYING [BINARY]*. Содержат значения переменной длины с ограничением до 255 символов. Если не указано ключевое слово *BINARY*, то значения хранятся и сравниваются без учета регистра.

REAL(p,s) (SQL2003 тип данных: REAL)

Синоним для *DOUBLE PRECISION*.

SERIAL

Синоним для BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE.

SET("значение1", "значение2",..."значениеN") [CHARACTER SET кодовая_страница] [COLLATE схема_упорядочения] (SQL2003 тип данных: отсутствует)

Содержит от 0 до 64 значений типа *CHAR* из списка допустимых значений.

SMALLINT[(n)] [UNSIGNED] [ZEROFILL] (SQL2003 тип: SMALLINT)

Содержит знаковые или беззнаковые целые числа. Диапазон знаковых чисел от -32768 до 32767. Диапазон беззнаковых чисел от 0 до 65535.

TEXT (SQL2003 тип: отсутствует)

Содержит до 65535 символов. Тип данных *TEXT* иногда хранится отдельно от остальных данных таблицы, в зависимости от используемого механизма хранения, в то время как данные всех остальных типов (кроме *BLOB*) хранятся в структуре файла, соответствующей таблице. Тип *TEXT* функционально эквивалентен (за исключением допустимой длины) типу *VARCHAR* с неуказанной верхней границей. Сравнения производятся без учета регистра. *TEXT* отличается от *VARCHAR* также тем, что не допускает использование значений по умолчанию, задаваемых опцией *DEFAULT*. Столбцы типа *TEXT* не могут использоваться во фразах *GROUP BY* и *ORDER BY*. Возможность индексирования столбцов типа *TEXT* появилась в MySQL с версии 3.23.2.

TIME (SQL2003 тип: отсутствует)

Хранит значение времени в диапазоне от '-838:59:59' до '838:59:59' в формате 'чч:мм:сс' ('HH:MM:SS'). Может присваиваться как строка или число.

TIMESTAMP (SQL2003 тип: TIMESTAMP)

Хранит временные значения в диапазоне от '1970-01-01 00:00:01' вплоть до 2038 года. Значение хранится как количество секунд от '1970-01-01 00:00:01'. Всегда выводится в формате 'гггг-мм-дд чч:мм:сс' ('YYYY-MM-DD HH:MM:SS').

TINYBLOB (SQL2003 тип: BLOB)

Хранит значения *BLOB* размером до 255 байт. Первый байт задействован для хранения общего размера значения.

TINYTEXT (SQL2003 тип: отсутствует)

Хранит значения типа *TEXT* длиной до 255 символов (или меньше при многобайтной кодовой странице). Первый байт задействован для хранения общего размера значения.

VARBINARY(n) (SQL2003 тип: BLOB)

Хранит двоичные байтовые массивы переменной длины, не большей *n*. В остальном тип эквивалентен типу *VARCHAR*.

YEAR (SQL2003 datatype: none)

Хранит год в формате двух или четырех (по умолчанию) цифр. Формат из двух цифр допускает значения от '70' до '69', обозначая соответственно 1970 и 2069. Формат из четырех цифр допускает значения в диапазоне от 1901 до 2155 и 0. Значения типа *YEAR* всегда выводятся в формате 'гггг' (YYYY), но могут присваиваться как строки и числа.

Типы данных Oracle

Как вы увидите из содержания этого раздела, в Oracle поддерживается множество типов данных, включая большинство типов SQL3 и некоторых специальных типов. Специальные типы данных, однако, часто требуют установки дополнительных компонент. К примеру, в Oracle поддерживаются пространственные типы данных, но только при установленной опции Oracle Spatial (за исключением типа данных SDO_GEOMETRY, который поддерживается в любой инсталляции). Пространственные типы данных, такие как *SDO_GEOMETRY*, *SDO_TOPO_GEOMETRY* и *SDO_GEORASTER* выходят за рамки этой книги. За подробными сведениями об этих типах обращайтесь к документации по Oracle Spatial.

Типы данных опции Oracle Multimedia являются объектными типами, похожими на классы C++ и Java. Набор типов данных Oracle Multimedia включает *ORDAudio*, *ORDImage*, *ORDVideo*, *ORDDoc*, *ORDDicom*, *SI_Stillimage*, *SI_Color*, *SI_AverageColor*, *SI_ColorHistogram*, *SI_PositionalColor*, *SI_Texture*, *SI_FeatureList* и *ORDImageSignature*.

Oracle также поддерживает типы данных 'Any Type'. Эти типы используются в параметрах функций или столбцах таблиц тогда, когда неизвестен настоящий тип данных. Набор этих типов данных включает *ANYTYPE*, *ANYDATA* и *ANYDATASET*.

Вот полный список поддерживаемых в Oracle типов данных:

BFILE (SQL2003 mun: DATALINK)

Содержит указатель на значение *BLOB* максимальным размером 4 Гб (начиная с 10g фактически размер ограничен максимальным поддерживаемым размером файла в операционной системе), хранящееся вне базы данных, но на том же локальном сервере. *BLOB*-значение может читаться в базу данных, но не может записываться из базы данных. При удалении строки, содержащей значение типа *BFILE*, удаляется только указатель, а внешний файл остается.

BINARY_DOUBLE (SQL2003 mun: FLOAT)

Хранит 64-битное числовое значение с плавающей запятой.

BINARY_FLOAT (SQL2003 mun: FLOAT)

Хранит 32-битное числовое значение с плавающей запятой.

BLOB (SQL2003 mun: BLOB)

Содержит большой двоичный объект (*BLOB*) с максимальным размером от 8 Тб до 128 Тб в зависимости от размера блока данных. В Oracle большие бинарные объекты (*BLOB*, *CLOB* и *NCLOB*) имеют следующие ограничения:

- Не могут извлекаться удаленно
- Не могут храниться в кластере
- Не могут использоваться в массиве *VARRAY*
- Не могут использоваться во фразах *GROUP BY* и *ORDER BY*
- Не могут использоваться в агрегирующих функциях
- Не могут использоваться в запросах с *DISTINCT*, *UNIQUE* или в условиях соединения
- Не могут использоваться в выражениях *ANALYZE...COMPUTE* и *ANALYZE...ESTIMATE*
- Не могут быть частью первичного ключа или индекса
- Не могут использоваться во фразе *UPDATE OF* при создании триггера на *UPDATE*.

CHAR(n [BYTE | CHAR]), CHARACTER(n [BYTE | CHAR]) (SQL2003 тип: CHARACTER(n))

Хранит символьное значение фиксированной длины до 2000 байт. Ключевое слово *BYTE* говорит о том, что размер указывается в байтах. *CHAR* используется для указания размера в символах.

CLOB (SQL2003 тип: CLOB)

Содержит большой символьный объект (*CLOB*) с максимальным размером от 8 Тб до 128 Тб в зависимости от размера блока данных. Список ограничений при использовании *CLOB* приводится в описании типа *BLOB*.

DATE (SQL2003 тип: DATE)

Хранит значение даты и времени в диапазоне от '4712-01-01 00:00:00 до н.э.' до '9999-12-31 23:59:59 н.э.'.

DECIMAL(p,s) (SQL2003 тип: DECIMAL(p,s))

Синоним для типа *NUMBER* с указанием значений масштаба и точности.

DOUBLE PRECISION (SQL2003 тип: DOUBLE PRECISION)

Хранит числовое значение с плавающей запятой с двойной точностью. То же самое, что *FLOAT(126)*.

FLOAT(n) (SQL2003 тип: FLOAT(n))

Хранит числовое значение с плавающей запятой с двоичной точностью до 126.

INTEGER(n) (SQL2003 тип: INTEGER)

Хранит знаковые и беззнаковые целые значения с точностью до 38 знаков. *INTEGER* является синонимом для *NUMBER*.

INTERVAL DAY(n) TO SECOND(x) (SQL2003 тип: INTERVAL)

Хранит временной интервал в днях, часах, минутах и секундах; *n* задает число цифр для хранения дней (допустимо от 0 до 9, по умолчанию – 2), *x* задает число цифр для хранения дробной части секунд (от 0 до 9, по умолчанию – 6).

INTERVAL YEAR(n) TO MONTH (SQL2003 тип: INTERVAL)

Хранит временной интервал в годах и месяцах, где n задает число цифр для хранения годов (от 0 до 9, по умолчанию – 2).

LONG (SQL2003 тип: отсутствует)

Хранит символьное значение переменной длины размером не более 2 Гб. Обратите внимание: Oracle не планирует долгосрочной поддержки этого типа данных. Так что по возможности используйте вместо *LONG* другие типы данных, например *CLOB*. Тип *LONG* не рекомендован к использованию.

LONG RAW (SQL2003 тип: отсутствует)

Хранит бинарные данные переменной длины размером не более 2 Гб. *LONG RAW* и *RAW* обычно используются для хранения графики, звуков, документов и других больших структур данных. Вместо *LONG RAW* предпочтительнее использовать тип *BLOB*, так как он имеет меньше ограничений. Тип *LONG RAW* не рекомендован к использованию.

NATIONAL CHARACTER VARYING(n), NATIONAL CHAR VARYING(n), NCHAR VARYING(n) (SQL2003 тип: NCHAR VARYING (n))

Синоним для типа *NVARCHAR2*.

NCHAR(n), NATIONAL CHARACTER(n), NATIONAL CHAR(n) (SQL2003 тип: NATIONAL CHARACTER)

Содержит символьные данные в кодировке *UNICODE* размером от 1 до 2000 байт. Значение по умолчанию – 1 байт.

NCLOB (SQL2003 тип: NCLOB)

Представляет собой значение *CLOB* с поддержкой многобайтных и *UNICODE* кодировок. Список ограничений при использовании *NCLOB* приводится в описании типа *BLOB*.

NUMBER(p,s), NUMERIC(p,s) (SQL2003 тип: NUMERIC(p,s))

Хранит числовые значения с точностью от 1 до 38 и масштабом от -84 до 127.

NVARCHAR2(n) (SQL2003 тип: отсутствует)

Предпочтительный тип для хранения в Oracle символьных данных переменной длины в кодировке *UNICODE*. Размер значения может быть от 1 до 4000 байт.

RAW(n) (SQL2003 тип: отсутствует)

Хранит бинарные данные переменной длины размером до 2000 байт.

REAL (SQL2003 тип: REAL)

Хранит числовое значение с плавающей запятой с одинарной точностью. То же самое, что *FLOAT(63)*.

ROWID (SQL2003 тип: отсутствует)

Содержит уникальный идентификатор в формате base64 любой строки в базе данных. Часто используется в паре с псевдостолбцом *ROWID*.

SMALLINT (SQL2003 тип: SMALLINT)

Синоним для *INTEGER*.

TIMESTAMP(n) {[WITH TIME ZONE] | [WITH LOCAL TIME ZONE]} (SQL2003 тип: TIMESTAMP[WITH TIME ZONE])

Хранит значение даты и времени, n используется для указания числа знаков при хранения дробной части секунд (от 0 до 9, по умолчанию – 6). Если указа-

но *WITH TIME ZONE*, то сохраняется также выбранный часовой пояс (по умолчанию часовой пояс сессии пользователя), и затем значение возвращается в этом часовом поясе. При использовании *WITH LOCAL TIME ZONE* данные перед сохранением преобразуются во временную зону сервера, а при извлечении преобразуются обратно в часовой пояс извлекающей сессии.

UROWID[(n)] (SQL2003 тип: отсутствует)

Хранит в формате base-64 логический адрес строки в таблице. По умолчанию позволяет записать до 4000 байт данных, но вы можете по желанию ограничить максимальную длину любым значением до 4000.

VARCHAR(n), CHARACTER VARYING(n), CHAR VARYING(n) (SQL2003 тип: CHARACTER VARYING(n))

Содержит символьные значения переменной длины размером от 1 до 4000 байт.

Oracle не рекомендует использовать *VARCHAR* и уже долгие годы настаивает на использовании *VARCHAR2*.

VARCHAR2(n [BYTE|CHAR]) (SQL2003 тип: CHARACTER VARYING(n))

Хранит символьное значение переменной длины до 4000 байт (определяется *n*). Ключевое слово *BYTE* говорит о том, что размер указывается в байтах. *CHAR* используется для указания размера в символах. Если вы используете *CHAR*, то количество символов все равно трансформируются в определенное количество байт, которое также не должно быть больше 4000 байт.

XMLTYPE (SQL2003 тип: XML)

Хранит в базе данных Oracle данные в формате *XML*. Для доступа к данным используются выражения XPath, а также набор встроенных XPath функций, *XML* функций и PL/SQL пакетов. Тип данных *XMLTYPE* является системным, и потому может быть использован для параметров функций или столбцов таблиц. При использовании в таблице данные могут быть сохранены в формате *CLOB* или в объектно-реляционном виде.

Типы данных PostgreSQL

PostgreSQL поддерживает большую часть типов данных SQL2003, а также чрезвычайно богатый набор типов для хранения геометрических и пространственных данных. PostgreSQL содержит множество полезных операторов и функций для работы с геометрическими данными, включая такие функции, как вращение, нахождение пересечений, масштабирование. Также поддерживаются дополнительные варианты стандартных типов данных, занимающие меньше места при хранении, чем их полноценные аналоги. К примеру, PostgreSQL предлагает несколько вариантов типа *INTEGER* для хранения больших и маленьких значений и занимающих, соответственно, больше или меньше дискового пространства. Вот перечень поддерживаемых типов данных:

BIGINT, INT8 (SQL2003 тип: отсутствует)

Хранит знаковые или беззнаковые 8-байтные целые числа в диапазоне от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807.

BIGSERIAL

Смотрите описание типов *SERIAL*.

BIT (SQL2003 тип: BIT)

Хранит битовую строку фиксированной длины.

BIT VARYING(n), VARBIT(n) (SQL2003 тип: BIT VARYING)

Хранит битовую строку переменной длины, максимальная длина задается параметром *n*.

BOOL, BOOLEAN (SQL2003 тип: BOOLEAN)

Хранит булевы значения (истина/ложь/неизвестно). Для значений желательно использовать ключевые слова *TRUE* и *FALSE*, хотя PostgreSQL поддерживает следующие литералы для обозначения истины: *TRUE*, *t*, *true*, *y*, *yes* и *1*. Для ложного значения используются литералы: *FALSE*, *f*, *false*, *n*, *no* и *0*.

BOX((x1, y1), (x2, y2)) (SQL2003 тип: отсутствует)

Хранит координаты прямоугольника на плоскости. Значение хранится в 32 байтах в формате *((x1, y1), (x2, y2))*, задавая противоположные вершины прямоугольника (верхнюю правую и нижнюю левую соответственно). Внешние скобки опциональны.

BYTEA (SQL2003 тип: BINARY LARGE OBJECT)

Содержит неструктурированные двоичные данные. Часто используется для хранения изображений, звуков, документов. При хранении требует пространства в количестве 4 байта плюс размер самих двоичных данных.

CHAR(n), CHARACTER(n) (SQL2003 тип: CHARACTER(n))

Хранит строки фиксированной длины, дополняемые пробелами до размера *n*. При попытке вставки строки большего размера произойдет ошибка (если только лишние символы не являются пробелами и могут быть отброшены, так что длина строки станет меньше либо равной *n*).

CIDR(x.x.x.x/y) (SQL2003 тип: отсутствует)

Используется для хранения сетевого адреса протокола IPv4, занимает 12 байт. Диапазон значений совпадает с множеством допустимых сетевых адресов IPv4. Данные типа *CIDR* хранятся в формате *x.x.x.x/y*, где иксы представляют ip-адрес, а *y* задает количество бит в сетевой маске. *CIDR* не допускает ненулевых битов справа от нулевых битов в сетевой маске.

CIRCLE(x, y, r) (SQL2003 тип: отсутствует)

Описывает окружность на плоскости. Значение хранится в 24 байтах и представляется как *(x,y,r)*: *x* и *y* задают координаты центра окружности, *r* задает величину радиуса. По желанию можно использовать круглые или угловые скобки в качестве разделителей значений *x*, *y* и *r*.

DATE (SQL2003 тип: DATE)

Хранит дату (год, месяц, день) без указания времени. Для хранения используется 4 байта. Диапазон значений от '4714 до н.э.' до '32767 н.э.' Минимальное разрешение типа, естественно, равно одному дню.

DECIMAL[(p,s)], NUMERIC[(p,s)] (SQL2003 тип: DECIMAL(p,s), NUMERIC(p,s))

Хранит точные числовые значения с точностью p (от 0 до 9) и масштабом s (от 0 до бесконечности).

FLOAT4, REAL (SQL2003 тип: FLOAT(p))

Хранит числовые значения с плавающей запятой с точностью от 0 до 8 знаков и 6 знаками после запятой.

FLOAT8, DOUBLE PRECISION (SQL2003 тип: FLOAT(p), 7 <= p < 16)

Хранит числовые значения с плавающей запятой с точностью от 0 до 16 знаков и 15 знаками после запятой.

INET(x.x.x.x/y) (SQL2003 тип: отсутствует)

Используется для хранения сетевого адреса протокола IPv4, занимает 12 байт. Диапазон значений совпадает с множеством допустимых сетевых адресов IPv4. Данные хранятся в формате *x.x.x.x/y*, где иксы представляют ip-адрес, а *y* задает количество бит в сетевой маске. Сетевая маска по умолчанию равна 32. В отличие от *CIDR*, тип *INET* допускает ненулевые биты справа от нулевых битов в сетевой маске.

INTEGER, INT, INT4 (SQL2003 тип: INTEGER)

Хранит знаковые или беззнаковые 4-байтные целые числа в диапазоне от -2 147 483 648 до 2 147 483 647.

INTERVAL(p) (SQL2003 тип: отсутствует)

Используется для хранения интервалов времени в диапазоне от -178 000 000 до 178 000 000 лет, при хранении занимает 12 байт. Максимальная точность хранения – 1 микросекунда. Этот тип не соответствует стандарту ANSI, требующему именования типа *INTERVAL YEAR TO MONTH*.

LINE((x1, y1), (x2, y2)) (SQL2003 тип: отсутствует)

Хранит бесконечную прямую линию на плоскости. Значение занимает 32 байта и представляется в виде *((x1, y1), (x2, y2))*, задающем координаты двух точек, через которые проходит прямая. Внешние скобки опциональны.

LSEG((x1, y1), (x2, y2)) (SQL2003 тип: отсутствует)

Хранит отрезок на плоскости, включающий конечные точки. Значение занимает 32 байта и представляется в виде *((x1, y1), (x2, y2))*, задающем координаты начальной и конечной точки соответственно. Внешние скобки опциональны. Кстати, отрезок – это то, что обычно называют линией. Например, линии игрового поля на самом деле являются отрезками.

Если придерживаться точных геометрических терминов, то прямая – это бесконечная линия, не имеющая конечных точек ни с одной стороны, а отрезок – это часть прямой, ограниченная двумя точками. В PostgreSQL есть типы данных для обоих понятий, но функционально эти типы эквивалентны.

MACADDR (SQL2003 тип: отсутствует)

Хранит MAC-адрес сетевой карты компьютера, занимает 6 байт. *MACADDR* допускает различные представления значений, используемые производителями, например:

08002B:010203
08002B-010203
0800.2B01.0203
08-00-2B-01-02-03
08:00:2B:01:02:03

MONEY, DECIMAL(9,2) (SQL2003 тип: отсутствует)

Хранит денежное значение в диапазоне от -21 474 83.48 до 21 474 836,47.

NUMERIC[(p,s)], DECIMAL[(p,s)] (SQL2003 тип: отсутствует)

Хранит точные числовые значения с точностью *p* и масштабом *s*.

OID (SQL2003 тип: отсутствует)

Хранит уникальные идентификаторы объектов.

PATH((x1, y1), ...(xn, yn)), PATH[(x1, y1), ... (xn, yn)] (SQL2003 тип: отсутствует)

Используется для хранения замкнутых и незамкнутых ломаных линий. Значения представляются как *[(x1, y1), ... (xn, yn)]* и занимают *4+32*n* байт. Каждое значение *(x,y)* задает координаты точки отрезка ломаной. Линия называется незамкнутой, если начало ее первого отрезка не совпадает с концом последнего отрезка, в противном случае ломаная называется замкнутой. Для замкнутых ломаных используются круглые скобки, для незамкнутых – квадратные.

POINT(x, y) (SQL2003 тип: отсутствует)

Хранит координаты точки на плоскости, занимает 16 байт. Значения представляются как *(x, y)*. Точка является основой всех остальных геометрических типов данных в PostgreSQL. Скобки являются необязательными.

POLYGON((x1, y1), ...n) (SQL2003 тип: отсутствует)

Используется для хранения многоугольников. Значения представляются как *[(x1, y1), ... (xn, yn)]* и занимают *4+32*n* байт. Каждое значение *(x,y)* задает координаты вершины многоугольника. По-сути является тем же, что и тип *PATH* при хранении замкнутых ломаных.

SERIAL, SERIAL4 (SQL2003 тип: отсутствует)

Хранит уникальный, автоматически увеличивающийся идентификатор, используемый для индексирования и внешних ключей. Занимает до 4 байт данных (значения в диапазоне от 0 до 2 147 483 647). Таблицы с полями этого типа удаляются следующим образом: сначала выполняется команда *DROP SEQUENCE*, а потом *DROP TABLE*.

SERIAL8, BIGSERIAL (SQL2003 тип: отсутствует)

Хранит уникальный, автоматически увеличивающийся идентификатор, используемый для индексирования и внешних ключей. Занимает до 8 байт данных (значения в диапазоне от 0 до 9 223 372 036 854 775 807). Таблицы с по-

лями этого типа удаляются следующим образом: сначала выполняется команда *DROP SEQUENCE*, а потом *DROP TABLE*.

SMALLINT (SQL2003 тип: SMALLINT)

Хранит знаковые или беззнаковые 2-байтные целые числа в диапазоне от -32768 до 32767. Так же для этого типа есть синоним *INT2*.

TEXT (SQL2003 тип: CLOB)

Используется для хранения больших текстовых данных переменной длины размером до 1 Гб. Данные типа *TEXT* в PostgreSQL автоматически сжимаются, поэтому место, используемое для хранения, может быть меньше размера исходных данных.

TIME[(p)] [WITHOUT TIME ZONE | WITH TIME ZONE] (SQL2003 тип: TIME)

Используется для хранения времени без указания часового пояса (занимает 8 байт) или с указанием часового пояса сервера СУБД (занимает 12 байт). Диапазон допустимых значений от 00:00:00.00 до 23:59:59.99, точность хранения равна одной микросекунде. Имейте в виду, что часовой пояс на большинстве Unix-систем доступен только для дат с 1902 до 2038 года.

TIMESTAMP[(p)] [WITHOUT TIME ZONE | WITH TIME ZONE] (SQL2003 тип: TIMESTAMP [WITH TIME ZONE | WITHOUT TIME ZONE])

Используется для хранения даты и времени без указания или с указанием часового пояса сервера СУБД. Диапазон допустимых значений от '4317 г. до н.э.' до '1465001 г. н.э.', точность хранения равна одной микросекунде, для хранения используется 8 байт. Имейте в виду, что часовой пояс на большинстве Unix-систем доступен только для дат с 1902 до 2038 года.

TIMETZ (SQL2003 тип: TIME WITH TIME ZONE)

Хранит время с указанием часового пояса.

VARCHAR(n), CHARACTER VARYING(n) (SQL2003 тип: CHARACTER VARYING(n))

Хранит символьные данные переменной длины размером не более *n*. Пробелы в конце строки не сохраняются.

Типы данных SQL Server

Microsoft SQL Server поддерживает большинство из стандартных типов SQL2003, а также некоторые дополнительные типы, например UNIQUEIDENTIFIER, для уникальной идентификации строк в таблицах в нескольких базах данных и даже на разных серверах. Эти типы данных включены для поддержки философии Microsoft относительно аппаратного обеспечения: горизонтальное масштабирование (развертывание системы на множестве недорогих Intel-серверов) вместо вертикального (развертывание на одном большом высокопроизводительном Unix-сервере или Microsoft Windows Datacenter Server).

У типа данных DATETIME есть одна интересная особенность: временные типы данных поддерживают даты начиная с 1753 года, вы не сможете сохранить более раннюю дату ни в одном из временных

типов. Почему? Объяснение следующее: англоговорящий мир начал использовать Григорианский календарь в 1753 году, а конвертирование дат из более раннего, Юлианского календаря может быть сопряжено с определенными трудностями.

В SQL Server поддерживаются следующие типы данных:

BIGINT (SQL2003 тип: BIGINT)

Хранит знаковые и беззнаковые целые значения в диапазоне от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807, занимает 8 байт. Поддерживает свойство *IDENTITY* (описание приводится в комментарии к типу *INT*).

BINARY[(n)] (SQL2003 тип: BLOB)

Хранит бинарное значение фиксированной длины размером от 1 до 8000 байт. При хранении занимает n+4 байт.

BIT (SQL2003 тип: BOOLEAN)

Хранит 1, 0 или NULL («значение неизвестно»). До восьми столбцов типа *BIT* в одной таблицы будут занимать 1 байт места при хранении, на каждые последующие 8 столбцов *BIT* приходится еще по одному байту. По столбцам *BIT* нельзя создавать индексы.

CHAR[(n)], CHARACTER[(n)] (SQL2003 тип: CHARACTER(n))

Хранит символьные данные фиксированной длины от 1 до 8000 символов. Все свободное пространство в конце строки по умолчанию дополняется пробелами (эту опцию можно отключить). При хранении занимает *n* байт.

CURSOR (SQL2003 тип: отсутствует)

Специальный тип данных, используемый для использования курсора в переменной или выходном параметре хранимой процедуры. Не может использоваться в операторе *CREATE TABLE*. Значению типа *CURSOR* всегда можно присвоить NULL.

DATE (SQL2003 тип: DATE)

Хранит даты в диапазоне от 1 января 0001 года до 31 декабря 9999 года.[1]

DATETIME (SQL2003 тип: TIMESTAMP)

Хранит дату и время в диапазоне от 1753-01-01 00:00:00 до 9999-12-31 23:59:59. Занимает 8 байт.

DATETIME2 (SQL2003 тип: TIMESTAMP)

Хранит даты в диапазоне от 1 января 0001 года до 31 декабря 9999 года с точностью в 100 наносекунд.

DATETIMEOFFSET (SQL2003 тип: TIMESTAMP)

Хранит даты в диапазоне от 1 января 0001 года до 31 декабря 9999 года с точностью в 100 наносекунд, включает информацию о часовом поясе. Занимает 10 байт.

---

[1] Этот тип данных относится к новым временным типам данных, которые поддерживают расширенный диапазон дат. – *Прим. науч. ред.*

DECIMAL(p,s), DEC(p,s), NUMERIC(p,s) (SQL2003 тип: DECIMAL(p,s), NUMERIC(p,s))

Хранит числовые значения с записью до 38 цифр. Значения p и s задают, соответственно, точность и масштаб. Масштаб по умолчанию равен 0. От заданной точности зависит количество байт, используемое для хранения значения:

Точность в пределах 1–9 требует 5 байт

Точность в пределах 10–19 требует 9 байт

Точность в пределах 20–28 требует 13 байт

Точность в пределах 29–38 требует 17 байт

Поддерживает свойство *IDENTITY* (описание приводится в комментарии к типу *INT*).

DOUBLE PRECISION (SQL2003 тип: отсутствует)

Синоним для типа *FLOAT(53)*.

FLOAT[(n)] (SQL2003 тип: FLOAT, FLOAT(n))

Хранит числа с плавающей запятой в диапазоне от -1.79E+308 до 1.79E+308. Точность, заданная параметром *n*, может быть в пределах от 1 до 53. Для хранения 7 разрядов при точности от 1 до 24 занимает 4 байта. Большая точность требует 8 байт.

HIERARCHYID (SQL2003 тип: отсутствует)

Используется для хранения в реляционных данных иерархии или древовидной структуры. Для хранения HIERARCHYID обычно используется до 5 байт, хотя в некоторых случаях может использоваться и больше. За дополнительной информацией об этом типе данных обращайтесь к документации разработчика.

IMAGE (SQL2003 тип: BLOB)

Хранит бинарные данные переменной длины размером до 2 147 483 647 байт. Этот тип данных широко применяется для хранения графики, аудиоданных и файлов (например, документов Word и таблиц Excel). Есть некоторые особенности при работе с данными типа *IMAGE*: на использование столбцов с типами *TEXT* и *IMAGE* наложено множество ограничений. Обратитесь к описанию типа *TEXT* за списком операторов и функций, применяемых при работе с *IMAGE*.

INT [IDENTITY [(начальное_значение, шаг)]] (SQL2003 тип: INTEGER)

Хранит знаковые и беззнаковые целочисленные значения в диапазоне от -2 147 483 648 до 2 147 483 647, занимает 4 байта. Все целочисленные типы данных и типы *DECIMAL* поддерживают опцию *IDENTITY*, используемую для генерации автоматически возрастающих идентификаторов строк. За дополнительной информацией обращайтесь к разделу «Оператор *CREATE/ALTER DATABASE*» главы 3.

MONEY (SQL2003 тип: отсутствует)

Хранит денежные значения в диапазоне от -922 337 203 685 477.5808 до 922 337 203 685 477.5807, занимает 8 байт.

NCHAR(n), NATIONAL CHAR(n), NATIONAL CHARACTER(n) (SQL2003 тип: NATIONAL CHARACTER(n))

Хранит данные фиксированной длины в формате UNICODE размером до 4000 символов. Количество занимаемых байт равно *2*n*.

NTEXT, NATIONAL TEXT (SQL2003 тип: NCLOB)

Хранит текстовые данные в формате UNICODE размером до 1 073 741 823 символов. Для получения информации об используемых с типом функциях и операторах смотрите описание типа *TEXT*.

NUMERIC(p,s) (SQL2003 тип: DECIMAL(p,s))

Синоним для *DECIMAL*. Поддерживает свойство *IDENTITY* (описание приводится в комментарии к типу *INT*).

NVARCHAR(n), NATIONAL CHAR VARYING(n), NATIONAL CHARACTER VARYING(n) (SQL2003 тип: NATIONAL CHARACTER VARYING(n))

Хранит символьные данные переменной длины в формате UNICODE размером до 4000 символов. Количество занимаемых байт равно удвоенному количеству хранимых символов (*2*n*) плюс 2 байта. Системная настройка *SET ANSI_PADDING* всегда включена для типов *NCHAR* и *NVARCHAR*.

REAL, FLOAT(24) (SQL2003 тип: REAL)

Хранит числа с плавающей запятой в диапазоне от -3.40E+38 до 3.40E+38, занимает 4 байта. Тип *REAL* функционально эквивалентен типу *FLOAT(24)*.

ROWVERSION (SQL2003 тип: отсутствует)

Хранит уникальное в пределах базы данных значение, увеличивающееся при каждом изменении записи в таблице. В предыдущих версиях называлось *TIMESTAMP*.

SMALLDATETIME (SQL2003 тип: отсутствует)

Хранит дату и время в диапазоне от '1900-01-01 00:00' до '2079-06-06 23:59' с точностью до минуты. (При значениях до 29.998 секунды округляются в меньшую сторону, при больших значениях – в большую сторону). При хранении занимает 4 байта.

SMALLINT (SQL2003 тип: SMALLINT)

Хранит знаковые и беззнаковые целочисленные значения в диапазон от -32768 до 32767, занимает 2 байта. Поддерживает свойство *IDENTITY* (описание приводится в комментарии к типу *INT*).

SMALLMONEY (SQL2003 тип: отсутствует)

Хранит денежные значения в диапазоне от -214748.3648 до 214748.3647, занимает 4 байта.

SQL_VARIANT (SQL2003 тип: отсутствует)

Хранит данные любого другого типа, за исключением *TEXT*, *NTEXT*, *ROWVERSION* и самого *SQL_VARIANT*. Поддерживает хранение до 8016 байт, может принимать значение NULL, а также допускает установку значения по умолчанию при помощи *DEFAULT*. Используется в столбцах таблиц, параметрах, переменных и возвращаемых значениях хранимых процедур и функций.

TABLE (SQL2003 тип: отсутствует)

Специальный тип данных для сохранения результата запроса для дальнейшей обработки. Используется исключительно при процедурной обработке и не может использоваться в операторе *CREATE STATEMENT*. Этот тип данных позволяет приложениям обходиться без использования временных таблиц. Уменьшает необходимость рекомпиляции хранимых процедур, увеличивая тем самым скорость выполнения хранимых процедур и пользовательских функций.

TEXT (SQL2003 тип: CLOB)

Хранит большие наборы текстовых данных с размером до 2 147 483 647 байт. Типы данных *TEXT* и *IMAGE* сложнее в использовании, чем, скажем, тип *VARCHAR*. К примеру, по столбцу с типом *TEXT* или *IMAGE* нельзя создать индекс. Работа со значениями этих типов осуществляется с помощью функций *DATALENGTH*, *PATINDEX*, *SUBSTRING*, *TEXTPTR* и *TEXTVALID*, а также команд *READTEXT*, *SET TEXTSIZE*, *UPDATETEXT* и *WRITETEXT*.

TIMESTAMP (SQL2003 тип: TIMESTAMP)

Хранит автоматически генерируемое число, гарантированно уникальное в пределах базы данных. Соответственно, этот тип отличается от типа *TIMESTAMP* в ANSI. Занимает 8 байт. Сейчас предпочтительнее вместо типа *TIMESTAMP* использовать *ROWVERSION* для отслеживания изменений записей.

TIME (SQL2003 тип: TIME)

Хранит время суток в 24-часовом формате без указания часового пояса с точностью в 100 наносекунд. Занимает 5 байт.

TINYINT (SQL2003 тип: отсутствует)

Хранит беззнаковые целые числа в интервале от 0 до 255, занимает 1 байт. Поддерживает свойство *IDENTITY* (описание приводится в комментарии к типу *INT*).

UNIQUEIDENTIFIER (SQL2003 тип: отсутствует)

Представляет собой идентификатор, уникальный в пределах всех таблиц всех баз данных на всех серверах. Значение имеет формат *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*, где каждый *x* обозначает шестнадцатеричную цифру (от 0 до 9 или от A до F). Поддерживаются только операторы сравнения и проверки на равенство *NULL*. Столбцы с типом *UNIQUEIDENTIFIER* поддерживают использование ограничений и свойств, за исключением *IDENTITY*.

VARBINARY[(n)] (SQL2003 тип: BLOB)

Используется для хранения двоичных данных переменной длины размером до 8000 байт. Используемое для хранения пространство равно размеру данных + 2 байта.

VARCHAR[(n)], CHAR VARYING[(n)], CHARACTER VARYING[(n)] (SQL2003 тип: CHARACTER VARYING(n))

Хранит символьные данные переменной длины размером от 1 до 8000 байт. Необходимое для хранения дисковое пространство определяется размером вставляемых данных, а не *n*.

XML (SQL2003 тип: XML)

Хранит XML данные в столбце таблицы или в переменной. Размер данных не должен превышать 2 Гб.

Ограничения целостности

Ограничения целостности позволяют автоматически контролировать целостность данных и фильтровать данные, загружаемые в базу данных. Ограничения являются правилами, определяющими то, какие данные являются корректными результатами работы *INSERT*, *UPDATE* и *DELETE*. Если в результате работы команды нарушаются ограничения, то такая команда отменяется.

В стандарте ANSI описываются четыре типа ограничений: CHECK, PRIMARY KEY, UNIQUE и FOREIGN KEY. (Некоторые СУБД предлагают более широкий перечень, обратитесь к главе 3 за подробностями).

Область применения

Ограничения целостности могут создаваться на уровне таблицы или на уровне отдельных столбцов.

Ограничения уровня столбца

Объявляются как часть определения столбца и относятся только к этому конкретному столбцу.

Ограничения уровня таблицы

Объявляются независимо от определений столбцов (обычно в конце оператора CREATE TABLE) и могут относиться к одному или нескольким столбцам. Ограничение на уровне таблицы необходимо в том случае, когда оно относится к нескольким столбцам.

Синтаксис

Ограничения объявляются при создании или изменении таблицы. Общий синтаксис для ограничений следующий:

```
CONSTRAINT [имя_ограничения] тип_ограничения
[(столбец[, ...])]
[предикат] [откладываемость]
[начальная_отложенность]
```

Ниже рассмотрены синтаксические элементы:

CONSTRAINT [имя_ограничения]

Начало объявления ограничения, в котором опционально указывается имя ограничения. Если имя ограничения не указано, то оно будет сгенерировано автоматически. На некоторых платформах ключевое слово *CONSTRAINT* также необязательно.

Сгенерированные системой имена ограничений обычно неинтерпретируемы. Считается хорошим тоном указывать осмысленные и легко читаемые имена ограничений.

тип_ограничения

Устанавливает один из типов ограничения: *CHECK*, *PRIMARY KEY*, *UNIQUE* или *FOREIGN KEY*. Более подробная информация по каждому типу ограничений приводится дальше в этой главе.

[(столбец[, ...])]

Связывает ограничение с одним или несколькими столбцами. Столбцы указываются в круглых скобках через запятую. Список столбцов опускается для ограничений уровня столбца. Список столбцов используется не в каждом типе ограничений. Например, ограничения *CHECK* не используют список столбцов.

предикат

Указывает предикат для ограничения типа *CHECK*.

откладываемость

Описывает ограничение как откладываемое (*DEFERRABLE*) или неотклады-ваемое (*NOT DEFERRABLE*). Если ограничение откладываемое, то можно сделать так, что проверка ограничения будет осуществляться в конце транзакции. Если ограничение неоткладываемое, то оно проверяется в конце каждого SQL оператора.

начальная_отложенность

Если ограничение является откладываемым, то оно может быть изначально отложенным (*INITALLY DEFERRED*) или изначально немедленным (*INITIALLY IMMEDIATE*). Если указана опция *INITIALLY DEFERRED*, то проверка ограничения будет выполняться в конце транзакции, даже если транзакция состоит из нескольких операторов. В этом случае ограничение должны быть откладываемым (*DEFERRABLE*). Если же указана опция *INITIALLY IMMEDIATE*, то проверка ограничения осуществляется после каждого оператора. В этом случае ограничение может быть как откладываемым (*DEFERRABLE*), так и неоткладываемым (*NOT DEFERRABLE*). Опцией по умолчанию является *INITIALLY IMMEDIATE*.

Имейте в виду, что синтаксис в разных СУБД может отличаться. Читайте детальное описание в главе 3 в секциях, относящихся к конкретным платформам.

Первичные ключи

Первичный ключ (*PRIMARY KEY*) – это ограничение, указывающее, что значения в одном или нескольких столбцах уникально идентифицируют каждую запись в таблице. Первичный ключ считается частным случаем ограничения *UNIQUE*. Вот основные правила, касающиеся первичных ключей:

- Таблица может иметь только один первичный ключ
- В состав первичного ключа не могут входить столбцы с типами *BLOB*, *CLOB*, *NCLOB* и *ARRAY*
- Первичный ключ может быть указан на уровне столбца или на уровне таблицы (если состоит из нескольких столбцов)
- Значения столбца первичного ключа должны быть уникальны и не содержать *NULL*

- Если первичный ключ состоит из нескольких столбцов (такой ключ называется *составным*), то комбинация значений всех столбцов ключа должна быть уникальной и не равной *NULL*
- Для указания связей между таблицами (или иногда внутри одной таблицы) создаются внешние ключи, ссылающиеся на первичные ключи.

Следующий код, соответствующий стандарту ANSI, демонстрирует на примере таблицы **distributors** создание первичного ключа на уровне столбца и на уровне таблицы. Первый пример демонстрирует создание первичного ключа на уровне столбца, второй – на уровне таблицы:

```
-- Ограничение на уровне столбца
CREATE TABLE distributors
  (dist_id        CHAR(4)    NOT NULL PRIMARY KEY,
  dist_name       VARCHAR(40),
  dist_address1   VARCHAR(40),
  dist_address2   VARCHAR(40),
  city            VARCHAR(20),
  state           CHAR(2)    ,
  zip             CHAR(5)    ,
  phone           CHAR(12)   ,
  sales_rep       INT        );

-- Ограничение на уровне таблицы
CREATE TABLE distributors
  (dist_id        CHAR(4)    NOT NULL,
  dist_name       VARCHAR(40),
  dist_address1   VARCHAR(40),
  dist_address2   VARCHAR(40),
  city            VARCHAR(20),
  state           CHAR(2)    ,
  zip             CHAR(5)    ,
  phone           CHAR(12)   ,
  sales_rep       INT        ,
  CONSTRAINT pk_dist_id PRIMARY KEY (dist_id));
```

В варианте создания первичного ключа на уровне таблицы мы легко могли бы создать составной ключ путем перечисления нескольких столбцов через запятую.

Внешние ключи

Внешний ключ (*FOREIGN KEY*) указывает, что один или несколько столбцов одной таблицы ссылаются на столбцы первичного или уникального ключа другой таблицы. (Внешний ключ может ссылаться на ту же таблицу, в которой создан он сам, но такие ситуации достаточно редки). Внешние ключи предотвращают ввод данных в таблицу, не имеющих соответствующих значений в связанной таблице. Внешние ключи являются основным способом создания связей между таблицами в базе данных. Вот некоторые правила, касающиеся внешних ключей:

- В одной таблице может существовать одновременно несколько внешних ключей
- Внешний ключ может ссылаться на первичный или на уникальный ключ другой таблицы для указания связи между таблицами

Полный синтаксис SQL2003 для создания внешних ключей более сложен, чем общий синтаксис для ограничений, приведенный ранее, и зависит от того, создается ли ограничение на уровне столбца или на уровне таблицы:

```
-- Внешний ключ на уровне таблицы
[CONSTRAINT [имя_ограничения] ]
FOREIGN KEY (локальный_столбец[, ...] )
REFERENCES адресуемая_таблица [(адресуемый_столбец[, ...]) ]
[MATCH {FULL | PARTIAL | SIMPLE} ]
[ON UPDATE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT} ]
[ON DELETE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT} ]
[откладываемость] [начальная отложенность]

-- Внешний ключ на уровне столбца
[CONSTRAINT [имя_ограничения] ]
REFERENCES адресуемая_таблица [(адресуемый_столбец[, ...]) ]
[MATCH {FULL | PARTIAL | SIMPLE} ]
[ON UPDATE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT} ]
[ON DELETE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT} ]
[откладываемость] [начальная отложенность]
```

Ключевые слова, используемые при создании стандартного ограничения, были описаны ранее в разделе «Синтаксис». Специальные ключевые слова, используемые при создании внешних ключей, описываются далее:

FOREIGN KEY (локальный_столбец[, ...])

Указывает, что один или несколько столбцов создаваемой или изменяемой таблицы составляет внешний ключ. Этот вариант синтаксиса используется *только* при создании ограничений на уровне таблицы и не используется на уровне столбцов. Мы рекомендуем, чтобы порядок следования и типы списка локальных столбцов совпадали с порядком следования и типами адресуемых столбцов.

REFERENCES адресуемая_таблица [(адресуемый_столбец[, ...])]

Указывает таблицу и набор столбцов, на которые ссылается внешний ключ. Адресуемые столбцы должны уже быть перечислены либо в первичном ключе, либо в уникальном ключе, с опцией *NOT DEFERRABLE*. Типы таблиц также должны совпадать: если одна таблица является локальной временной таблицей, то обе таблицы, указываемые при создании внешнего ключа, должны быть локальными временными.

MATCH {FULL | PARTIAL | SIMPLE}

Определяет необходимую степень соответствия между локальными и адресуемыми столбцами внешнего ключа при наличии значений NULL:

FULL

Указывает, что соответствие внешнего ключа не нарушается, если: 1) ни один из ссылающихся столбцов не равен NULL и значения ссылающихся и адресуемых столбцов совпадают или 2) все ссылающиеся столбцы имеют

значения NULL. В общем случае вам следует либо использовать *MATCH FULL*, либо для всех используемых столбцов использовать ограничение *NOT NULL*.

`PARTIAL`

Указывает, что соответствие внешнего ключа не нарушается, если хотя бы один из ссылающихся столбцов равен NULL, а значения остальных совпадают со значениями в соответствующих адресуемых столбцах.

`SIMPLE`

Указывает, что требования внешнего ключа выполняются, если все значения ссылающихся столбцов либо равны NULL, либо совпадают со значениями адресуемых столбцов. Является опцией по умолчанию.

`ON UPDATE`

Указывает, что нужно выполнить определенное действие для поддержания ссылочной целостности, когда операция *UPDATE* затрагивает один или несколько столбцов таблицы, на которую ссылается внешний ключ. *ON UPDATE* может указываться отдельно или вместе с *ON DELETE*. Если опция не указана, то согласно стандарту по умолчанию используется *ON UPDATE NO ACTION*.

`ON DELETE`

Указывает, что нужно выполнить определенное действие для поддержания ссылочной целостности, когда операция *DELETE* затрагивает один или несколько столбцов таблицы, на которую ссылается внешний ключ. *ON DELETE* может указываться отдельно или вместе с *ON UPDATE*. Если опция не указана, то согласно стандарту по умолчанию используется *ON DELETE NO ACTION*.

`NO ACTION | CASCADE | RESTRICT | SET NULL | SET DEFAULT`

Определяет действие, выполняемое для поддержания ссылочной целостности, когда удаляются или изменяются значения в столбцах первичного или уникального ключа таблицы, на которую ссылается внешний ключ:

`NO ACTION`

Указывает, что не нужно выполнять никаких действий при удалении или изменении значений первичного или уникального ключа, на который ссылается внешний ключ.

`CASCADE`

Указывает, что при удалении или изменении значений первичного или уникального ключа аналогичное действие нужно выполнить с соответствующими значениями внешнего ключа.

`RESTRICT`

Запрещает изменения или удаление значений, на которые ссылается внешний ключ.

`SET NULL`

Указывает, что при удалении или изменении значений первичного или уникального ключа столбцам внешнего ключа нужно присвоить значения NULL.

SET DEFAULT

Указывает, что при удалении или изменении значений первичного или уникального ключа столбцам внешнего ключа нужно присвоить значения по умолчанию.

Как и в примере для первичных ключей, вы можете использовать этот общий синтаксис для создания внешних ключей как уровня столбца, так и уровня таблицы. И те и другие действуют абсолютно одинаково, только описываются в разных частях команды CREATE TABLE. В следующем примере мы создаем внешний ключ, состоящий из столбца **sales_rep**, ссылающийся на столбец **empid** таблицы **employee**. Мы создаем ключ двумя разными способами, сначала на уровне столбца, затем на уровне таблицы:

```
-- создание ограничения на уровне столбца
CREATE TABLE distributors
  (dist_id      CHAR(4)    PRIMARY KEY,
  dist_name     VARCHAR(40),
  dist_address1 VARCHAR(40),
  dist_address2 VARCHAR(40),
  city          VARCHAR(20),
  state         CHAR(2)    ,
  zip           CHAR(5)    ,
  phone         CHAR(12)   ,
  sales_rep     INT        NOT NULL
  REFERENCES employee(empid));

-- создание ограничения на уровне таблицы
CREATE TABLE distributors
  (dist_id      CHAR(4)    NOT NULL,
  dist_name     VARCHAR(40),
  dist_address1 VARCHAR(40),
  dist_address2 VARCHAR(40),
  city          VARCHAR(20),
  state         CHAR(2)    ,
  zip           CHAR(5)    ,
  phone         CHAR(12)   ,
  sales_rep     INT        ,
CONSTRAINT pk_dist_id PRIMARY KEY (dist_id),
CONSTRAINT fk_empid
  FOREIGN KEY (sales_rep)
  REFERENCES employee(empid));
```

Уникальные ключи

Уникальный ключ (*UNIQUE*), называемый иногда *возможным ключом* (*candidate key*) – это ограничение, определяющее, что значения столбца или комбинация значений набора столбцов должны быть уникальны. Существуют следующие правила для уникальных ключей:

- В уникальных ключах не могут использоваться столбцы с типами *BLOB*, *CLOB*, *NCLOB* и *ARRAY*.
- Столбец или набор столбцов одного уникального ключа не может быть таким же, как и у другого уникального или первичного ключа.

- Если уникальный ключ допускает значения NULL, то возможно наличие не более чем одного такого значения.
- Стандарт SQL2003 допускает замену списка столбцов в общем синтаксисе ограничения ключевым словом (*VALUE*). Опция *UNIQUE(VALUE)* указывает, что все столбцы таблицы являются частью уникального ключа. Также ключевое слово *VALUE* запрещает создание любых других уникальных ограничений или первичных ключей в таблице.

В следующем примере мы ограничиваем список наших дистрибьюторов одной компанией на каждый почтовый индекс. А также разрешаем одного (и только одного) дистрибьютора, покрывающего все регионы, с неуказанным почтовым индексом. Эта функциональность достигается созданием уникального ключа на уровне столбца или таблицы:

```
-- создание ограничения на уровне столбца
CREATE TABLE distributors
  (dist_id      CHAR(4)    PRIMARY KEY,
  dist_name     VARCHAR(40),
  dist_address1 VARCHAR(40),
  dist_address2 VARCHAR(40),
  city          VARCHAR(20),
  state         CHAR(2)    ,
  zip           CHAR(5)    UNIQUE,
  phone         CHAR(12)   ,
  sales_rep     INT        NOT NULL
  REFERENCES employee(empid));

-- создание ограничения на уровне таблицы
CREATE TABLE distributors
  (dist_id      CHAR(4)    NOT NULL,
  dist_name     VARCHAR(40),
  dist_address1 VARCHAR(40),
  dist_address2 VARCHAR(40),
  city          VARCHAR(20),
  state         CHAR(2)    ,
  zip           CHAR(5)    ,
  phone         CHAR(12)   ,
  sales_rep     INT        ,
CONSTRAINT pk_dist_id PRIMARY KEY (dist_id),
CONSTRAINT fk_emp_id FOREIGN KEY (sales_rep)
  REFERENCES employee(empid),
CONSTRAINT unq_zip UNIQUE (zip) );
```

Проверочные ограничения целостности

Проверочные ограничения целостности (ограничения целостности *CHECK*) позволяют проверить, удовлетворяют ли данные определенным требованиям. Синтаксис проверочных ограничений очень похож на общий синтаксис ограничений:

```
[CONSTRAINT] [имя_ограничения] CHECK (условие_проверки)
[откладываемость] [начальная отложенность]
```

Большинство элементов проверочных ограничений были уже описаны ранее в этом разделе. Уникальным является только следующий элемент:

условие_проверки

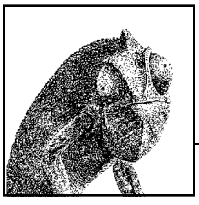
Определяет одно или несколько условий, ограничивающих вставляемые в таблицу данные, используя одно или несколько выражений и предикатов. На столбец может быть наложено одно или несколько условий, соединенных операторами *AND* и *OR* (как в операторе *WHERE*).

Проверочное ограничение считается выполненным, если указанное условие проверки принимает значение *TRUE* или *UNKNOWN* (неизвестно). В проверочных ограничениях должны использоваться булевы операторы (например =, >=, <= и <>) и любые предикаты из SQL2003, такие как *IN* или *LIKE*. Вот некоторые правила для проверочных ограничений:

- Столбец или таблица могут иметь одно или несколько проверочных ограничений.
- Проверочное условие не может содержать недетерминированных функций и подзапросов.
- Проверочное ограничение может ссылаться только на однотипные объекты. То есть, если ограничение создается для глобальной временной таблицы, оно не может ссылаться на обычную таблицу.
- Проверочное условие не может использовать следующие функции из стандарта ANSI: CURRENT_USER, SESSION_USER, SYSTEM_USER, USER, CURRENT_PATH, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP.

В следующем примере создаются проверочные ограничения для столбцов **dist_id** и **zip**. (В примере используется синтаксис SQL Server.) Поле **zip** должно попадать в допустимый набор почтовых индексов, а в поле **dist_id** допускается либо 4 буквы, либо 2 буквы и 2 цифры:

```
-- Создание проверочного ограничения на уровне столбца
CREATE TABLE distributors
  (dist_id      CHAR(4)
    CONSTRAINT pk_dist_id PRIMARY KEY
    CONSTRAINT ck_dist_id CHECK
      (dist_id LIKE '[A-Z][A-Z][A-Z][A-Z]' OR
      dist_id LIKE '[A-Z][A-Z][0-9][0-9]'),
  dist_name     VARCHAR(40),
  dist_address1 VARCHAR(40),
  dist_address2 VARCHAR(40),
  city          VARCHAR(20),
  state         CHAR(2)
    CONSTRAINT def_st DEFAULT ("CA"),
  zip CHAR(5)
    CONSTRAINT unq_dist_zip UNIQUE
    CONSTRAINT ck_dist_zip CHECK
      (zip LIKE '[0-9][0-9][0-9][0-9][0-9]'),
  phone         CHAR(12),
  sales_rep     INT NOT NULL DEFAULT USER
  REFERENCES employee(emp_id))
```



3

Справочник операторов SQL

Эта глава является основной в книге. Она содержит алфавитный перечень всех команд SQL с детальными объяснениями и примерами. Для каждой команды в табл. 3.1 указывается степень поддержки этой команды каждой из четырех рассматриваемых платформ: MySQL 5.1, Oracle Database 11g, PostgreSQL 8.2.1, Microsoft SQL Server 2008. Степень поддержки оценивается одним из следующих значений: «поддерживается», «поддерживается с ограничениями», «поддерживается с вариациями», «не поддерживается». После короткого описания команды по стандарту SQL2003 приводится подробное описание реализации команды в каждой платформе. Если команда платформой не поддерживается, то этот факт отмечается в таблице в начале описания команды и эта команда для соответствующей платформы не рассматривается. Хотя эта книга и не претендует на всестороннее представление стандарта SQL2003, тем не менее, все рассматриваемые команды сравниваются со стандартом.

Как читать эту главу?

При поиске информации в этой главе следует придерживаться следующей последовательности:

1. Прочитайте раздел «Поддержка SQL платформами».
2. Изучите таблицу, в которой приводится информация по степени поддержки команд платформами.
3. Прочитайте раздел, в котором описывается синтаксис команды согласно стандарту SQL2003 (даже если вас интересует реализация в конкретной платформе).
4. Наконец, прочитайте информацию о поддержке команды в интересующей вас платформе.

Разделы, посвященные поддержке команд платформами, могут не содержать детального описания всех аспектов и нюансов команды, так как некоторые детали уже могли быть описаны в разделе, посвященном SQL2003. Поэтому имейте в виду, что если ключевое слово приводится в описании синтаксиса, но описание самого ключевого слова отсутствует, то это происходит потому, что мы просто не повторяем то, что уже написано при описании команды согласно ANSI-стандарту.

Поддержка SQL платформами

В таблице 3.1 приведен список SQL-операторов, а также указана степень поддержки операторов платформами. Таблица устроена следующим образом:

1. В первом столбце перечислены в алфавитном порядке все SQL-операторы.
2. Во втором столбце приводится класс оператора согласно стандарту SQL.
3. В последующих столбцах указывается уровень поддержки оператора каждой платформой:

   Поддерживается (П)

   Платформа поддерживает оператор в соответствии со стандартом SQL2003

   Поддерживается с вариациями (ПСВ)

   Платформа поддерживает оператор в соответствии со стандартом SQL2003, но используется специальный синтаксис или код

   Поддерживается с ограничениями (ПСО)

   Платформа поддерживает некоторые, но не все возможности оператора, описанные в стандарте SQL2003

   Не поддерживается (НП)

   Платформа не поддерживает оператор в соответствии со стандартом SQL2003

В последующих разделах операторы описываются детально. Связанные команды *ALTER* и *CREATE* (например, *ALTER DATABASE* и *CREATE DATABASE*) описываются вместе (например, в разделе «Оператор *CREATE/ALTER DATABASE*»).

Помните, что даже если команда отмечена как не поддерживаемая в конкретной платформе, то как правило существует альтернативный способ или синтаксис для той же команды или функции. Поэтому внимательно читайте описание и примеры. Также в табл. 3.1 приведено несколько команд не из стандарта SQL2003; для таких команд вместо класса операторов указано «Не ANSI».

В таблице есть несколько команд (например, CREATE DOMAIN или ALTER DOMAIN), которые не поддерживаются ни одной платформой. Так как эта книга фокусируется на реализации SQL, то неподдерживаемые команды указаны в таблице, но их описание не приводится.

Таблица 3.1. Алфавитный краткий справочник команд SQL

Команда	Класс SQL2003	MySQL v5.1	Oracle v11g	PostgreSQL v8.2.1	SQL Server 2008
ALL/ANY/SOME	SQL-data	П	П	П	П
BETWEEN	SQL-data	П	П	П	П
ALTER DATABASE	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
ALTER FUNCTION	SQL-schema	ПСО	ПСВ	ПСО	ПСВ
ALTER INDEX	Не ANSI	ПСВ	ПСВ	ПСВ	ПСВ
ALTER METHOD	SQL-schema	НП	НП	НП	НП

Команда	Класс SQL2003	MySQL v5.1	Oracle v11g	PostgreSQL v8.2.1	SQL Server 2008
ALTER PROCEDURE	SQL-schema	ПСВ	ПСВ	НП	ПСВ
ALTER ROLE	SQL-schema	НП	ПСВ	ПСВ	ПСО
ALTER SCHEMA	SQL-schema	ПСО	НП	ПСО	ПСО
ALTER TABLE	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
ALTER TRIGGER	He ANSI	НП	ПСВ	ПСВ	ПСВ
ALTER TYPE	SQL-schema	НП	ПСВ	НП	НП
ALTER VIEW	He ANSI	ПСВ	ПСВ	ПСВ	ПСВ
CALL	SQL-control	П	ПСВ	НП	НП
CLOSE CURSOR	SQL-data	НП	П	П	ПСВ
COMMIT	SQL-transaction	ПСВ	ПСВ	ПСВ	ПСВ
CONNECT	SQL-connection	НП	П	НП	ПСВ
CREATE DATABASE	He ANSI	ПСВ	ПСВ	ПСВ	ПСВ
CREATE FUNCTION	SQL-schema	ПСО	ПСВ	ПСО	ПСВ
CREATE INDEX	He ANSI	ПСВ	ПСВ	ПСВ	ПСВ
CREATE METHOD	SQL-schema	НП	НП	НП	НП
CREATE PROCEDURE	SQL-schema	ПСВ	ПСВ	НП	ПСВ
CREATE ROLE	SQL-schema	НП	ПСВ	ПСВ	ПСО
CREATE SCHEMA	SQL-schema	ПСО	ПСВ	ПСО	ПСО
CREATE TABLE	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
CREATE TRIGGER	SQL-schema	ПСО	ПСВ	ПСВ	ПСВ
CREATE TYPE	SQL-schema	НП	ПСО	ПСВ	ПСВ
CREATE VIEW	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
DECLARE CURSOR	SQL-data	ПСО	ПСО	ПСО	ПСО
DELETE	SQL-data	ПСВ	ПСВ	ПСВ	ПСВ
DISCONNECT	SQL-connection	НП	ПСО	НП	ПСО
DROP DATABASE	He ANSI	ПСВ	НП	ПСВ	ПСВ
DROP DOMAIN	SQL-schema	НП	НП	П	НП
DROP FUNCTION	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
DROP INDEX	He ANSI	ПСВ	ПСВ	ПСВ	ПСВ
DROP METHOD	SQL-schema	НП	ПСВ	НП	НП
DROP PROCEDURE	SQL-schema	ПСВ	П	НП	П
DROP ROLE	SQL-schema	НП	ПСВ	ПСВ	ПСВ
DROP SCHEMA	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ

Таблица 3.1 (продолжение)

Команда	Класс SQL2003	MySQL v5.1	Oracle v11g	PostgreSQL v8.2.1	SQL Server 2008
DROP TABLE	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
DROP TRIGGER	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
DROP TYPE	SQL-schema	НП	П	П	НП
DROP VIEW	SQL-schema	ПСВ	П	П	П
EXCEPT	SQL-data	НП	ПСО	ПСО	ПСО
EXISTS	SQL-data	П	П	П	П
FETCH	SQL-data	ПСО	ПСО	ПСВ	ПСВ
GRANT	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
IN	SQL-data	П	П	П	П
INSERT	SQL-data	ПСВ	ПСВ	ПСВ	ПСВ
INTERSECT	SQL-data	НП	ПСО	ПСО	ПСО
IS	SQL-data	П	П	П	П
JOIN	SQL-data	ПСВ	П	ПСВ	ПСО
LIKE	SQL-data	П	П	ПСВ	П
MERGE	SQL-data	П	ПСВ	П	ПСВ
OPEN	SQL-data	П	П	НП	П
ORDER BY	SQL-data	ПСО	ПСВ	ПСВ	ПСО
RELEASE SAVEPOINT	SQL-transaction	П	НП	П	НП
RETURN	SQL-control	П	П	П	П
REVOKE	SQL-schema	ПСВ	ПСВ	ПСВ	ПСВ
ROLLBACK	SQL-transaction	ПСО	ПСВ	ПСВ	ПСВ
SAVEPOINT	SQL-transaction	П	П	П	ПСО
SELECT	SQL-data	ПСВ	ПСВ	ПСВ	ПСВ
SET	SQL-session	П	НП	П	П
SET CONNECTION	SQL-connection	НП	НП	НП	ПСО
SET CONSTRAINT	SQL-connection	НП	ПСВ	ПСВ	НП
SET PATH	SQL-session	НП	НП	НП	НП
SET ROLE	SQL-session	НП	ПСВ	НП	НП
SET SCHEMA	SQL-session	НП	НП	НП	НП
SET SESSION AUTHORIZATION	SQL-session	НП	НП	П	НП
SET TIME ZONE	SQL-session	НП	ПСВ	ПСО	НП
SET TRANSACTION	SQL-session	ПСВ	ПСО	П	ПСВ

Команда	Класс SQL2003	MySQL v5.1	Oracle v11g	PostgreSQL v8.2.1	SQL Server 2008
START TRANSACTION	SQL-transaction	ПСО	НП	НП	НП
SUBQUERY	SQL-data	ПСО	П	П	П
TRUNCATE TABLE	В стиле SQL-data, но отсутствует в SQL3	П	П	П	П
UNION	SQL-data	НП	ПСО	ПСО	ПСО
UPDATE	SQL-data	ПСВ	ПСВ	ПСВ	ПСВ
WHERE	SQL-data	П	П	П	П

Перечень операторов SQL

ALL/ANY/SOME

Оператор ALL проверяет, что булево выражение истинно для всех строк, возвращаемых подзапросом. Оператор ANY и его синоним SOME проверяют, что булево выражение истинно хотя бы для одной строки подзапроса.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
SELECT ...
WHERE выражение сравнение {ALL | ANY | SOME} ( подзапрос )
```

Ключевые слова

WHERE выражение

Скалярное *выражение* (например, столбец) сравнивается с каждой строкой *подзапроса* (при использовании *ALL*) или с каждой строкой до первого обнаруженного совпадения (при использовании *ANY* или *SOME*). При использовании *ALL* оператор возвращает результат *TRUE* только в том случае, если условие сравнения выполняется для всех строк подзапроса. При использовании *ANY* или *SOME* оператор возвращает *TRUE*, если условие выполняется для одной или более записей подзапроса.

сравнение

Оператор сравнения выражения и подзапроса. Должен использоваться стандартный оператор сравнения, такой как =, <>, !=, >, >=, < или <=.

Общие правила

Оператор *ALL* возвращает булево значение *TRUE* в одном из двух случаев: либо подзапрос возвращает пустой результат (т. е. ни одной строки), либо каждая строка подзапроса удовлетворяет условию сравнения. Если ни одна строка не удовлетворяет условию сравнения, то *ALL* возвращает *FALSE*. Операторы *ANY* и *SOME* возвращают *TRUE*, если хотя бы одна строка подзапроса удовлетворяет условию сравнения. Если же строк, удовлетворяющих условию сравнения, не найдено либо подзапрос возвращает пустой результат, то возвращается *FALSE*. Если хотя бы одно возвращаемое подзапросом значение равно NULL, то оператор также возвращает NULL.[1]

Не используйте в подзапросе специальные фразы вида *ORDER BY*, *GROUP BY*, *CUBE*, *ROLLUP*, *WITH* и т. п.

Например, следующий запрос возвращает авторов, не имеющих на текущий момент книг:

```
SELECT au_id
FROM authors
WHERE au_id <> ALL(SELECT au_id FROM titleauthor)
```

Вы можете использовать *ANY* или *SOME* для совершенно разных проверок. Например, следующий запрос возвращает записи из таблицы **employees**, в которых значение поля **job_lvl** совпадает со значением этого же поля в таблице **employees_archive** для какого-либо сотрудника, работающего в городе Анкоридж:

```
SELECT *
FROM employees
WHERE job_lvl = ANY(SELECT job_lvl FROM employees_archive
                    WHERE city = 'Anchorage')
```

Советы и хитрости

К использованию операторов *ALL* и *ANY/SOME* достаточно сложно привыкнуть. Большинство разработчиков считает более простыми и удобными операторы *IN* и *EXISTS*.

Оператор *EXISTS* семантически эквивалентен *ANY/SOME*.

MySQL

MySQL поддерживает *ALL* и *ANY/SOME* в соответствии с приведенным описанием, за исключением версий младше 4.0. В ранних версиях можно использовать оператор *IN* вместо *EXISTS* для *ANY/SOME*.

---

[1] Корректно только для случая ALL; для случая ANY, если есть совпадение, но встретился NULL, все равно будет возвращено TRUE. – *Прим. науч. ред.*

Oracle

Oracle поддерживает *ALL* и *ANY/SOME* в соответствии со стандартом ANSI, но с небольшим дополнением: вместо подзапроса можно указывать список значений. Например, для поиска всех сотрудников со значением **job_lvl**, равным 9 или 14, можно использовать следующий запрос:

```
SELECT * FROM employee
WHERE job_lvl = ALL(9, 14);
```

PostgreSQL

PostgreSQL поддерживает *ALL* и *ANY/SOME* в соответствии с приведенным описанием.

SQL Server

SQL Server поддерживает *ALL* и *ANY/SOME* в соответствии со стандартом ANSI. Также поддерживаются дополнительные операторы сравнения: «не больше чем» (*!>*) и «не меньше чем» (*!<*).

См. также

BETWEEN
EXISTS
SELECT
WHERE

BETWEEN

Оператор *BETWEEN* проверяет принадлежность значения диапазону. Оператор возвращает *TRUE*, если значение попадает в диапазон, и *FALSE*, если значение выходит за границы диапазона. Результат равен NULL, если любая из границ диапазон равна NULL.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
SELECT ...
WHERE выражение
      [NOT] BETWEEN нижняя_граница AND верхняя_граница
```

Ключевые слова

WHERE выражение

Скалярное выражение, например, столбец, сравнивается с диапазоном, задаваемым *нижней_границей* и *верхней_границей*.

[NOT] BETWEEN *нижняя_граница* *AND* *верхняя_граница*

Сравнивает выражение с *нижней_границей* и *верхней_границей*. Границы считаются включенными в диапазон, то есть условие проверки можно прочитать как «[не] больше или равно *нижней_границе* и меньше или равно *верхней_границе*».

Общие правила

Оператор *BETWEEN* используется для проверки принадлежности значения диапазону. Оператор можно использовать с любыми типами данных, кроме *BLOB*, *CLOB*, *NCLOB*, *REF* и *ARRAY*.

Например, следующий запрос возвращает идентификаторы книг, для которых объем продаж с начала года составляет от 10000 до 20000:

```
SELECT title_id
FROM titles
WHERE ytd_sales BETWEEN 10000 AND 20000
```

При сравнении с помощью *BETWEEN* границы считаются включенными в диапазон, поэтому значения 10000 и 20000 также попадают в условие поиска. Если вы хотите исключить границы диапазона, то вам следует использовать операторы «больше» (>) и «меньше» (<):

```
SELECT title_id
FROM titles
WHERE ytd_sales > 10000
  AND ytd_sales < 20000
```

Оператор *NOT BETWEEN* позволяет искать значения, не попадающие в диапазон. Например, вы можете найти все книги, которые были опубликованы не в 2003 году:

```
SELECT title_id
FROM titles
WHERE pub_date NOT BETWEEN '01-JAN-2003'
  AND '31-DEC-2003'
```

Советы и хитрости

Аккуратные программисты полагают, что необходимо очень внимательно относиться к использованию ключевого слова *AND* во фразе *WHERE*. Для того чтобы *AND* оператора *BETWEEN* не был случайно спутан с логическим оператором *AND*, можно заключить *BETWEEN* в скобки:

```
SELECT title_id
FROM titles
WHERE (ytd_sales BETWEEN 10000 AND 20000)
  AND pubdate >= '1991-06-12 00:00:00.000'
```

Отличия в платформах

Во всех платформах оператор *BETWEEN* поддерживается в соответствии с приведенным описанием.

См. также

ALL/ANY/SOME

EXISTS
SELECT
WHERE

CALL

Оператор *CALL* вызывает хранимую процедуру.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается с вариациями
PostgreSQL	Не поддерживается
SQL Server	Не поддерживается

Синтаксис SQL2003

```
CALL процедура([параметр[, ...]])
```

Ключевые слова

CALL процедура

Указывает хранимую процедуру, которую вы хотите вызвать. Процедура должна быть предварительно объявлена и доступна в текущем пользовательском контексте (экземпляре, базе данных, схеме и т. д.)

([параметр[, …]])

Задает значения входных параметров процедуры. Параметры должны быть указаны в том же порядке, в котором они указаны в процедуре: например, пятый параметр в списке задает значение для пятого параметра процедуры. Параметры должны быть заключены в скобки и разделены запятыми. Обратите внимание, что скобки необходимы даже при отсутствии параметров, то есть, если процедура не принимает входных параметров, для вызова все равно нужно писать *CALL …( )*. Строки необходимо заключать в одинарные кавычки. Если процедура имеет только выходные параметры, то в скобках необходимо указывать соответствующие переменные (host variables) или маркеры.

Общие правила

Оператор *CALL* позволяет вызвать хранимую процедуру. Просто укажите имя процедуры и в скобках перечислите параметры. В следующем примере для Oracle создается и затем вызывается простая процедура:

```
CREATE PROCEDURE update_employee_salary
  (emp_id NUMBER, updated_salary NUMBER)
IS
BEGIN
  UPDATE employee SET salary = updated_salary
WHERE employee_id =emp_id;
END;
/
CALL update_employee_salary(1517, 95000);
```

Советы и хитрости

Результат работы хранимой процедуры обычно можно получить с помощью оператора *GET DIAGNOSTIC*. *GET DIAGNOSTIC* не очень широко поддерживается различными платформами, поэтому обращайтесь за информацией к документации.

В некоторых платформах поддерживается оператор *EXECUTE*, имеющий ту же функциональность. В некоторых случаях вы можете предпочесть *EXECUTE*, потому что с помощью этого оператора можно вызывать не только хранимые процедуры, но любые варианты SQL-кода, включая функции, методы и пакеты операторов.

MySQL

MySQL поддерживает оператор CALL в соответствии со стандартом ANSI.

Oracle

В Oracle оператор CALL используется для вызова как отдельных хранимых процедур и функций, так и процедур и функций, содержащихся в пакетах или типах. В Oracle используется следующий синтаксис:

```
CALL [схема.][{тип | пакет}.]процедура[@dblink]
[(параметр[, ...])]
[INTO :переменная [[INDICATOR] :индикатор]]
```

где:

CALL [схема.][{тип | пакет}.]процедура[@dblink]

Вызывает именованный объект. Вы можете либо полностью указать имя объекта, включая схему, тип и т. д., либо использовать текущую схему и экземпляр базы данных. Если процедура или функция находится в другой базе данных, то используйте предварительно созданный указатель на эту базу данных (в синтаксисе обозначен как *dblink*).

INTO :переменная

Указывает имя предварительно объявленной переменной, в которой будет сохранено возвращаемое значение при вызове функции. *INTO* необходим только при вызове функций.

INDICATOR :индикатор

Указывает состояние переменной (например, равно возвращаемое значение NULL или нет) для функций, созданных на Pro*C/C++.

В качестве параметров оператора *CALL* в Oracle не могут использоваться псевдостолбцы и функции *VALUE* или *REF*. Вам следует использовать локальные переменные для всех параметров, соответствующих аргументам процедуры или функции с типами *IN* или *IN OUT*.

PostgreSQL

Не поддерживается.

SQL Server

Не поддерживается. Вместо этого используйте не входящий в стандарт ANSI оператор *EXECUTE*.

См. также

CREATE/ALTER PROCEDURE

CLOSE CURSORE

Оператор *CLOSE CURSOR* закрывает серверный курсор, созданный с помощью оператора *DECLARE CURSOR*.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
CLOSE курсор
```

Ключевые слова

курсор

Указывает имя курсора, созданного ранее с помощью оператора *DECLARE CURSOR*.

Общие правила

Оператор *CLOSE* закрывает курсор и удаляет соответствующее курсору результирующее множество (result set). Все СУБД при этом также освобождают все блокировки, удерживаемые курсором, хотя в стандарте об этом ничего не сказано. Пример:

```
CLOSE author_names_cursor;
```

Советы и хитрости

Вы можете также закрыть курсор неявно с помощью оператора *COMMIT*, а курсоры, созданные с опцией *WITH HOLD*, – оператором *ROLLBACK*.

MySQL

MySQL поддерживает оператор в соответствии со стандартом ANSI.

Oracle

Oracle поддерживает оператор в соответствии со стандартом ANSI.

PostgreSQL

PostgreSQL поддерживает оператор в соответствии со стандартом ANSI. PostgreSQL неявно выполняет оператор *CLOSE* для каждого открытого курсора при завершении транзакции с помощью *COMMIT* или *ROLLBACK*.

SQL Server

SQL Server поддерживает синтаксис, определенный стандартом ANSI, а также дополнительное ключевое слово *GLOBAL*:

```
CLOSE [GLOBAL] курсор
```

где:

GLOBAL

Указывает, что ранее созданный курсор является глобальным.

В стандарте ANSI написано, что при закрытии курсора удаляется соответствующее ему результирующее множество. Блокировки являются физическими особенностями каждой СУБД, следовательно, не могут являться частью стандарта ANSI SQL. Тем не менее, все рассматриваемые СУБД при закрытии курсора освобождают все удерживаемые блокировки. Особенностью физической реализации SQL Server является то, что при закрытии курсора освобождаемое пространство в памяти не возвращается в пул свободной памяти. Для перемещения освобождаемой памяти в пул необходимо выполнить команду *DEALLOCATE курсор*.

В следующем примере для SQL Server открывается курсор и извлекаются записи о сотрудниках, фамилии которых начинаются с 'K':

```
DECLARE employee_cursor CURSOR FOR
  SELECT lname, fname
  FROM pubs.dbo.employee
  WHERE lname LIKE 'K%'
OPEN employee_cursor
FETCH NEXT FROM employee_cursor
WHILE @@FETCH_STATUS = 0
BEGIN
  FETCH NEXT FROM employee_cursor
END
CLOSE employee_cursor
DEALLOCATE employee_cursor
GO
```

См. также

DECLARE CURSOR
FETCH
OPEN

COMMIT

Оператор *COMMIT* явно заканчивает текущую транзакцию и фиксирует все сделанные в базе данных изменения. Транзакции неявно начинаются при выполнении операторов *INSERT*, *UPDATE* и *DELETE*, а могут быть начаты явно с помо-

щью команды *START*. В любом из этих случаев *COMMIT* завершит открытую транзакцию.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
COMMIT [WORK] [AND [NO] CHAIN]
```

Ключевые слова

COMMIT [WORK]

Завершает текущую открытую транзакцию и фиксирует в базе данных все сделанные в рамках транзакции изменения. Опциональное ключевое слово *WORK* не имеет никакого эффекта.

AND [NO] CHAIN

AND CHAIN указывает СУБД, что следующую транзакцию нужно обрабатывать так, как если бы она была частью текущей транзакции. На самом деле, транзакции являются отдельными элементами работы, но при этом имеют общее окружение (например, уровень изоляции транзакции). Опциональное ключевое слово *NO* явно указывает, что нужно придерживаться поведения по умолчанию, описанного в ANSI. Само по себе слово *COMMIT* функционально эквивалентно оператору *COMMIT WORK AND NO CHAIN*.

Общие правила

В простых случаях вы будете выполнять транзакции (то есть код SQL, изменяющий данные и объекты в базе данных) без их явного объявления. Тем не менее, лучшим вариантом является явное завершение транзакций при помощи оператора *COMMIT*. Так как на протяжении всей транзакции могут быть заблокированы отдельные записи или даже целые таблицы, то чрезвычайно важно заканчивать транзакции как можно раньше. Поэтому явное выполнение *COMMIT* поможет вам избежать проблем, связанных с блокировками и параллельным выполнением.

Советы и хитрости

Самое важное, что нужно иметь в виду, это то, что некоторые СУБД выполняют автоматически *неявные* транзакции, в то время как другие требуют использования явных транзакций. Если вы предположите, что СУБД использует какой-то определенный способ, то вы можете ошибиться. Поэтому при переходе от одной платформы к другой следует придерживаться стандартного способа работы с транзакциями. Мы рекомендуем всегда явно начинать транзакцию с помощью *START TRAN* на платформах, поддерживающих этот оператор, и закачивать транзакцию, используя *COMMIT* и *ROLLBACK*.

Кроме фиксации изменений, выполненных одной или несколькими командами, *COMMIT* имеет ряд интересных побочных эффектов, влияющих на другие аспекты транзакций. Во-первых, при *COMMIT* закрываются все связанные открытые курсоры. Во-вторых, очищаются все временные таблицы, при создании которых было указано *ON COMMIT DELETE ROWS* (опция оператора *CREATE TABLE*). В-третьих, проверяются все отложенные ограничения. Если какие-то отложенные ограничения нарушаются, то транзакция откатывается. Наконец, освобождаются все удерживаемые транзакцией блокировки. Обратите внимание, что SQL2003 требует неявного начала транзакции при выполнении одной из следующих команд:

- *ALTER*
- *CLOSE*
- *COMMIT AND CHAIN*
- *CREATE*
- *DELETE*
- *DROP*
- *FETCH*
- *FREE LOCATOR*
- *GRANT*
- *HOLD LOCATOR*
- *INSERT*
- *OPEN*
- *RETURN*
- *REVOKE*
- *ROLLBACK AND CHAIN*
- *SELECT*
- *START TRANSACTION*
- *UPDATE*

Если перед выполнением одной из указанных команд вы не начали транзакцию явно, то стандарт требует от СУБД начать транзакцию автоматически.

MySQL

MySQL поддерживает *COMMIT* при использовании транзакционных движков InnoDB и NDB Cluster, используя следующий синтаксис:

```
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

Ключевое слово *RELEASE* указывает, что после завершения текущей транзакции нужно закрыть клиентское подключение. Ключевое слово *NO*, как для *CHAIN*, так и для *RELEASE*, заменяет действие на противоположное, запрещая сцепление транзакций или автоматическое закрытие соединения.

Oracle

В Oracle не поддерживается фраза AND [NO] CHAIN, но есть и некоторые расширения относительно стандарта:

```
COMMIT [WORK] [ {COMMENT 'текст' | FORCE 'текст'[,целое]} ];
```

где:

COMMENT 'текст'

Связывает с текущей транзакцией комментарий длиной до 255 символов. В случае отката транзакции комментарий сохраняется в представлении словаря данных с названием **DBA_2PC_PENDING**.

FORCE 'текст' [, целое]

Позволяет завершить распределенную транзакцию, параметр *'текст'* содержит ID локальной или глобальной транзакции. Транзакции можно идентифицировать, запрашивая данные из представления **DBA_2PC_PENDING**. В качестве опционального параметра можно задать *целое* число, явно присваивающее транзакции системный номер изменения (SCN). Если параметр не указан, то используется текущее значение SCN.

Выполнение команды *COMMIT* с фразой *FORCE* завершает только транзакцию, указанную в параметрах. На текущую транзакцию никакого действия оказано не будет (если только ее идентификатор явно не указан в *FORCE*). Фразу *FORCE* нельзя использовать в PL/SQL. В следующем примере заканчивается транзакция, и с ней связывается комментарий:

```
COMMIT WORK COMMENT 'Сомнительная транзакция, позвоните (949) 555-1234';
```

PostgreSQL

В PostgreSQL реализован следующий синтаксис:

```
COMMIT [WORK | TRANSACTION];
```

Ключевые слова *WORK* и *TRANSACTION* необязательны. Когда вы выполняете *COMMIT*, все изменения, сделанные в транзакции, записываются на диск и становятся доступными другим пользователям. Например:

```
INSERT INTO sales VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832');
COMMIT WORK;
```

SQL Server

SQL Server не поддерживает фразу *AND [NO] CHAIN*. Поддерживается ключевое слово *TRANSACTION* как синоним *WORK*, общий синтаксис следующий:

```
COMMIT { [TRAN[SACTION] [имя_транзакции] ] | [WORK] }
```

Microsoft SQL Server позволяет создавать именованные транзакции с помощью оператора *START TRAN*. Синтаксис *COMMIT* позволяет явно задать имя закрываемой транзакции или сохранить его в переменную. Любопытно, что SQL Server закрывает только самую последнюю открытую транзакцию, не принимая во внимание указываемое имя транзакции.

При использовании *COMMIT WORK* завершаются все открытые транзакции и фиксируются все сделанные изменения. Имя транзакции в *COMMIT WORK* указывать нельзя.

См. также

ROLLBACK

START TRANSACTION

CONNECT

Оператор CONNECT устанавливает соединение с СУБД и определенной базой данных внутри СУБД.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается
PostgreSQL	Не поддерживается
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
CONNECT TO {DEFAULT | {[имя_сервера] [AS имя_подключения]
  [USER пользователь]}
```

Ключевые слова

DEFAULT

Инициирует сессию с сервером и базой данных, подключение к которым устанавливается по умолчанию. Стандарт указывает, что, если при создании сессии явно не выполнен *CONNECT*, то автоматически выполняется *CONNECT TO DEFAULT*.

имя_сервера

Устанавливает подключение к указанному серверу. Для указания имени сервера можно использовать строковой литерал в одинарных кавычках или локальную переменную.

AS имя_подключения

Задает имя подключения. Имя может быть задано строкой в одинарных кавычках либо передано через переменную. Опция необязательна только при первом подключении к серверу, во всех последующих подключениях использование *AS* обязательно. Имя подключения позволяет различать между собой сессии, открытые на одном сервере несколькими (или даже одним) пользователями.

USER пользователь

Указывает пользователя, от имени которого создается подключение к серверу.

Общие правила

Используйте оператор *CONNECT* для создания интерактивного соединения с СУБД. Период между выполнением команд *CONNECT* и *DISCONNECT* обычно называется сессией. Как правило, вы выполняете всю работу с СУБД в рамках явно создаваемых сессий.

Если вы не укажете имя сервера, подключения или пользователя, то СУБД будет использовать значения по умолчанию. Значения по умолчанию меняются от платформы к платформе.

Для подключения к серверу *houston* под определенным именем пользователя можно выполнить следующую команду:

```
CONNECT TO houston USER pubs_admin
```

Если СУБД требует использования именованных сессий, то можно использовать следующий синтаксис:

```
CONNECT TO houston USER pubs_admin
  AS pubs_administrative_session
```

А если вам требуется простейшее кратковременное подключение, то используйте следующее:

```
CONNECT TO DEFAULT
```

Советы и хитрости

Если вы выполняете оператор *CONNECT* до того, как явно закрыли предыдущую сессию, то старая сессия становится неактивной, а новая – активной. Затем вы можете переключаться между сессиями, используя оператор *SET CONNECTION*.

Программа SQL*Plus в Oracle использует команду *CONNECT* для другого: для подключения пользователя к определенной схеме.

MySQL

Не поддерживается.

Oracle

CONNECT позволяет указать конкретное имя пользователя при подключении. Кроме того, можно создать подключение со специальными привилегиями – *AS SYSOPER* или *AS SYSDBA*. В Oracle используется следующий синтаксис:

```
CONN[ECT] [[имя_пользователя/пароль] [AS {SYSOPER|SYSDBA}]]
```

где:

CONN[ECT] [*имя_пользователя/пароль*]

Устанавливает подключение к экземпляру базы данных.

AS {SYSOPER|SYSDBA}

Устанавливает подключение с одной из двух опциональных системных ролей.

Если уже есть открытое подключение, то *CONNECT* фиксирует все текущие транзакции, закрывает текущую сессию и открывает новую.

Оператор *CONNECT* допустим в программах SQL*Plus и iSQL*Plus.

PostgreSQL

PostgreSQL не поддерживает оператор *CONNECT явно*. Тем не менее, в программном интерфейсе сервера (Server Programming Interface, SPI) есть оператор *SPI_CON*NECT, а в программном пакете PG/TCL есть оператор *PG_CONNECT*.

SQL Server

SQL Server поддерживает базовые возможности оператора *CONNECT* при использовании из Embedded SQL (из программ, написанных на C++ и Visual Basic) в соответствии со следующим синтаксисом:

```
CONNECT TO [имя_сервера.]база_данных [AS имя_подключения]
USER {имя_пользователя[.пароль] | $integrated}
```

где:

CONNECT TO [*имя_сервера*.]*база_данных*

Указывает имя сервера и базы данных, к которым вы хотите подключиться. Имя сервера можно не указывать – по умолчанию будет выполнено подключение к локальному серверу.

AS имя_подключения

Задает имя подключения. Имя подключения является строкой длиной до 30 символов. Символы могут быть любыми, кроме дефиса, но первый символ должен обязательно быть буквой. Использование зарезервированных слов *CURRENT* и *ALL* запрещено. Имя подключения является обязательным только в том случае, если вы создаете больше чем одно подключение.

USER {*имя_пользователя*[.*пароль*] | *$integrated*}

Указывает имя пользователя и пароль, используемые при подключении. Пароль можно задать явно или использовать встроенные средства безопасности Windows. При подключении пароль можно не указывать.

К примеру, мы можем подключиться к серверу *new_york*, используя пользователя Windows с именем *pubs_admin*:

```
CONNECT TO new_york.pubs USER pubs_admin
```

Та же команда может быть выполнена с использованием средств безопасности SQL Server:

```
EXEC SQL CONNECT TO new_york.pubs USER pubs_admin
```

Та же команда, выполненная с использованием средств безопасности Windows:

```
EXEC SQL CONNECT TO new_york.pubs USER $integrated
```

Для переключения между сессиями используйте оператор *SET CONNECTION*.

См. также

SET CONNECTION

CREATE/ALTER DATABASE

Стандарт ANSI не содержит описания оператора CREATE DATABASE. Но так как работа с базами данных без этого оператора практически невозможна, мы решили включить сюда описание этого оператора. Практически все платформы поддерживают оператор в том или ином виде.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Общие правила

С помощью этой команды создается новая пустая база данных. В большинстве платформ для создания базы данных необходимы права администратора. После того как база данных создана, вы можете наполнить ее объектами (таблицами, представлениями, триггерами и т. д.) и наполнить таблицы данными.

На некоторых платформах оператор *CREATE DATABASE* создает соответствующие файлы на диске для хранения данных и метаданных базы данных.

Советы и хитрости

Так как CREATE DATABASE не является стандартным оператором, то его синтаксис в зависимости от платформы может меняться кардинально.

MySQL

В MySQL оператор *CREATE DATABASE* создает новый каталог для хранения объектов базы данных:

```
CREATE { DATABASE | SCHEMA } [IF NOT EXISTS] имя_базы
  [ [DEFAULT] CHARACTER SET кодовая_страница ]
  [ [DEFAULT] COLLATE схема_упорядочения ]
```

Для оператора ALTER DATABASE используется следующий синтаксис:

```
ALTER { DATABASE | SCHEMA } имя_базы
{ [ [DEFAULT] CHARACTER SET кодовая_страница]
  [ [DEFAULT] COLLATE схема_упорядочения] |
  UPGRADE DATA DIRECTORY NAME }
```

где:

{CREATE|ALTER}{DATABASE|SCHEMA} *имя_базы*

Создает базу данных и каталог с соответствующими именами. Каталог создается внутри каталога с данными MySQL. Таблицам соответствуют файлы внутри каталога базы данных. Слово *SCHEMA* является синонимом слова *DATABASE*.

IF NOT EXISTS

Указывает, что оператор не должен вызывать ошибки в случае, если база данных с таким именем уже существует.

[DEFAULT] CHARACTER SET *кодовая_страница*

Опциональный параметр, задающий кодовую страницу, используемую в базе данных по умолчанию. За полным перечнем допустимых кодовых страниц обращайтесь к документации MySQL.

[DEFAULT] COLLATE схема_упорядочения

Опциональный параметр, задающий схему упорядочения, используемую в базе данных по умолчанию. За полным перечнем допустимых схем упорядочения обращайтесь к документации MySQL.

UPGRADE DATA DIRECTORY NAME

Обновляет имя каталога базы данных для соответствия алгоритму отображения имени базы данных на имя каталога, используемому в MySQL 5.1 и в более поздних версиях.

К примеру, база данных **sales_archive** может уже быть создана в MySQL. Если база данных уже создана, то мы хотим, чтобы команда была завершена без сообщения об ошибке:

```
CREATE DATABASE IF NOT EXISTS sales_archive
```

Все таблицы, которые мы создадим в базе **sales_archive**, будут храниться в отдельных файлах в каталоге *sales_archive*.

Oracle

Oracle предоставляет исключительные возможности контроля над файловой структурой базы данных, намного большие чем просто указание имени базы и каталога для хранения файлов. *CREATE* и *ALTER DATABASE* являются очень мощными командами, некоторые замысловатые опции которых используются только опытными администраторами. Также эти команды являются очень сложными: например, только команде *ALTER DATABASE* посвящено около 50 страниц документации.

Новичкам следует знать, что оператор *CREATE DATABASE* удаляет все данные в существующих файлах данных. Аналогично, удаляются все данные базы данных.

Для создания новой базы данных в Oracle используется следующий синтаксис:

```
CREATE DATABASE [имя_базы]
{[USER SYS IDENTIFIED BY пароль
  | USER SYSTEM IDENTIFIED BY пароль]}
[CONTROLFILE REUSE]
{[LOGFILE определение[, ...]] [MAXLOGFILES целое_число]
  [[MAXLOGMEMBERS] целое_число]
  [[MAXLOGHISTORY] целое_число]
  [{ARCHIVELOG | NOARCHIVELOG}] [FORCE LOGGING]}
[MAXDATAFILES целое_число]
[MAXINSTANCES целое_число]
[CHARACTER SET кодовая_страница]
[NATIONAL CHARACTER SET кодовая_страница]
[EXTENT MANAGEMENT {DICTIONARY | LOCAL
  [ {AUTOALLOCATE | UNIFORM [SIZE int [K | M]]} ]}]
[DATAFILE определение[, ...]]
[SYSAUX DATAFILE определение [, ...]]
[DEFAULT TABLESPACE имя_табличного_пространства
  [DATAFILE определение_файла]
  EXTENT MANAGEMENT {DICTIONARY |
    LOCAL {AUTOALLOCATE|UNIFORM [SIZE целое_число [K|M]]}}]
```

```
[ [{BIGFILE | SMALLFILE}] DEFAULT TEMPORARY TABLESPACE
  имя_табличного_пространства
  [TEMPFILE определение_файла]
  EXTENT MANAGEMENT {DICTIONARY |
    LOCAL {AUTOALLOCATE|UNIFORM [SIZE целое_число [K|M]]}} ]
[ [{BIGFILE | SMALLFILE}] UNDO TABLESPACE
  имя_табличного_пространства
  [DATAFILE определение_ файла] ]
[SET TIME_ZONE = '{ {+ | -} чч:мм | часовой_пояс}']
[SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE]
```

Для изменения существующей базы данных применяется следующий синтаксис:

```
ALTER DATABASE [имя_базы]
[ARCHIVELOG | NOARCHIVELOG] |
  {MOUNT [{STANDBY | CLONE} DATABASE] | OPEN
    [READ ONLY | READ WRITE] [RESETLOGS | NORESETLOGS] |
    [UPGRADE | DOWNGRADE]} |
  {ACTIVATE [PHYSICAL | LOGICAL] STANDBY DATABASE
    [FINISH   APPLY] [SKIP [STANDBY LOGFILE]] |
  SET STANDBY [DATABASE] TO MAXIMIZE
    {PROTECTION |   AVAILABILITY | PERFORMANCE} |
  REGISTER [OR REPLACE] [PHYSICAL | LOGICAL] LOGFILE
    ['файл'] [FOR имя_сессии_logminer] |
  {COMMIT | PREPARE} TO SWITCHOVER TO
    {{[PHYSICAL | LOGICAL] PRIMARY | STANDBY} [WITH[OUT]
    SESSION SHUTDOWN] [WAIT | NOWAIT]} |
    CANCEL} |
  START LOGICAL STANDBY APPLY [IMMEDIATE] [NODELAY]
    [{INITIAL целое_число | NEW PRIMARY имя_dblink |
      {FINISH | SKIP FAILED TRANSACTION}}] |
  {STOP | ABORT} LOGICAL STANDBY APPLY |
  [CONVERT TO {PHYSICAL | SNAPSHOT} STANDBY] |
{RENAME GLOBAL_NAME TO база_данных[.домен[.домен ...]] |
  CHARACTER SET кодовая страница|
  NATIONAL CHARACTER SET кодовая страница |
  DEFAULT TABLESPACE имя_табличного_пространства |
  DEFAULT TEMPORARY TABLESPACE
    {GROUP целое_число | имя_табличного_пространства } |
  {DISABLE BLOCK CHANGE TRACKING | ENABLE BLOCK CHANGE
     TRACKING [USING FILE 'файл'] [REUSE]} |
  FLASHBACK {ON | OFF} |
  SET TIME_ZONE = '{ {+ | -} чч:мм | часовой_пояс }' |
  SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE} |
{ENABLE | DISABLE} { [PUBLIC] THREAD целое_число
  | INSTANCE 'имя_экземпляра' } |
{GUARD {ALL | STANDBY | NONE}} |
{CREATE DATAFILE 'файл'[, ...]
  [AS {NEW | определение_файла[, ...]}] |
  {DATAFILE 'файл' | TEMPFILE 'файл'}[, ...]
  {ONLINE | OFFLINE [FOR DROP |
  RESIZE целое_число [K | M]] | END BACKUP |
  AUTOEXTEND {OFF | ON [NEXT целое_число [K | M]]}
  [MAXSIZE [UNLIMITED | целое_число [K | M]]] |
```

```
  {TEMPFILE 'файл' | TEMPFILE 'файл'}[, ...]
  {ONLINE | OFFLINE | DROP [INCLUDING DATAFILES] |
  RESIZE целое_число [K | M] |
  AUTOEXTEND {OFF | ON [NEXT целое_число [K | M]]}
  [MAXSIZE [UNLIMITED | целое_число [K | M]]] |
  RENAME FILE 'файл' [, ...] TO 'новое_имя' [, ...]} |
{[[NO] FORCE LOGGING] | [[NO]ARCHIVELOG [MANUAL]] |
  [ADD | DROP] SUPPLEMENTAL LOG DATA
    [(ALL | PRIMARY KEY | UNIQUE | FOREIGN KEY |
    FOR PROCEDURAL REPLICATION)[, ...]] COLUMNS |
  [ADD | DROP] [STANDBY] LOGFILE
    {{[THREAD целое_число | INSTANCE 'имя_экземпляра']}
    {[GROUP целое_число | имя_файла[, ...]]}
    [SIZE целое_число [K | M]] | [REUSE] |
    [MEMBER] 'файл' [REUSE][, ...] TO имя_файла[, ...]} |
  ADD LOGFILE MEMBER 'файл' [REUSE][, ...]
    TO {[GROUP целое_число | имя_файла [, ...]]} |
  DROP [STANDBY] LOGFILE {MEMBER 'файл' |
    {[GROUP целое_число |   имя_файла [, ...]]}}
  CLEAR [UNARCHIVED] LOGFILE
    {[GROUP целое_число | имя_файла [, ...]]}[, ...]
[UNRECOVERABLE DATAFILE]} |
{CREATE [LOGICAL | PHYSICAL]
  STANDBY CONTROLFILE AS 'файл'  [REUSE] |
  BACKUP CONTROLFILE TO
  {' файл ' [REUSE] | TRACE [AS ' файл ' [REUSE]]}
  [ {RESETLOGS | NORESETLOGS} ]} |
{RECOVER
  {[AUTOMATIC [FROM 'путь']] |
    {[STANDBY] DATABASE
      {[UNTIL {CANCEL | TIME дата | CHANGE целое_число}] |
      USING BACKUP CONTROLFILE} |
    {{[STANDBY]
      {TABLESPACE имя_табличного_пространства [, ...] |
      DATAFILE 'файл'[, ...]}
      [UNTIL [CONSISTENT WITH] CONTROLFILE]} |
      TABLESPACE имя_табличного_пространства [, ...] |
      DATAFILE 'файл' [, ...]} |
    LOGFILE имя_файла [, ...]} [{TEST
      | ALLOW целое_число CORRUPTION
      | [NO]PARALLEL целое_число }]} |
    CONTINUE [DEFAULT] |
    CANCEL}
  {MANAGED STANDBY DATABASE
    {[USING CURRENT LOGFILE]
    [DISCONNECT [FROM SESSION]]
    [NODELAY]
    [UNTIL CHANGE целое_число]
    [FINISH]
    [CANCEL]} |
  TO LOGICAL STANDBY {имя_базы | KEEP IDENTITY}}
{{BEGIN | END} BACKUP}
```

Рассмотрим назначение синтаксических элементов. Сначала для оператора *CREATE DATABASE*:

{CREATE|ALTER} DATABASE [имя_базы]

Создает или изменяет базу данных с заданным именем. Имя может иметь длину до 8 байт и не может содержать европейских и азиатских символов. Можно не указывать имя базы данных, тогда Oracle сгенерирует имя за вас. Но имейте в виду, что оно не будет интуитивно понятным.

USER SYS IDENTIFIED BY пароль |USER SYSTEM IDENTIFIED BY пароль

Задает пароли для пользователей *SYS* и *SYSTEM*. Вы можете указать пароль для обоих пользователей, либо не указывать ни для кого, но нельзя задать пароль лишь для одного из двух.

CONTROLFILE REUSE

Говорит о необходимости повторного использования управляющих файлов, позволяя указать существующие управляющие файлы в параметре *CONTROL_FILES* в *INIT.ORA*. Oracle перезапишет любую информацию, которая содержится в этих файлах. Эта фраза обычно используется при пересоздании базы данных. Поэтому вы вряд ли захотите использовать ее одновременно с *MAXLOGFILES*, *MAXLOGMEMBER*, *MAXLOGHISTORY*, *MAXDATAFILES* или *MAXINSTANCES*.

LOGFILE определение

Служит для создания одного или нескольких журнальных файлов. Вы можете создать несколько файлов одинакового размера и с одинаковыми характеристиками, используя параметр файл (см. ниже), или вы можете создать несколько файлов с разными размерами и характеристиками. Полный синтаксис для описания журнальных файлов достаточно громоздкий, но предоставляет полный контроль:

```
LOGFILE { ('файл'[, ...]) [SIZE int [K | M]]
[GROUP int] [REUSE] }[, ...]
```

LOGFILE ('файл'[,...])

Описывает один или несколько журнальных файлов; параметр *файл* должен содержать путь и имя файла. Все файлы, созданные в операторе *CREATE STATEMENT*, присваиваются журнальному потоку номер 1. При указании нескольких журнальных файлов каждый файл нужно заключать в одинарные кавычки, а файлы между собой разделять запятыми. Весь список должен быть заключен в скобки.

SIZE целое_число [K|M]

Задает размер журнального файла в байтах. Размер можно задавать и в более крупных единицах, добавляя *K* (в килобайтах) или *M* (в мегабайтах).

GROUP целое_число

Задает целочисленный идентификатор группы журнальных файлов. Значение идентификатора должно быть от 1 до *MAXLOGFILES*. База данных Oracle должна иметь минимум две группы журнальных файлов. Если вы не укажете идентификатор, то Oracle создаст соответствующую группу за вас, с размером журнального файла в 100 Мбайт.

REUSE

Указывает, что необходимо повторно использовать существующие журнальные файлы.

MAXLOGFILES *целое_число*

Устанавливает максимальное число журнальных файлов в создаваемой базе данных. Минимальное, максимальное и значение по умолчанию зависят от операционной системы.

MAXLOGMEMBERS *целое_число*

Устанавливает максимально число членов (копий) группы журнальных файлов. Минимальное значение равно 1, максимальное значение и значение по умолчанию зависят от операционной системы.

MAXLOGHISTORY *целое_число*

Устанавливает максимальное число архивных журнальных файлов, доступных при использовании Real Application Cluster (RAC). Вы можете использовать этот параметр, только если база данных работает в режиме *ARCHIVELOG* в кластере. Минимальное значение равно 0, максимальное значение и значение по умолчанию зависят от операционной системы.

ARCHIVELOG | *NOARCHIVELOG*

Определяет режим работы журнальных файлов. При использовании в операторе *ALTER DATABASE* позволяет менять одно значение на другое. В режиме *ARCHIVELOG* оперативные журнальные файлы сохраняются в архивный каталог, что позволяет восстанавливать базу данных при сбое носителя. В режиме *NOARCHIVELOG* журнальные файлы не архивируются. Режим *NOARCHIVELOG* (по умолчанию) также позволяет восстановить базу данных, но не при сбое носителя.

FORCE LOGGING

Переводит все экземпляры баз данных в режим *FORCE LOGGING*, при котором журналируются все изменения в базе данных, за исключением изменений во временных сегментах и табличных пространствах. Установка этого режима имеет более высокий приоритет, чем любые установки на уровне табличного пространства или объекта.

MAXDATAFILES *целое_число*

Устанавливает максимально число файлов данных в создаваемой базе данных. Имейте в виду, что параметр *DB_FILES* в файле INIT.ORA также ограничивает число файлов данных, доступных экземпляру.

MAXINSTANCES *целое_число*

Устанавливает максимальное число экземпляров, которые могут монтировать и открывать создаваемую базу данных. Минимальное значение равно 1, максимальное значение и значение по умолчанию зависят от операционной системы.

CHARACTER SET *кодовая_страница*

Устанавливает кодировку, в которой хранятся данные. Кодовой страницей не может быть *AL16UTF16*. Значение по умолчанию зависит от операционной системы.

NATIONAL CHARACTER SET кодовая_страница

Устанавливает кодировку, в которой хранятся данные в столбцах с типом *NCHAR*, *NCLOB* и *NVARCHAR2*. Может равняться либо *AL16UTF16* (по умолчанию), либо *UTF8*.

EXTENT MANAGEMENT {DICTIONARY | LOCAL}

Создает локально управляемое табличное пространство *SYSTEM* (в противном случае табличное пространство *SYSTEM* будет управляться с помощью словаря данных). Эта фраза требует также создания временного табличного пространства по умолчанию. Если вы опустите фразу *DATAFILE*, то можно опустить также фразу с временным табличным пространством, так как Oracle создаст оба за вас.

DATAFILE определение

Создает один или несколько файлов данных. (Все эти файлы данных становятся частью табличного пространства SYSTEM.) Вы можете указать несколько имен для создания нескольких файлов с одинаковыми размерами и характеристиками. Либо вы можете повторить полностью фразу DATAFILE, так что каждое вхождение будет создавать один или несколько файлов одинакового размера и характеристик. Полный синтаксис создания файлов данных достаточно велик, но зато дает полный контроль:

```
DATAFILE { ('файл1'[, ...])
[SIZE целое_число [K | M]] [REUSE]
[AUTOEXTEND {OFF | ON [NEXT целое_число [K | M]]}]
[MAXSIZE [UNLIMITED | целое_число [K | M]]] } [,...]
```

DATAFILE ('файл1'[, ...])

Создает один или несколько файлов данных, параметр *файл1* содержит и имя файла, и путь. При указании нескольких файлов каждый файл нужно заключать в одинарные кавычки, а файлы между собой разделять запятыми. Весь список должен быть заключен в скобки.

SIZE целое_число [K|M]

Задает размер файла данных в байтах. Размер можно задавать и в более крупных единицах, добавляя *K* (в килобайтах) или *M* (в мегабайтах).

REUSE

Указывает, что необходимо повторно использовать существующие файлы данных.

AUTOEXTEND {OFF|ON [NEXT целое_число [K|M]]}

Разрешает (ON) автоматическое расширение файла данных или временного файла (но не журнального файла). NEXT указывает размер пространства (в байтах, килобайтах или мегабайтах), на который увеличивается файл при нехватке свободного места.

MAXSIZE [UNLIMITED | целое_число [K|M]]

Устанавливает максимальный допустимый размер файла, до которого разрешено автоматическое расширение. *UNLIMITED* позволяет файлу увеличиваться без ограничений (но, конечно, пока есть свободное место на диске). В противном случае вы можете указать максимальный размер в байтах, килобайтах или мегабайтах.

SYSAUX DATAFILE определение

Создает один или несколько файлов данных для табличного пространства SYSAUX. По умолчанию Oracle создает табличные пространства SYSTEM и SYSAUX и управляет ими автоматически. Вы должны использовать эту фразу, если вы явно создали файлы данных для SYSTEM. Если вы опустите эту фразу при использовании Oracle-managed files (OMF), то Oracle автоматически создаст локально управляемое табличное пространство SYSAUX, содержащее один файл данных размером в 100 Мбайт, с автоматическим управлением сегментами и включенным журналированием.

BIGFILE | SMALLFILE

Указывает тип файлов, используемый в создаваемом табличном пространстве. Опция *BIGFILE* означает, что табличное пространство будет состоять из одного файла данных, размером до 8 экзабайт (8 миллионов терабайт). *SMALLFILE* означает, что создается обычное табличное пространство. Значением по умолчанию является *SMALLFILE*.

DEFAULT TABLESPACE имя_табличного_пространства

Указывает постоянное табличное пространство, используемое по умолчанию для всех пользователей, кроме *SYSTEM*. Если фраза опущена, то табличным пространством по умолчанию для всех пользователей является *SYSTEM*.

DEFAULT TEMPORARY TABLESPACE имя_табличного_пространства [TEMPFILE определение_файла]

Указывает имя и расположение временного табличного пространства, используемого по умолчанию. Все пользователи, для которых временное табличное пространство не указано явно, будут работать в этом табличном пространстве. Если вы не создадите временного табличного пространства по умолчанию, то будет использовано табличное пространство *SYSTEM*. Используя эту фразу в операторе *ALTER DATABASE*, вы можете изменить временное табличное пространство, используемое по умолчанию.

TEMPFILE определение_файла

Указание временного файла необязательно, если вы установили параметр *DB_CREATE_FILE_DEST* в INIT.ORA. В противном случае вам необходимо использовать фразу *TEMPFILE*. Синтаксис создания временного файла идентичен синтаксису создания файла данных, описанному ранее в этом разделе.

EXTENT MANAGEMENT {DICTIONARY | LOCAL {AUTOALLOCATE | UNIFORM [SIZE целое_число [K | M]]}}

Определяет способ управления табличным пространством *SYSTEM*. По умолчанию *SYSTEM* управляется с помощью словаря данных. Если же *SYSTEM* создается как локально управляемое табличное пространство, то оно не может быть преобразовано в табличное пространство, управляемое с помощью словаря данных, и не может быть создано ни одного нового табличного пространства, управляемого с помощью словаря данных.

DICTIONARY

Указывает, что табличное пространство управляется с помощью словаря данных. Является значением по умолчанию. Параметры *AUTOALLOCATE* и *UNIFORM* в этом случае не используются.

LOCAL

Указывает, что табличное пространство управляется локально. Эта фраза необязательна, так как все временные табличные пространства управляются локально по умолчанию. Использование этой фразы требует табличного пространства по умолчанию. Если вы не создадите его вручную, то Oracle создаст за вас табличное пространство *TEMP* размером 10 Мбайт с выключенной опцией *AUTOEXTEND*.

AUTOALLOCATE

Указывает, что в локально управляемом табличном пространстве новые экстенты выделяются автоматически по мере необходимости. Размер экстентов выбирается автоматически.

UNIFORM [SIZE целое_число [K|M]]

Указывает, что все экстенты табличного пространства должны быть одинакового размера. Опция *SIZE* позволяет указать размер экстента в байтах, килобайтах или мегабайтах. Значение по умолчанию – 1Мбайт.

UNDO TABLESPACE имя_табличного_пространства [DATAFILE определение_файла]

Определяет название и расположение табличного пространства для данных отката, если при этом параметр *UNDO_MANAGEMENT* в файле *INIT.ORA* установлен в значение *AUTO*. Если вы не используете эту фразу, то Oracle работает с данными отката в сегментах отката. (Вы можете установить для параметра в *INIT.ORA* значение *UNDO_TABLESPACE*. В этом случае необходимо, чтобы значение параметра совпадало с указанным во фразе *UNDO TABLESPACE* именем табличного пространства.)

[DATAFILE определение_ файла]

Создает файл данных для табличного пространства отката. Полный синтаксис смотрите в приведенном ранее описании фразы *DATAFILE*. Фраза обязательна, если вы не установили в *INIT.ORA* параметр *DB_CREATE_FILE_DEST*.

SET TIME_ZONE = '{ {+|-} чч:мм | часовой_пояс}'

Устанавливает часовой пояс либо указанием разницы с Гринвичским временем (известным также как универсальное глобальное время), либо указанием названия часового пояса. (Список часовых поясов можно получить, запросив столбец **tzname** представления *v$timezone_names*.) Если вы не укажете часовой пояс, то будет использоваться часовой пояс, выбранный в операционной системе.

SET DEFAULT {BIGFILE|SMALLFILE} TABLESPACE

Указывает, какого типа по умолчанию будут создаваться табличные пространства (*BIGFILE* или *SMALLFILE*). При создании базы данных эта фраза влияет на табличные пространства *SYSTEM* и *SYSAUX*.

Теперь рассмотрим оператор *ALTER DATABASE*:

MOUNT [{STANDBY | CLONE}] DATABASE]

Монтирует базу данных для пользовательского доступа. Ключевое слово *STANDBY* используется при монтировании физической резервной базы данных, позволяя ей получать архивные журнальные файлы с основной базы данных. Ключевое слово *CLONE* используется при монтировании клона базы данных. Эта фраза не может использоваться одновременно с *OPEN*.

OPEN [READ WRITE | READ ONLY] [RESETLOGS | NORESETLOGS] [UPGRADE | DOWNGRADE]

Открывает предварительно примонтированную базу данных. *READ WRITE* открывает базу данных в режиме чтения/записи, то есть позволяет пользователям генерировать журнальные данные. *READ ONLY* позволяет читать данные, но запрещает изменение данных. *RESETLOGS* отбрасывает все данные повторного выполнения (redo), не примененные при восстановлении базы данных, и устанавливает порядковый номер журнала в значение 1. *NORESETLOGS* оставляет журналы в их текущем состоянии. Опциональные параметры *UPGRADE* и *DOWNGRADE* говорят о необходимости динамического изменения системных параметров, как того требует повышение (upgrade) или понижение (downgrade) версии базы данных. Значением по умолчанию является *OPEN READ WRITE NORESETLOGS*.

ACTIVATE [PHYSICAL | LOGICAL] STANDBY DATABASE [FINISH APPLY] [SKIP [STANDBY LOGFILE]]

Переводит резервную базу данных в статус основной. Вы можете опционально указать *PHYSICAL* (значение по умолчанию) для физической резервной базы данных или *LOGICAL* для логической. *FINISH APPLY* инициирует применение оставшихся журналов повторного выполнения, приводя логическую резервную базу данных в то же состояние, в котором находится основная. Когда этот процесс завершается, логическая резервная база данных окончательно становится основной. Используйте *SKIP* для немедленного перевода физической резервной базы данных в статус основной, с отбрасыванием всех данных, еще не примененных командой *RECOVER MANAGED STANDBY DATABASE FINISH*. Фраза *STANDBY LOGFILE* не несет дополнительного смысла.

SET STANDBY [DATABASE] TO MAXIMIZE {PROTECTION | AVAILABILITY | PERFORMANCE}

Устанавливает уровень защиты данных в основной базе данных. Старые термины *PROTECTED* и *UNPROTECTED* эквивалентны *MAXIMIZE PROTECTION* и *MAXIMIZE PERFOMANCE* соответственно.

PROTECTION

Устанавливает максимальный уровень защиты данных, что приводит к большим накладным расходам и отрицательно влияет на доступность. В этом режиме транзакция фиксируется только после того, как все данные, необходимые для восстановления, были записаны как минимум в одну физическую резервную базу данных, использующую режим передачи журналов *SYNC*.

AVAILABILITY

Устанавливает более низкий, чем при *PROTECTION*, уровень защиты данных, но максимальный уровень доступности. В этом режиме транзакция фиксируется только после того, как все данные, необходимые для восстановления, были записаны как минимум в одну физическую или логическую резервную базу данных, использующую режим передачи журналов *SYNC*.

PERFOMANCE

Обеспечивает максимальную производительность в ущерб безопасности и доступности данных. В этом режиме транзакция фиксируется до того, как в резервную базу данных попадают все данные, необходимые для восстановления.

REGISTER [OR REPLACE] [PHYSICAL | LOGICAL] LOGFILE ['файл']

Выполняется на резервной базе данных для ручной регистрации журнальных файлов с вышедшей из строя основной базы данных. Журнальный файл может быть по желанию объявлен физическим или логическим. Фраза *OR REPLACE* разрешает изменять детали существующих записей об архивных журналах.

FOR имя_сессии_logminer

Регистрирует журнальный файл в специальной сессии LogMiner при работе с Oracle Streams.

{COMMIT | PREPARE} TO SWITCHOVER TO {[PHYSICAL | LOGICAL] PRIMARY | STANDBY}

Выполняет переключение между основной и резервной базами данных, переводя основную в статус резервной, а резервную в статус основной. (При работе с архитектурой RAC нужно останавливать все экземпляры, за исключением текущего.) Для переключения команду необходимо выполнить дважды. Первый раз нужно выполнить *PREPARE TO SWITHOVER* для того, чтобы с основной базы на резервную были переданы журнальные файлы. Затем нужно выполнить *COMMIT TO SWITCHOVER* для окончательного переключения на резервную базу данных. *PHYSICAL* делает основную базу данных физической резервной, а *LOGICAL* – логической резервной. Затем нужно выполнить команду *ALTER DATABASE START LOGICAL STANDBY APPLY*.

[WITH[OUT] SESSION SHUTDOWN] [WAIT | NOWAIT]

WITH SESSION SHUTDOWN закрывает все активные сессии и откатывает все незафиксированные транзакции для переключения физических резервных баз (но не логических). *WITHOUT SESSION SHUTDOWN* (значение по умолчанию) запрещает выполнение *COMMIT TO SWITCHOVER* при обнаружении активных сессий приложений. *WAIT* возвращает контроль консоли только после завершения команды *SWITCHOVER*, а *NOWAIT* возвращает контроль сразу, не дожидаясь завершения.

START LOGICAL STANDBY APPLY [IMMEDIATE] [NODELAY] [{INITIAL целое_число | NEW PRIMARY имя_dblink} | {FINISH | SKIP FAILED TRANSACTION}]

Начинает применение журнальных файлов на логической резервной базе данных. *IMMEDIATE* указывает, что Oracle LogMiner должен сразу начать чтение журнальных файлов. *NODELAY* игнорирует задержку в применении

журнальных файлов, например, в случае недоступности основной базы данных. *INITIAL* используется при первом применении журнальных файлов на резервной базе данных. Фраза *NEW PRIMARY* необходима после переключения между основной и резервной базами или после того, как на резервной базе данных обработаны все журнальные файлы, и другая резервная база данных становится основной. Используйте *SKIP FAILED TRANSACTION* для пропуска последней транзакции и перезапуска процесса применения журналов. Используйте *FINISH* для применения журнальных файлов, когда основная база данных отключена.

[STOP|ABORT] LOGICAL STANDBY APPLY

Останавливает применение журнальных файлов на резервной базе данных. *STOP* выполняет корректное завершение, *ABORT* выполняет немедленную остановку.

CONVERT TO {PHYSICAL|SNAPSHOT} STANDBY

Переводит основную базу данных или моментальную резервную копию базы данных в физическую резервную базу данных (опция *PHYSICAL*) или переводит физическую резервную базу данных в моментальную резервную копию (опция *SNAPSHOT*).

RENAME GLOBAL_NAME TO имя_базы[.домен[.домен ...]]

Меняет глобальное имя базы данных на новое имя длиной не более 8 байт. Можно дополнительно указать домен, описывающий расположение базы данных в сети. Изменение имени не распространяется на другие объекты, такие как синонимы, хранимые процедуры и т. д.

{DISABLE | ENABLE} BLOCK CHANGE TRACKING [USING FILE 'файл'] [REUSE]

Запускает или останавливает процесс отслеживания физического положения всех измененных данных, с сохранением информации в *файле отслеживания измененных блоков*. Если вы не укажете имя и путь к файлу, используя фразу *USING FILE*, то Oracle автоматически создаст этот файл в каталоге, указанном в параметре *DB_CREATE_FILE_DEST*. *REUSE* говорит, что можно перезаписать содержимое файла, если он уже существует. Фразы *USING* и *REUSE* допустимы только при использовании *ENABLE*.

FLASHBACK {ON|OFF}

Включает (*ON*) или выключает (*OFF*) в базе данных режим *FLASHBACK*. При включенном режиме *FLASHBACK* в базе данных создаются и ведутся специальные журналы в специальной области flash recovery area. При выключении режима эти журналы удаляются и становятся недоступными.

SET TIME ZONE

Устанавливает часовой пояс сервера. Обратитесь к описанию фразы *SET TIME ZONE* оператора *CREATE DATABASE* для дополнительной информации.

{ENABLE|DISABLE}{[PUBLIC] THREAD целое_число |INSTANCE 'имя_экземпляра'}

При работе с кластером (RAC) вы можете включить (*ENABLE*) или выключить (*DISABLE*) журнальный поток с определенным номером. Также можно явно указать *имя_экземпляра*, чтобы включить или выключить поток, привя-

занный к нему. Имя экземпляра может иметь длину до 80 символов. Ключевое слово *PUBLIC* означает, что поток доступен любому экземпляру, иначе поток доступен, только когда он явно запрашивается. В потоке должно быть как минимум две группы журнальных файлов, чтобы он мог быть включен. Для отключения потока требуется, чтобы база данных была открыта, но не примонтирована экземпляром, использующим поток.

GUARD {ALL | STANDBY | NONE}

Защищает данные в базе от изменений. *ALL* разрешает изменение данных только пользователю *SYS*. *STANDBY* разрешает изменение в логической резервной базе только пользователю *SYS*. *NONE* устанавливает обычный режим безопасности.

CREATE DATAFILE 'файл' [, ...] [AS {NEW | определение_файла]

Создает новый пустой файл данных, замещая существующий. Параметр *'файл'* идентифицирует файл (по имени или по номеру), который был потерян или поврежден, без возможности восстановления из резервной копии. *AS NEW* создает новый файл со сгенерированным именем в каталоге, используемом по умолчанию. *AS определение_файла* позволяет указать имя файла и настройки размера, как это описано для опции *TEMPFILE ранее*.

DATAFILE 'файл' | TEMPFILE 'файл'}[, ...] {ONLINE | OFFLINE [FOR DROP] | [RESIZE целое_число [K | M]] | END BACKUP | AUTOEXTEND {OFF | ON [NEXT целое_число [K | M]]} [MAXSIZE [UNLIMITED | целое_число [K | M]]]

Меняет атрибуты, такие как размер одного или нескольких файлов данных или временных файлов. В параметре *'файл'* вы можете указать через запятую несколько имен или номеров файлов. Не используйте одновременно параметры *DATAFILE* и *TEMPFILE*.

ONLINE

Переводит файл в активное состояние.

OFFLINE [FOR DROP]

Переводит файл в неактивное состояние, позволяя проведение восстановления после сбоя носителя. *FOR DROP* требуется для перевода файла в неактивное состояние при работе в режиме *NOARCHIVELOG*, но сам файл не удаляется. В режиме *ARCHIVELOG* эта опция игнорируется.

RESIZE целое_число [K | M]]

Устанавливает новый размер файла.

END BACKUP

Описывается позже в этом разделе. Используется только во фразе *DATAFILE*.

AUTOEXTEND {OFF | ON [NEXT целое_число [K | M]]} [MAXSIZE [UNLIMITED | целое_число [K | M]]]

Уже описано в разделе, посвященном фразе *DATAFILE*.

DROP [INCLUDING DATAFILES]

Удаляет не только временный файл, но и все файлы данных, связанные с временным файлом. В сигнальный файл (alert log) записываются сообщения о каждом удаленном файле. Используется только во фразе *TEMPFILE*.

RENAME FILE 'файл' [, ...] TO 'новое_имя_файла' [, ...]

Переименовывает файл данных, временный файл или журнальный файл. Вы можете переименовать несколько файлов, перечислив старые и новые имена через запятую. Эта команда не переименовывает файлы на уровне операционной системы, а просто указывает, какие имена будут использоваться при открытии этих файлов. На уровне операционной системы переименовать файл вы должны самостоятельно.

[NO] FORCE LOGGING

Включает (*FORCE LOGGING*) или выключает (*NO FORCE LOGGING*) в базе данных режим принудительного журналирования. В этом режиме Oracle сохраняет в журнальных файлах все изменения за исключением изменений во временных файлах и сегментах. Установка принудительного журналирования на уровне базы данных имеет приоритет над любыми установками на уровне табличных пространств.

[NO] ARCHIVELOG [MANUAL]

Говорит о том, что журнальные файлы должны быть созданы, но архивированием оперативных журналов пользователь будет управлять явно. Фраза используется только с оператором *ALTER DATABASE* и *только* для обратной совместимости в системах с архивированием на ленты. Если фраза опущена, то Oracle архивирует журнальные файлы в каталог, указанный в инициализационном параметре *LOG_ARCHIVE_DEST_n*.

[ADD | DROP] SUPPLEMENTAL LOG DATA [(ALL | PRIMARY KEY | UNIQUE | FOREIGN KEY | FOR PROCEDURAL REPLICATION)[, ...] COLUMNS

ADD помещает дополнительную информацию о столбцах в журнальный поток при изменениях данных. При этом обеспечивается *минимальная дополнительная журнализация* (*minimal supplemental logging*), необходимая для того, чтобы LogMiner мог поддерживать расщепленные (chained) строки и специальные механизмы хранения, например кластеризованные таблицы. Вы можете добавить фразы *PRIMARY KEY COLUMNS*, *UNIQUE KEY COLUMNS*, *FOREIGN KEY COLUMNS* или *ALL* (включает все три предыдущие опции), если вам необходима полная ссылочная целостность через внешние ключи в другой базе данных (например, в логической резервной базе данных), или *FOR PROCEDURAL REPLICATION* для журналирования вызовов PL/SQL. Oracle помещает в журнал либо столбцы первичного ключа, либо столбцы уникального ключа (если нет ни первичного, ни уникального ключа, то используется набор столбцов, уникально идентифицирующий запись в таблице), либо столбцы внешнего ключа, либо все вместе. *DROP* приостанавливает дополнительную журнализацию.

[ADD | DROP] [STANDBY] LOGFILE {{[THREAD целое_число | INSTANCE 'имя_экземпляра']} {[GROUP целое_число | имя_файла[, ...]]} [SIZE целое_число [K | M]] | [REUSE] | [MEMBER] 'файл' [REUSE][, ...]

ADD добавляет для указанного экземпляра базы данных одну или несколько основных или резервных (standby) групп журнальных файлов. *THREAD* привязывает добавляемые файлы к определенному потоку (в кластерной среде). По умолчанию используется поток, привязанный к текущему экземпляру. *GROUP* привязывает журнальные файлы к определенной группе внутри по-

тока. *MEMBER* добавляет указанный файл (или несколько файлов, перечисленных через запятую) к существующей журнальной группе. *REUSE* указывается, если файл уже существует. *DROP LOGFILE MEMBER* удаляет один или несколько членов журнальной группы после выполнения оператора *ALTER SYSTEM SWITCH LOGFILE*.

CLEAR [UNARCHIVED] LOGFILE {[GROUP целое_число | имя_файла [, ...]]}[, ...] [UNRECOVERABLE DATAFILE]

Повторно инициализирует один или несколько оперативных журнальных файлов. Фраза *UNRECOVERABLE DATAFILE* необходима, когда какой-либо файл данных переведен в автономный режим, а база данных работает в режиме *ARCHIVELOG*.

CREATE {LOGICAL | PHYSICAL} STANDBY CONTROLFILE AS 'файл' [REUSE]}

Создает управляющий файл для работы логической или физической резервной базы данных. Ключевое слово *REUSE* необходимо в том случае, если файл уже существует.

BACKUP CONTROLFILE TO {'файл' [REUSE] | TRACE [AS 'файл' [REUSE]]} [{RESETLOGS | NORESETLOGS}]

Создает резервную копию управляющего файла открытой или примонтированной базы данных. *TO 'файл'* указывает полный путь к файлу. *TO TRACE* записывает в трассировочный файл SQL-оператор для пересоздания управляющего файла. Ключевое слово *REUSE* необходимо в том случае, если файл уже существует. Ключевое слово *RESETLOGS* инициализирует трассировочный файл при помощи оператора *ALTER DATABASE OPEN RESETLOGS* и допустимо, только когда оперативные журнальные файлы недоступны. Ключевое слово *NORESETLOGS* инициализирует трассировочный файл при помощи оператора *ALTER DATABASE OPEN NORESETLOGS* и допустимо, только когда оперативные журнальные файлы доступны.

RECOVER

Восстанавливает базу данных, резервную базу данных, табличное пространство или файл после сбоя носителя. В Oracle оператор *ALTER DATABASE* является основным средством восстановления базы данных. Используйте *RECOVER*, когда база данных примонтирована (в эксклюзивном режиме), файлы и табличные пространства не используются (находятся в режиме offline) и база данных находится в открытом или закрытом состоянии. Полностью база данных может быть восстановлена только в закрытом состоянии, но отдельные файлы или табличные пространства могут быть восстановлены только при открытой базе данных.

AUTOMATIC [FROM 'путь']

Указывает, что при восстановлении автоматически определяется имя следующего необходимого архивного журнального файла. Если файл не будет найден, то Oracle об этом сообщит. *FROM* указывает, где искать архивные журнальные файлы. При автоматическом восстановлении можно использовать следующие дополнительные фразы:

STANDBY

Указывает, что восстанавливаемая база данных является резервной.

DATABASE {[UNTIL {CANCEL | TIME время | CHANGE целое_число}] | USING BACKUP CONTROLFILE}

Восстанавливает базу данных полностью. Ключевое слово *UNTIL* позволяет восстанавливать базу данных либо до выполнения оператора *ALTER DATABASE RECOVER CANCEL*, либо до определенного времени, указанного в формате ГГГГ-ММ-ДД:чч:мм:сс, либо до достижения определенного системного номера изменения (SCN). Фраза *USING BACKUP CONTROLFILE* позволяет использовать не текущий управляющий файл, а его резервную копию.

[STANDBY] [TABLESPACE имя_табличного_пространства[, ...] | DATAFILE 'файл'[, ...]]

Восстанавливает один или несколько файлов данных или табличных пространств. Вы можете указать несколько файлов данных или табличных пространств в списке через запятую. Также можно указать номера, а не имена файлов данных. Табличное пространство должно быть в нормальном или резервном режиме. Резервное табличное пространство восстанавливается с помощью архивных журнальных файлов и управляющего файла основной базы данных.

UNTIL [CONSISTENT WITH] CONTROLFILE

Восстанавливает старое резервное табличное пространство или файл данных, используя текущий резервный управляющий файл. *CONSISTENT WITH* не несет дополнительной смысловой нагрузки.

LOGFILE имя_файла[, ...]

Продолжает восстановление путем применения одного или нескольких архивных журнальных файлов, перечисленных через запятую.

TEST | ALLOW целое_число CORRUPTION | [NO]PARALLEL целое_число

TEST позволяет провести пробное восстановление для выявления возможных проблем. *ALLOW CORRUPTION* указывает допустимое количество поврежденных блоков данных, при достижении которого восстановление прекращается. Значение должно равняться 1 для реального восстановления, но может быть любым при тестовом. *NOPARALLEL* является значением по умолчанию, при котором идет последовательное чтение диска. *PARALLEL* позволяет проводить восстановление с заданной степенью параллелизма.

CONTINUE [DEFAULT]

Позволяет определить, что многошаговое восстановление продолжается после перерыва. *CONTINUE DEFAULT* – то же самое, что и *RECOVER AUTOMATIC*, только не требует указания имени файла.

CANCEL

Прерывает операцию восстановления при достижении границы очередного журнального файла. При этом процесс восстановления должен быть начат с использованием фразы *UNTIL CANCEL*.

MANAGED STANDBY DATABASE

Определяет режим автоматического восстановления физической резервной базы данных. Эта команда не используется для создания новой базы данных,

а только для восстановления после сбоя носителя. Используются следующие параметры:

USING CURRENT LOGFILE

В этом режиме информация из оперативных журналов применяется по мере поступления в режиме реального времени, еще до того, как эти журнальные файлы заархивированы.

DISCONNECT [FROM SESSION]

Позволяет процессу восстановления выполняться в фоновом режиме, оставляя текущий процесс доступным для других задач. *FROM SESSION* не несет смысловой нагрузки. Опция *DISCONNECT* несовместима с TIMEOUT.

NODELAY

Аннулирует значение *DELAY* параметра *LOG_ARCHIVE_DEST_n* на основной базе данных. Если фраза опущена, то применение журнальных файлов задерживается на время, указанное этим параметром.

UNTIL CHANGE целое_число

Выполняет автоматическое восстановление до достижения (не включительно) указанного номера системного изменения (SCN).

FINISH

Немедленно применяет информацию из оперативных журналов при переводе физической резервной базы данных в режим основной. Фраза *FINISH* известна как *терминальное восстановление* (terminal recovery) и должна использоваться только при сбое основной базы данных.

CANCEL

Немедленно останавливает применение журнальных файлов и возвращает управление после остановки процесса.

TO LOGICAL STANDBY {имя_базы_данных | KEEP IDENTITY}

Преобразует физическую резервную базу данных в логическую. *Имя_базы_данных* идентифицирует новую логическую резервную базу данных. *KEEP IDENTITY* говорит о том, что логическая резервная база данных используется для применения обновлений и не может использоваться для других общих задач.

{BEGIN | END} BACKUP

Управляет режимом оперативного резервирования файлов данных. *BEGIN* переводит файлы данных в режим оперативного резервирования. База данных должна быть примонтирована и открыта, работать в режиме *ARCHIVELOG*. (Имейте в виду, что когда база данных находится в режиме оперативного резервирования, она не может быть остановлена, а табличные пространства не могут быть отключены или переведены в режим «только чтение».) *END* выключает режим оперативного резервирования. База данных должна быть примонтирована, но не обязательно открыта.

После этого длинного описания синтаксиса рассмотрим некоторые базовые концепции Oracle.

В Oracle базы данных бывают двух типов – основные и резервные. Основная база данных находится в примонтированном и открытом состоянии и доступна поль-

зователям. *Основная* база данных регулярно отправляет свои журналы повторного выполнения на *резервную* базу данных. С помощью этих журнальных файлов резервная база данных постоянно восстанавливается и таким образом является актуальной копией основной базы данных.

Уникальной особенностью Oracle является использование файла параметров INIT.ORA, в котором задаются имя базы данных и множество других параметров. В INIT.ORA должны быть определены все параметры, например имена управляющих файлов, необходимые для запуска базы данных, иначе база данных не будет запущена. Начиная с Oracle 9.1 вы можете использовать двоичный файл параметров вместо INIT.ORA.

При создании группы журнальных файлов список файлов нужно заключать в скобки. Скобки необязательны при создании группы только с одним членом, но так делается очень редко. Вот пример создания группы журнальных файлов:

```
CREATE DATABASE publications
LOGFILE ('/s01/oradata/loga01',
         '/s01/oradata/loga02') SIZE 5M
DATAFILE;
```

В этом примере создается база данных с названием publications с явно создаваемыми журнальными файлами и автоматически создаваемым файлом данных. Следующий пример оператора *CREATE DATABASE* более сложный:

```
CREATE DATABASE sales_reporting
CONTROLFILE REUSE
LOGFILE
  GROUP 1 ('diskE:log01.log', 'diskF:log01.log') SIZE 15M,
  GROUP 2 ('diskE:log02.log', 'diskF:log02.log') SIZE 15M
MAXLOGFILES 5
MAXLOGHISTORY 100
MAXDATAFILES 10
MAXINSTANCES 2
ARCHIVELOG
CHARACTER SET AL32UTF8
NATIONAL CHARACTER SET AL16UTF16
DATAFILE
  'diskE:sales_rpt1.dbf' AUTOEXTEND ON,
  'diskF:sales_rpt2.dbf' AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED
DEFAULT TEMPORARY TABLESPACE temp_tblspc
UNDO TABLESPACE undo_tblspc
SET TIME_ZONE = '-08:00';
```

В этом примере явно создаются журнальные файлы и файлы данных, а также указывается используемая кодировка. Также мы задаем некоторые параметры базы данных, такие как работа в режиме *ARCHIVELOG*, перезапись управляющего файла (*CONTROLFILE REUSE*), часовой пояс, максимальное число экземпляров, файлов данных и т. д. В этом примере предполагается, что в файле INIT.ORA установлен параметр *DB_CREATE_FILE_DEST*, поэтому можно не создавать вручную табличные пространства для временных данных и данных отката с помощью фраз *DEFAULT TEMPORARY TABLESPACE* и *UNDO TABLESPACE*.

При выполнении пользователем с привилегией *SYSDBA* этот оператор создает базу данных и делает ее доступной пользователям в монопольном или параллельном режиме, в зависимости от параметра *CLUSTER_DATABASE*. Любые данные, которые могут находиться в указанных файлах данных, удаляются. Как правило, вы захотите затем создать табличные пространства и сегменты отката. (Обращайтесь к официальной документации за информацией о командах *CREATE TABLESPACE* и *CREATE ROLLBACK SEGMENT*.)

В новой версии Oracle усилен контроль за безопасностью в отношении стандартных пользователей, создаваемых по умолчанию при установке. Многие пользователи теперь создаются заблокированными. В 11g остались без изменений только пользователи *SYS*, *SYSTEM*, *SCOTT*, *DBSNMP*, *OUTLN*, *AURORAJISUTILITY$*, *AURORA$ORB$UNAUTHENTICATED* и *OSE$HTTP$ADMIN*. Вы должны вручную разблокировать и установить новый пароль всем заблокированным пользователям, равно как и задать пароли для *SYS* и *SYSTEM* при установке.

В следующем примере мы создаем в существующей базе данных новые журнальные файлы, а затем добавляем файлы данных:

```
ALTER DATABASE ADD LOGFILE GROUP 3
  ('diskf: log3.sales_arch_log','diskg:log3.sales_arch_log')
SIZE 50M;
ALTER DATABASE sales_archive
CREATE DATAFILE 'diskF:sales_rpt4.dbf'
AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED;
```

Мы можем выбрать новое временное табличное пространство, используемое по умолчанию, что демонстрируется в следующем примере:

```
ALTER DATABASE DEFAULT TEMPORARY TABLESPACE sales_tbl_spc_2;
```

Дальше мы можем выполнить простое полное восстановление (recover) базы данных:

```
ALTER DATABASE sales_archive RECOVER AUTOMATIC DATABASE;
```

В следующем примере мы выполняем более сложное частичное восстановление:

```
ALTER DATABASE
  RECOVER STANDBY DATAFILE 'diskF:sales_rpt4.dbf'
UNTIL CONTROLFILE;
```

Теперь мы выполним простое восстановление резервной базы данных в режиме управляемого восстановления:

```
ALTER DATABASE RECOVER sales_archive MANAGED STANDBY DATABASE;
```

В следующем примере мы переключаемся с основной базы данных на логическую резервную базы данных, делая последнюю основной:

```
-- Делаем основную базу данных резервной
ALTER DATABASE COMMIT TO SWITCHOVER TO LOGICAL STANDBY;
-- На новой резервной базе начинаем процесс применения
-- журналов повторного выполнения
ALTER DATABASE START LOGICAL STANDBY APPLY;
-- Делаем текущую резервную базу данных основной
ALTER DATABASE COMMIT TO SWITCHOVER TO PRIMARY;
```

PostgreSQL

В PostgreSQL оператор *CREATE DATABASE* создает базу данных и определяет место расположения файлов данных:

```
CREATE DATABASE имя_базы_данных [ WITH ]
  [OWNER [=] владелец]
  [TEMPLATE [=] имя_шаблона]
  [ENCODING [=] кодировка]
  [TABLESPACE [=] имя_табличного_пространства]
  [CONNECTION LIMIT [=] целое_число]
```

Для команды *ALTER DATABASE* используется следующий синтаксис:

```
ALTER DATABASE имя_базы_данных [ WITH ]
  [CONNECTION LIMIT целое_число]
  [OWNER TO новый_владелец]
  [RENAME TO новое_имя_базы_данных]
  [RESET параметр]
  [SET параметр {TO | =} {значение | DEFAULT}]
```

где:

WITH

Необязательное ключевое слово для дальнейшего описания детальных настроек базы данных.

OWNER [=] владелец

Указывает имя владельца базы данных, если владельцем должен быть не тот пользователь, который выполняет оператор.

TEMPLATE [=] имя_шаблона

Указывает шаблон, используемый при создании базы данных. Можно не использовать эту фразу и принять шаблон, используемый по умолчанию (или использовать фразу *TEMPLATE = DEFAULT*). По умолчанию используется шаблон **template1**. Вы можете создать базу данных, содержащую только минимально необходимый набор объектов, указав *TEMPLATE = template0*.

ENCODING [=] кодировка

Указывает многобайтную кодировку, используемую в создаваемой базе данных. В качестве значения указывается либо название кодировки (например, 'SQL_ASCII'), либо ее номер, либо ключевое слово *DEFAULT* для использования кодировки по умолчанию.

TABLESPACE [=] имя_табличного_пространства

Указывает имя табличного пространства, связанного с базой данных.

CONNECTION_LIMIT [=] целое_число

Определяет максимальное допустимое число одновременных подключений к базе данных. Значение -1 означает отсутствие ограничений.

RENAME TO новое_имя_базы_данных

Устанавливает для базы данных новое имя.

RESET параметр | *SET параметр* {*TO* | = } { *значение* | *DEFAULT*}

Устанавливает (*SET*) значение параметра базы данных или сбрасывает его в значение по умолчанию (*RESET*).

Например, для создания базы данных **sales_revenue** в каталоге */home/teddy/private_db* можно использовать следующую команду:

```
CREATE DATABASE sales_revenue
WITH LOCATION = '/home/teddy/private_db';
```

Будьте осторожны при использовании абсолютных путей к файлам и каталогам, так как это влияет на безопасность и целостность данных.

Шаблонные базы данных в PostgreSQL следует использовать только для чтения. Их не следует использовать для копирования функциональности баз данных.

PostgreSQL *не поддерживает* оператор *ALTER DATABASE*.

SQL Server

SQL Server предлагает высокую степень контроля над структурами файловой системы, в которых хранится база данных и ее объекты. Синтаксис оператора *CREATE DATABASE* выглядит в SQL Server следующим образом:

```
CREATE DATABASE имя_базы_данных
[ ON определение_файла[, ...] ]
[, FILEGROUP имя_файловой_группы определение_файла [, ...] ]
[ LOG ON определение_файла [, ...] ]
[ COLLATE схема_упорядочения]
[ FOR { ATTACH [WITH {ENABLE_BROKER | NEW_BROKER | ERROR_BROKER_
CONVERSATIONS}] |
ATTACH_REBUILD_LOG }
[WITH [DB_CHAINING {ON | OFF}][, TRUSTWORTHY {ON | OFF}]] ]
[ AS SNAPSHOT OF источник]
```

Синтаксис оператора *ALTER DATABASE* следующий:

```
ALTER DATABASE имя_базы_данных
{ADD FILE определение_файла [, ...] [TO FILEGROUP имя_файловой_группы]
| ADD LOG FILE определение_файла [, ...]
| REMOVE FILE имя_файла
| ADD FILEGROUP имя_файловой_группы
| REMOVE FILEGROUP имя_файловой_группы
| MODIFY FILE определение_файла
| MODIFY NAME = новое_имя_базы_данных
| MODIFY FILEGROUP имя_файловой_группы
{NAME = новое_имя_файловой_группы | свойство_файловой_группы
{READONLY | READWRITE | DEFAULT}}
| SET {параметры_состояния | параметры_курсоров
| авто_параметры | параметры_sql | параметры_восстановления}
```

```
[, ...] [WITH параметры_завершения]
| COLLATE схема_упорядочения}
```

Параметры имеют следующий смысл:

{CREATE|ALTER} DATABASE имя_базы_данных

Создает или изменяет существующую базу данных. Имя базы данных должно быть не длиннее 128 символов. Если же вы не указываете логическое имя файла, то имя базы данных должно быть не длиннее 123 символов, так как логическое имя файла образуется из имени базы данных путем добавления суффикса.

{ON|ADD} определение_файла [,...]

Создает (в *CREATE DATABASE*) или добавляет (в *ALTER DATABASE*) файлы, в которых хранятся объекты базы данных. Ключевое слово *ON* в операторе *CREATE DATABASE* обязательно только в том случае, если вы хотите задать одно или несколько определений файлов. Синтаксис определения файла следующий:

```
{[PRIMARY] ( [NEW][NAME = имя_файла]
[, FILENAME = {'имя_файла_в_ОС' | 'имя_файлового_потока'}]
[, SIZE = целое_число [KB | MD | GB | TB]]
[, MAXSIZE = { целое_число | UNLIMITED }]
[, FILEGROWTH = целое_число ][, OFFLINE] )}[, ...]
```

где:

PRIMARY

Указывает, что файл является основным. В базе данных может быть только один основной файл. (Если вы не указываете явно, какой из файлов является основным, то SQL Server сделает основным автоматически создаваемый файл или первый из создаваемых пользователем файлов). Основной файл или файловая группа содержит логическое начало базы данных, все системные таблицы и все остальные объекты, которые не содержатся в пользовательских файловых группах.

[NEW]NAME = имя_файла

Задает логическое имя определяемого файла в операторе *CREATE DATABASE*. В операторе *ALTER DATABASE* используйте *NEWNAME* для задания нового логического имени файла. В любом случае логическое имя файла должно быть уникально в пределах базы данных. Эта фраза необязательна при использовании *FOR ATTACH*.

FILENAME = { 'имя_файла_в_ОС' | 'имя_файлового_потока' }

Указывает путь и имя файла в операционной системе. Файл не должен находиться в сжатом каталоге. Для разделов без файловой системы (raw partition) укажите только букву, обозначающую раздел.

SIZE = целое_число [KB|MD|GB|TB]

Устанавливает размер создаваемого файла. Эта фраза необязательна, но если ее не указать, то будет использован размер основного файла модельной базы данных, а он обычно очень мал. Размер по умолчанию для журнальных файлов и дополнительных файлов данных – 1 Мбайт. Значение размера по умолчанию задается в мегабайтах, но можно явно использо-

вать суффикс для задания размера в килобайтах (*KB*), мегабайтах (*MB*), гигабайтах (*GB*) или терабайтах (*TB*). Размер не может быть меньше, чем 512 Кб или размер основного файла модельной базы данных.

MAXSIZE = { целое_число | UNLIMITED }

Устанавливает максимальный размер, до которого может увеличиться файл. Можно использовать суффиксы для определения единиц, в которых указывается размер (как и для опции *SIZE*). *UNLIMITED* (значение по умолчанию) позволяет файлу увеличиваться до тех пор, пока не будет заполнено все свободное место на диске. Опция не используется для файлов на дисках без файловой системы.

FILEGROWTH = целое_число

Устанавливает размер, на который увеличивается файл за один раз. Можно использовать суффиксы для определения единиц, в которых указывается размер (как и для опции *SIZE*). Можно использовать в качестве суффикса знак процента (%) для указания того, что файл должен увеличиться на некоторый процент от своего текущего размера. Если не использовать фразу *FILEGROWTH*, то файл за раз будет увеличиваться на 10%, но не меньше чем на 64 Кб. Опция не используется для файлов на дисках без файловой системы.

OFFLINE

Переводит файл в неактивный режим, делая недоступными все объекты, содержащиеся в этом файле. Эту опцию следует использовать только в том случае, когда файл поврежден.

[ADD] LOG {ON | FILE} определение_файла

Создает (в *CREATE DATABASE*) или добавляет (в *ALTER DATABASE*) файлы для хранения журнала базы данных. Вы можете указать одно или несколько определений файлов через запятую. Полный синтаксис определения файла описан в предыдущих абзацах.

REMOVE FILE имя_файла

Удаляет файл из базы данных и стирает физический файл с диска. Файл предварительно должен быть очищен от содержимого.

[ADD] FILEGROUP имя_файловой_группы [определение_файла]

Создает пользовательские файловые группы и определяет их файлы. Во всех базах данных есть как минимум основная файловая группа (хотя часто основная файловая группа создается по умолчанию). Создание новых файловых групп и добавление в них файлов позволяет лучше контролировать дисковый ввод-вывод. (Тем не менее, мы не рекомендуем добавлять файловые группы без предварительного анализа и тестирования.)

REMOVE FILEGROUP имя_файловой_группы

Удаляет из базы данных файловую группу и все ее файлы. Файлы и файловые группы должны быть предварительно очищены.

MODIFY FILE определение_файла

Меняет определение файла. Эта фраза очень похожа на *[ADD] LOG {ON | FILE}*. Например: *MODIFY FILE (NAME = имя_файла, NEWNAME = новое_имя_файла, SIZE = ...)*.

MODIFY NAME = *новое_имя_базы_данных*

Меняет имя базы данных на новое.

MODIFY FILEGROUP имя_файловой_группы {NAME = новое_имя_файловой_группы | свойство_файловой_группы}

Используется в операторе *ALTER DATABASE*, имеет два варианта использования. Первый вариант позволяет изменить имя файловой группы, например: *MODIFY FILEGROUP имя_файловой_группы NAME = новое_имя*. Другой вариант позволяет установить одно из следующих свойств файловой группы:

READONLY

Переводит файловую группу в режим «только чтение» и запрещает изменение любых объектов, хранящихся в файловой группе. Свойство *READONLY* может быть установлено только пользователем с эксклюзивным доступом к базе данных и не может быть установлено для основной файловой группы. Можно также использовать написание *READ_ONLY*.

READWRITE

Отключает режим *READONLY* и позволяет выполнять изменения объектов, хранящихся в файловой группе. Свойство *READWRITE* может быть установлено только пользователем с эксклюзивным доступом к базе данных. Можно также использовать написание *READ_WRITE*.

DEFAULT

Назначает файловую группу используемой по умолчанию. Все таблицы и индексы создаются в этой файловой группе, за исключением объектов, для которых явно указана какая-либо другая файловая группа. В базе данных может быть только одна группа, используемая по умолчанию. (Оператор *CREATE DATABASE* назначает основную файловую группу используемой по умолчанию).

SET { *параметры_состояния* | *параметры_курсоров* | *авто_параметры* | *параметры_sql* | *параметры_восстановления* }

Определяет широкий набор параметров поведения базы данных. Конкретные параметры рассматриваются позднее в этой главе.

WITH параметр_завершения

Используется совместно с опцией *SET* для управления поведением транзакций, которые еще не завершились на момент изменения параметров базы данных. Если эта фраза опущена, то при изменении параметров базы данных все транзакции должны самостоятельно завершиться. *Параметр_завершения* может принимать два значения:

ROLLBACK AFTER целое_число [SECONDS] | ROLLBACK IMMEDIATE

Откатывает все транзакции через заданное количество секунд либо немедленно. Слово *SECONDS* необязательно и не меняет поведения фразы *ROLLBACK AFTER*.

NO_WAIT

В этом режиме не производится ожидание завершения текущих транзакций и изменение параметров базы данных завершается с ошибкой, если это изменение не может быть выполнено немедленно.

COLLATE схема_упорядочения

Устанавливает или изменяет текущую схему упорядочения. Может быть указано имя схемы упорядочения SQL Server или Windows. По умолчанию для новых баз данных устанавливается такая же схема упорядочения, как и у экземпляра SQL Server. (Вы можете выполнить запрос *SELECT * FROM ::fn_helpcollations( )* для получения списка всех доступных схем упорядочения.) Чтобы изменить используемую схему упорядочения, нужно быть единственным подключенным к базе данных пользователем, не должно быть объектов, привязанных к схеме (schema-bound objects), зависящих от текущей схемы упорядочения, и изменение не должно приводить к появлению дубликатов в именах любых объектов базы данных.

FOR { ATTACH [WITH {ENABLE_BROKER | NEW_BROKER | ERROR_BROKER_CONVERSATIONS}] | ATTACH_REBUILD_LOG }

Переводит базу данных в специальный режим запуска. *FOR ATTACH* создает базу данных из набора уже существующих файлов операционной системы (почти всегда это файлы ранее созданной базы данных). Поэтому новая база данных должна иметь те же кодовую страницу и схему упорядочения, что и предыдущая база данных. Требуется указывать определение только первого основного файла или тех файлов, путь к которым изменился с момента последнего запуска базы данных. Фраза *FOR ATTACH_REBUILD_LOG* указывает, что база данных создается подключением существующих файлов, с перестроением журнальных файлов в случае их отсутствия. В общем случае вместо оператора *CREATE DATABASE FOR ATTACH* следует использовать системную хранимую процедуру sp_attach_db, за исключением случаев, когда вы хотите указать определения более чем 16 файлов.

При использовании *FOR ATTACH* можно указывать опции компоненты Service Broker:

ENABLE_BROKER

Указывает, что в базе данных компонента Service Broker активна.

NEW_BROKER

Создает новый service_broker_guid, закрывает и очищает все точки подключения.

ERROR_BROKER_CONVERSATIONS

Завершает все диалоги Service Broker с сообщением об ошибке, означающем, что база данных прикрепляется или восстанавливается. Service Broker останавливается на время выполнения операции, а затем опять активируется.

WITH [DB_CHAINING {ON | OFF}][, TRUSTWORTHY {ON | OFF}]

Указывает, что база данных может участвовать в кросс-вызовах по цепочке принадлежностей (значение *DB_CHAINING ON*). Значение по умолчанию *OFF* означает, что база данных не участвует в цепочках принадлежностей. Опция *TRUSTWORTHY ON* разрешает объектам базы данных, использующим имперсонификацию, работать с внешними по отношению к базе данных ресурсами. В режиме OFF доступ к внешним ресурсам при имперсонификации запрещен. Опция *TRUSTWORTHY* имеет значение OFF всегда, когда ба-

за данных подключена. Опции *DB_CHAINING* и *TRUSTWORTHY* не могут иметь значение *ON* для модельной базы данных, баз **master** и **tempdb**.

AS SNAPSHOT OF источник

Указывает, что база данных создается как моментальный снимок исходной базы данных. Обе базы данных (моментальный снимок и копия) должны находиться в одном экземпляре SQL Server.

Оператор *CREATE DATABASE* нужно выполнять из системной базы данных master. В принципе, можно выполнить оператор *CREATE DATABASE имя_базы_данных* без каких-либо дополнительных параметров, при этом получится очень маленькая база данных с настройками по умолчанию.

SQL Server использует *файлы* (ранее называвшиеся устройствами) в качестве хранилища базы данных. Файлы образуют *файловые группы*, причем в каждой базе данных должна быть ровно одна основная (*PRIMARY*) группа. База данных может храниться в одном или нескольких файлах или файловых группах. SQL Server позволяет хранить журнал транзакций отдельно от данных, для этого используется фраза *LOG ON*. Все эти возможности позволяют планировать размещение файлов для достижения оптимального контроля за нагрузкой на диски. Например, мы можем создать базу данных **sales_report** с файлом данных и журнальным файлом:

```
USE master
GO
CREATE DATABASE sales_report
ON
( NAME = sales_rpt_data, FILENAME =
  'c:\mssql\data\salerptdata.mdf',
  SIZE = 100, MAXSIZE = 500, FILEGROWTH = 25 )
LOG ON
( NAME = 'sales_rpt_log',
  FILENAME = 'c:\mssql\log\salesrptlog.ldf',
  SIZE = 25MB, MAXSIZE = 50MB,
  FILEGROWTH = 5MB )
GO
```

При создании базы данных в нее копируются объекты модельной базы данных. Затем инициализируется все свободное пространство в создаваемых файлах, поэтому создание большой базы данных может занять некоторое время, особенно на медленных дисках.

В базе данных всегда есть как минимум основной файл данных и журнальный файл, но могут быть и дополнительные файлы как для данных, так и для компонент журнала базы данных. По умолчанию SQL Server использует следующие расширения файлов: *.mdf* для основных файлов данных, *.ndf* для дополнительных файлов, *.ldf* для журналов транзакций. В следующем примере создается база данных **sales_archive** с несколькими файлами данных, собранными в файловые группы:

```
USE master
GO
CREATE DATABASE sales_archive
```

```
ON
PRIMARY
    (NAME = sales_arch1,
      FILENAME = 'c:\mssql\data\archdata1.mdf',
      SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
    (NAME = sales_arch2,
      FILENAME = 'c:\mssql\data\archdata2.ndf',
      SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
    (NAME = sales_arch3,
      FILENAME = 'c:\mssql\data\archdat3.ndf',
      SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB)
FILEGROUP sale_rpt_grp1
    (NAME = sale_rpt_grp1_1_data,
      FILENAME = 'c:\mssql\data\SG1Fi1dt.ndf',
      SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
    (NAME = sale_rpt_grp1_1_data,
      FILENAME = 'c:\mssql\data\SG1Fi2dt.ndf',
      SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
FILEGROUP sale_rpt_grp2
    (NAME = sale_rpt_grp2_1_data,
      FILENAME = 'c:\mssql\data\SRG21dt.ndf',
      SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
    (NAME = sale_rpt_grp2_2_data,
      FILENAME = 'c:\mssql\data\SRG22dt.ndf',
      SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
LOG ON
    (NAME = sales_archlog1,
      FILENAME = 'd:\mssql\log\archlog1.ldf',
      SIZE = 100GB, MAXSIZE = UNLIMITED, FILEGROWTH = 25%),
    (NAME = sales_archlog2,
      FILENAME = 'd:\ mssql\log\archlog2.ldf',
      SIZE = 100GB, MAXSIZE = UNLIMITED, FILEGROWTH = 25%)
GO
```

Фраза *FOR ATTACH* обычно используется в ситуациях, когда, к примеру, продавец путешествует с базой данных, скопированной на компакт-диск. Эта фраза говорит SQL Server о том, что база данных подключается из набора существующих файлов, записанных на CD-ROM, DVD-ROM или переносной жесткий диск. При использовании *FOR ATTACH* новая база данных наследует все объекты и данные родительской базы данных, а не модельной.

В следующем примере показывается, как сменить имя базы данных, файла или файловой группы:

```
-- Переименовываем базу данных
ALTER DATABASE sales_archive MODIFY NAME = sales_history
GO
-- Переименовываем файл
ALTER DATABASE sales_archive MODIFY FILE
NAME = sales_arch1,
NEWNAME = sales_hist1
GO
-- Переименовываем файловую группу
ALTER DATABASE sales_archive MODIFY FILEGROUP
```

```
sale_rpt_grp1
NAME = sales_hist_grp1
GO
```

Бывают ситуации, когда требуется добавить свободное пространство в базу данных, в частности, если вы не используете автоматическое расширение файлов:

```
USE master
GO
ALTER DATABASE sales_report ADD FILE
( NAME = sales_rpt_added01, FILENAME = 'c:\mssql\data\salerptadded01.mdf',
  SIZE = 50MB, MAXSIZE = 250MB, FILEGROWTH = 25MB )
GO
```

При изменении базы данных можно настроить различные параметры ее поведения. Параметры состояния контролируют доступ пользователей к базе данных. Список возможных параметров состояния следующий:

SINGLE_USER | RESTRICTED_USER | MULTI_USER

Устанавливает число и тип пользователей, имеющих доступ к базе данных. Режим *SINGLE_USER* позволяет только одно пользовательское подключение в каждый момент времени. Режим *RESTRICTED_USER* позволяет подключаться к базе данных только пользователям с системными ролями *db_owner*, *dbcreator* и *sysadmin*. Режим *MULTI_USER*, использующийся по умолчанию, разрешает одновременную работу с базой данных всех пользователей, обладающих достаточными привилегиями.

OFFLINE | ONLINE

Делает базу данных доступной (*ONLINE*) или недоступной (*OFFLINE*).

READ_ONLY | READ_WRITE

Делает базу данных доступной только для чтения (*READ_ONLY*) или для чтения-записи (*READ_WRITE*). Базы данных в режиме *READ_ONLY* могут быть очень быстрыми при выполнении запросов, так как в этом режиме почти не используются блокировки.

Параметры курсоров определяют поведение курсоров. В синтаксисе *ALTER DATABASE*, приведенном ранее, вы можете заменить *параметры_курсоров* на любое из следующих ключевых слов:

CURSOR_CLOSE_ON_COMMIT { ON | OFF }

Если для параметра установлено значение *ON*, то все открытые курсоры закрываются при фиксации или откате транзакции. Если установлено значение *OFF*, то курсоры остаются открытыми при фиксации транзакции и закрываются при откате, за исключением курсоров типа *INSENSITIVE* и *STATIC*.

CURSOR_DEFAULT { LOCAL | GLOBAL }

Устанавливает область видимости по умолчанию для курсоров в значение *LOCAL* или *GLOBAL*. (Детали приводятся далее в этой главе.)

Во фразе *SET* *авто_параметры* контролируют автоматическую работу базы данных с файлами. Допустимые значения следующие:

AUTO_CLOSE {ON|OFF}

Если установлено значение *ON*, то при отключении последнего пользователя база данных останавливается и освобождаются все ресурсы. Если установлено значение *OFF*, то при отключении последнего пользователя база данных продолжает работать. *OFF* является значением по умолчанию.

AUTO_CREATE_STATISTICS {ON|OFF}

Если для параметра установлено значение *ON*, то SQL Server в процессе оптимизации запроса автоматически собирает недостающую статистику. При значении *OFF* статистика автоматически не собирается. *ON* является значением по умолчанию.

AUTO_SHRINK {ON|OFF}

Если для параметра установлено значение *ON*, то файлы базы данных могут быть автоматически сжаты (SQL Server периодически пытается сжать файлы, хотя делает это не всегда в предсказуемые моменты). Если установлено значение *OFF*, то файл данных будет сжиматься, только если вы явно выполните соответствующую команду. Значение *OFF* установлено по умолчанию.

AUTO_UPDATE_STATISTICS {ON|OFF}

При установленном значении *ON* устаревшая статистика обновляется автоматически в процессе оптимизации запросов. При значении *OFF* статистика обновляется только при явном выполнении команды *UPDATE STATISTICS*.

Параметры_sql настраивают совместимость базы данных со стандартом ANSI. Для включения режима максимальной совместимости со стандартом ANSI SQL92 вы можете использовать команду *SET ANSI_DEFAULTS ON*, вместо того чтобы настраивать каждый параметр отдельно. Во фразе *SET* вы можете заменить *параметры_sql* на любое из следующих значений:

ANSI_NULL_DEFAULT {ON|OFF}

При установленном значении *ON* все столбцы таблицы, для которых в операторе *CREATE TABLE* не указана допустимость пустых значений, по умолчанию создаются как *NULL (допускаются значения NULL)*. Если для параметра установлено значение *OFF*, то по умолчанию для столбцов используется *NOT NULL*. *OFF* является значением по умолчанию.

ANSI_NULLS {ON|OFF}

При установленном значении *ON* любое сравнение с NULL имеет результат *UNKNOWN*. Если для параметра установлено значение *OFF*, то результатом сравнения является NULL, только если оба значения равны NULL. *OFF* является значением по умолчанию.

ANSI_PADDING {ON|OFF}

При установленном значении *ON* замыкающие пробелы в строках с типом *VARCHAR* и *VARBINARY* при вставке или сравнении не усекаются. Если для параметра установлено значение *OFF*, то замыкающие пробелы (или замыкающие нули в случае двоичных значений) усекаются. По умолчанию установлено значение *ON* (и мы крайне не рекомендуем его менять!)

ANSI_WARNINGS {ON | OFF }

При установленном значении *ON* база данных выдает предупреждения при возникновении проблем типа «деление на ноль» или «NULL при группировке». Если для параметра установлено значение *OFF*, то предупреждения не создаются. *OFF* является значением по умолчанию.

ARITHABORT {ON | OFF }

При установленном значении *ON* ошибки переполнения и деления на ноль вызывают остановку выполнения запроса или пакета команд. Если для параметра установлено значение *OFF*, то возникают предупреждения об ошибочных операциях, но обработка продолжается. По умолчанию используется *ON* (и мы не рекомендуем менять это значение!)

CONCAT_NULL_YIELDS_NULL {ON | OFF }

При установленном значении *ON* конкатенация строки и значения NULL дает NULL. Если для параметра установлено значение *OFF*, то при конкатенации NULL обрабатывается как пустая строка. По умолчанию используется значение *OFF*.

NUMERIC_ROUNDABORT {ON | OFF }

При установленном значении *ON* потеря точности числового значения вызывает ошибку. Если для параметра установлено значение *OFF*, то при потере точности выполняется округление. По умолчанию используется *OFF*.

QUOTED_IDENTIFIER {ON | OFF }

При установленном значении *ON* двойные кавычки используются в качестве ограничителей идентификаторов, содержащих специальные символы и зарезервированные слова (например, таблица с названием **SELECT**). Если для параметра установлено значение *OFF*, то идентификаторы не могут содержать специальные символы и зарезервированные слова, а двойные кавычки используются для строковых литералов. *OFF* является значением по умолчанию.

RECURSIVE_TRIGGERS {ON | OFF }

Значение *ON* разрешает рекурсивное срабатывание триггеров. То есть действия одного триггера могут запускать другой триггер и т. д. Если для параметра установлено значение *OFF*, то один триггер не может быть запущен другим триггером. *OFF* является значением по умолчанию.

Параметры_восстановления определяют возможности восстановления базы данных. Используйте в операторе *ALTER DATABASE* любое из следующих значений:

RECOVERY {FULL | BULK_LOGGED | SIMPLE }

В режиме *FULL* резервные копии базы данных и журналы транзакций обеспечивают полное восстановление даже для массовых операций, например *SELECT...INTO*, *CREATE INDEX*. *FULL* является значением по умолчанию в редакциях Standard Edition и Enterprise Edition. *FULL* обеспечивает максимальный уровень восстанавливаемости данных, даже после потери дисковых носителей, но при этом использует больше места. В режиме *BULK_LOGGED* используется минимальное журналирование массовых операций. Этот режим обеспечивает экономию дискового пространства и меньшее количество операций ввода-вывода, но увеличивается риск потери данных. В режиме

SIMPLE базу данных можно восстанавливать только на момент последнего полного или дифференциального резервного копирования. *SIMPLE* является значением по умолчанию в редакциях Desktop Edition и Personal Edition.

TORN_PAGE_DETECTION {ON | OFF }

При установленном значении *ON* SQL Server проверяет отдельно каждый 512-байтный сектор 8-килобайтной страницы для обнаружения неполных операций ввода-вывода. (Поврежденные страницы обычно обнаруживаются в процессе восстановления.) По умолчанию установлено значение *ON*.

Например, мы можем поменять некоторые настройки базы данных **sales_report** без изменения файловой структуры:

```
ALTER DATABASE sales_report SET ONLINE, READ_ONLY,
AUTO_CREATE_STATISTICS ON
GO
```

В этом примере база данных делается активной и переводится в режим «только чтение». Также для параметра *AUTO_CREATE_STATISTICS* устанавливается значение *ON*.

См. также

CREATE SCHEMA

DROP

CREATE/ALTER FUNCTION/PROCEDURE

Операторы *CREATE FUNCTION* и *CREATE PROCEDURE* очень похожи в синтаксисе и использовании (как и аналогичные операторы *ALTER*).

Оператор *CREATE PROCEDURE* используется для создания хранимых процедур, принимающих на вход аргументы и выполняющих условную обработку различных объектов базы данных. В соответствии со стандартом ANSI хранимые процедуры не возвращают значений (хотя могут иметь выходные параметры). К примеру, вы можете использовать хранимую процедуру для закрытия финансового года.

Оператор *CREATE FUNCTION* создает пользовательские функции (user-defined function или UDF), принимающие на вход аргументы и возвращающие результат, как это делают системные функции вроде *CAST( )* или *UPPER( )*. Созданные функции можно использовать в запросах и в операциях с данными, такими как *INSERT* и *UPDATE*, а также во фразе *WHERE* оператора *DELETE*. В главе 4 описаны системные функции и их реализации у разных производителей.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с ограничениями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

Для создания хранимой процедуры или функции используйте следующий синтаксис:

```
CREATE {PROCEDURE | FUNCTION} имя_объекта
  ( [{[IN | OUT | INOUT] [имя_параметра] тип_данных
    [AS LOCATOR] [RESULT]} [, ...]] )
  [ RETURNS тип_данных [AS LOCATOR]
    [CAST FROM тип_данных [AS LOCATOR]] ]
[LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]
[PARAMETER STYLE {SQL | GENERAL}]
[SPECIFIC специальное_имя]
[DETERMINISTIC | NOT DETERMINISTIC]
[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
[DYNAMIC RESULT SETS целое_число]
[STATIC DISPATCH] блок_кода
```

Используйте следующий синтаксис для изменения существующей хранимой процедуры или функции:

```
ALTER {PROCEDURE | FUNCTION} имя_объекта
  [( {имя_параметра тип_данных }[, ...] )]
[NAME новое_имя_объекта]
[LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]
[PARAMETER STYLE {SQL | GENERAL}]
[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
[DYNAMIC RESULT SETS целое_число]
[CASCADE | RESTRICT]
```

Ключевые слова

CREATE {PROCEDURE | FUNCTION} *имя_объекта*

Создает новую хранимую процедуру или пользовательскую функцию с заданным именем. Функция возвращает результат, а процедура (по стандарту ANSI) – нет.

([{[IN | OUT | INOUT] [имя_параметра] тип_данных [AS LOCATOR] [RESULT]} [, ...]])

Определяет список параметров (в скобках, через запятую), передаваемых в хранимую процедуру. Параметры могут быть входными (IN), выходными (OUT) или быть и входными и выходными одновременно (INOUT).

Для определения параметров используется следующий синтаксис:

```
[{IN | OUT | INOUT}] имя_параметра_1 тип_данных,
[{IN | OUT | INOUT}] имя_параметра_2 тип_данных,[...]
```

Если вы указываете имя параметра, то убедитесь, что это имя уникально в пределах процедуры. Необязательная фраза *AS LOCATOR* используется для валидации внешних процедур с выходными параметрами типа *BLOB*, *CLOB*, *NCLOB*, *ARRAY* или пользовательского типа. Если вы хотите на лету менять тип данных выходного параметра, то используйте фразу *CAST* (описание

функции *CAST* приводится в главе 4), например: *RETURNS VARHCAR(12) CAST FROM DATE*. При использовании в операторе *ALTER* эта фраза применяется для добавления параметров в существующую хранимую процедуру. Информация о типах данных приводится в главе 2.

RETURNS тип_данных [AS LOCATOR] [CAST FROM тип_данных [AS LOCATOR]]

Определяет тип данных значения, возвращаемого функцией. (Фраза используется только в операторе *CREATE FUNCTION* и не нужна при создании хранимых процедур.) Основное назначение пользовательской функции – возврат значения.

LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}

Определяет язык, на котором написана хранимая процедура. Большинство баз данных не поддерживают все перечисленные языки, но могут поддерживать и некоторые языки не из этого списка, такие как Java. Значением по умолчанию является SQL. При использовании в операторе *ALTER* эта фраза меняет параметр *LANGUAGE* на новое значение.

PARAMETER STYLE {SQL | GENERAL}

Для внешних процедур определяется, нужно или не нужно явно передавать некоторые автоматические параметры. (Отличие стилей *SQL* и *GENERAL* заключается в том, что в первом стиле такие параметры, как индикаторы, передаются автоматически, а в стиле *GENERAL* параметры автоматически не передаются.) По умолчанию используется значение *PARAMETER STYLE SQL*. При использовании в операторе *ALTER* эта фраза меняет стиль параметров на новое значение.

SPECIFIC специальное_имя

Уникально идентифицирует функцию. Обычно используется с пользовательскими типами данных.

DETERMINISTIC | NOT DETERMINISTIC

Определяет природу значения, возвращаемого функцией. (Эта фраза используется только в операторах *CREATE/ALTER FUNCTION*.) Детерминированные функции (*DETERMINISTIC*) всегда возвращают одинаковые значения при одинаковых значениях входных параметров. Недетерминированные функции (*NOT DETERMINISTINC*) при одних и тех же значениях параметров могут возвращать разные результаты. Например, функция *CURRENT_TIME* является недетерминированной, так как возвращает постоянно увеличивающееся значение текущего времени.

NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA

Используется совместно с опцией *LANGUAGE* для определения типа SQL-операторов, содержащихся в функции. При использовании в операторе *ALTER* меняет текущее значение на новое.

NO SQL

Указывает, что в теле функции нет никаких операторов SQL. Используется для функций, написанных на не-SQL языках, например: *LANGUAGE ADA...CONTAINS NO SQL*.

CONTAINS SQL

Указывает, что в функции используются операторы SQL, которые не читают и не изменяют данные. Это значение по умолчанию.

READS SQL DATA

Указывает, что в функции используются операторы *SELECT* или *FETCH*.

MODIFIES SQL DATA

Указывает, что в функции используются операторы *INSERT*, *DELETE* или *UPDATE*.

RETURN NULL ON NULL INPUT | CALL ON NULL INPUT

Эти опции используются при написании функций на языках, не поддерживающих значения NULL. Если вы выбрали параметр *RETURN NULL ON NULL INPUT*, то функцию немедленно возвращает NULL, если значение NULL подается на вход. Если же используется параметр *CALL ON NULL INPUT*, то функция обрабатывает NULL в соответствии со стандартными правилами: к примеру, при сравнении двух значений NULL возвращается *UNKNOWN*. (Эта фраза используется в операторах *CREATE/ALTER FUNCTION/PROCEDURE*.) При использовании фразы в операторе *ALTER* можно поменять текущий стиль работы с NULL на другой.

DYNAMIC RESULT SETS *целое_число*

Определяет число курсоров, которые могут быть открыты в хранимой процедуре и останутся видимыми после выхода из процедуры. Значение по умолчанию равно 0. (Эта фраза не используется в операторах *CREATE FUNCTION*.) При использовании в операторе *ALTER* текущее значение *DYNAMIC RESULT SETS* меняется на новое.

STATIC DISPATCH

Возвращает статичные значения пользовательских типов данных и типов *ARRAY*. Требуется для не-SQL функций, использующих параметры пользовательских типов или типа *ARRAY*. (Эта фраза не используется в операторе *CREATE PROCEDURE*.) Фраза должна быть последней перед *блоком_кода*.

блок_кода

Определяет процедурные операторы для выполнения работы внутри процедуры или функции. Это самая важная и обычно самая большая часть процедуры или функции. Мы предполагаем, что вас в первую очередь интересуют функции, написанные на SQL, поэтому имейте в виду, что *блок_кода* не может содержать операторы управления схемой (SQL-Schema), операторы управления транзакциями (SQL-Transaction) и операторы управления подключениями (SQL-Connection).

Хотя мы и предполагаем, что вас интересуют функции и процедуры, написанные на SQL (ведь, в конце концов, эта книга об SQL), вы можете указать, что используется внешний код функции или процедуры. Для внешнего *блока_кода* используется следующий синтаксис:

```
EXTERNAL [NAME external_routine_name] [PARAMETER STYLE
  {SQL | GENERAL}] [TRANSFORM GROUP group_name]
```

где:

EXTERNAL [NAME имя_внешней_программы]

Определяет внешнюю программу и задает для нее имя. Если фраза опущена, то используется неименованная программа.

PARAMETER STYLE {SQL | GENERAL}

То же самое, что и для *CREATE PROCEDURE*.

TRANSFORM GROUP имя_группы

Трансформирует значения между пользовательскими типами данных и локальными переменными хранимых процедур и функций. Если фраза опущена, то по умолчанию используется *TRANSFORM GROUP DEFAULT*.

NAME новое_имя_объекта

Устанавливает для ранее созданной хранимой процедуры или функции новое имя. Используется только в операторах *ALTER FUNCTION/PROCEDURE*.

CASCADE | RESTRICT

Позволяет каскадно распространить (*CASCADE*) выполняемые изменения хранимой процедуры или функции на все зависимые процедуры и функции либо запретить такое распространение (*RESTRICT*). Мы категорически не рекомендуем выполнять оператор *ALTER* для изменения хранимых процедур и функций, имеющих зависимые объекты. Эта фраза используется только в операторах *ALTER FUNCTION* и *ALTER PROCEDURE*.

Общие правила

При создании пользовательской функции вы определяете входные параметры и одно возвращаемое значение. Затем вы можете вызывать пользовательскую функцию так же, как и любую другую функцию: например, в операторах *SELECT*, *INSERT* или во фразе *WHERE* оператора *DELETE*.

Пользовательские функции и хранимые процедуры имеют общее название – *хранимые подпрограммы*.

При создании хранимой процедуры вы определяете входные параметры, передаваемые в процедуру, и выходные параметры, возвращаемые из процедуры. Для вызова хранимой процедуры используется оператор *CALL*.

Содержимое *блока_кода* должно соответствовать правилам того языка, который поддерживается СУБД. В некоторых платформах отсутствуют встроенные процедурные языки, поэтому приходится использовать конструкцию *EXTERNAL*.

Например, вы могли бы создать пользовательскую функцию для SQL Server, возвращающую имя и фамилию человека, в одной строке:

```
CREATE FUNCTION formatted_name (@fname VARCHAR(30),
@lname VARCHAR(30) )
RETURNS VARCHAR(60)
AS
BEGIN
```

```
  DECLARE @full_name VARCHAR(60)
  SET @full_name = @fname + ' ' + @lname
  RETURN @full_name
END
```

Затем вы можете использовать эту функцию так же, как и любую другую:

```
SELECT formatted_name(au_fname, au_lname) AS name,
  au_id AS id
FROM authors
```

Оператор *SELECT* в этом примере вернет результат из двух столбцов – **name** и **id**.

Оператор *CREATE METHOD* из стандарта ANSI на текущий момент поддерживается только на платформе IBM UDB DB2. Метод – это просто пользовательская функция, имеющая специальное назначение. Для методов используется тот же синтаксис, что описан в этом разделе.

Хранимые процедуры создаются похожим образом. Процедура для SQL Server из следующего примера создает уникальное значение длиной 22 цифры (основанное на системной дате и времени) и возвращает это значение вызывающему процессу:

```
-- Хранимая процедура для Microsoft SQL Server
CREATE PROCEDURE get_next_nbr
   @next_nbr CHAR(22) OUTPUT
AS
BEGIN
  DECLARE @random_nbr INT
  SELECT @random_nbr = RAND( ) * 1000000
SELECT @next_nbr = RIGHT('000000'
  +CAST(ROUND(RAND(@random_nbr)*1000000,0)) AS CHAR(6), 6)
  +RIGHT('0000'+CAST(DATEPART(yy, GETDATE()) AS CHAR(4)),2)
  +RIGHT('000' +CAST(DATEPART(dy, GETDATE()) AS CHAR(3)),3)
  +RIGHT('00'  +CAST(DATEPART(hh, GETDATE()) AS CHAR(2)),2)
  +RIGHT('00'  +CAST(DATEPART(mi, GETDATE()) AS CHAR(2)),2)
  +RIGHT('00'  +CAST(DATEPART(ss, GETDATE()) AS CHAR(2)),2)
  +RIGHT('000' +CAST(DATEPART(ms, GETDATE()) AS CHAR(3)),3)
END
GO
```

В следующем (и последнем) примере ANSI SQL3 мы меняем имя существующей хранимой процедуры:

```
ALTER PROCEDURE get_next_nbr
NAME 'get_next_ID'
RESTRICT;
```

Советы и хитрости

Ключевым преимуществом хранимых процедур и функций является тот факт, что они *скомпилированы*, что означает, что после создания планы запросов сохраняются в базе данных. Скомпилированные процедуры и функции могут со-

храняться (хотя и не всегда) в кэше базы данных, что также дает прирост производительности. Процедуры и функции позволяют выполнить набор операторов за одно обращение к серверу базы данных, что снижает сетевой трафик.

Реализация пользовательских функций и хранимых процедур может сильно отличаться в разных СУБД. В некоторых платформах не поддерживаются внутренние *блоки_кода*; в этих платформах вы можете написать только внешний *блок_кода*. В следующих разделах описываются отличия и возможности каждой платформы.

Если вы выполняете оператор *ALTER PROCEDURE* или *ALTER FUNCTION*, то некоторые объекты могут стать невалидными после изменения объектов, от которых они зависят. Будьте внимательны и проверяйте все зависимости при изменении процедур и функций.

MySQL

В MySQL поддерживаются как операторы *ALTER* и *CREATE FUNCTION*, так и операторы *ALTER* и *CREATE PROCEDURE*.

Для операторов *CREATE* используется следующий синтаксис:

```
CREATE
[DEFINER = {пользователь | CURRENT_USER}]
{ FUNCTION | PROCEDURE } [база_данных.]имя_объекта
  ( [{IN | OUT | INOUT}] [параметр[, ...] )
[RETURNS тип_данных]
[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
[COMMENT 'строка']
тело;
```

где:

CREATE {FUNCTION | PROCEDURE } [база_данных.]имя_объекта

Создает внешнюю функцию или процедуру с заданным именем длиной не больше 64 символов. Модуль хранится в таблице *proctable* в базе данных.

DEFINER

Определяет пользователя, являющегося владельцем процедуры или функции. По умолчанию владельцем становится текущий пользователь.

([{IN | OUT | INOUT}] параметр[, ...])

Определяет один или несколько параметров. Все параметры функции должны быть входящими (*IN*), а параметры хранимых процедур могут быть любого из трех типов. В процедуре при неуказанном типе параметра он считается входящим.

RETURNS тип_данных

Указывает любой из допустимых типов данных MySQL, используемый для возвращаемого значения. Применяется только в операторах *CREATE/ALTER FUNCTION*.

COMMENT 'строка'

Добавляет комментарий, который затем можно получить при помощи операторов *SHOW CREATE PROCEDURE* и *SHOW CREATE FUNCTION*.

тело

Содержит один или несколько SQL-операторов. При использовании нескольких операторов их необходимо заключать в скобки *BEGIN* и *END*. В теле также могут содержаться такие элементы, как циклы, переменные, условные операторы.

Процедуры и функции могут содержать DDL-операторы *CREATE*, *ALTER* и *DROP*. Процедуры, но не функции также могут содержать операторы управления транзакциями, такие как *COMMIT* или *ROLLBACK*. В функциях нельзя использовать любые операторы, которые явно или неявно завершают транзакции, равно как нельзя использовать запросы без обработки их результатов, как, например, оператор *SELECT* без фразы *INTO*. Ни процедуры, ни функции не могут содержать команду *LOAD DATA INFILE*.

Любая созданная в MySQL функция может вызываться так же, как встроенные функции, например *ABS( )* или *SOUNDEX( )*. Хранимые процедуры вызываются при помощи оператора *CALL*.

Реализация оператора *CREATE FUNCTION* в MySQL давно поддерживает создание пользовательских функций, использующих внешний код на C/C++ в операционных системах, в которых возможна динамическая загрузка. Программа C/C++ указывается в опции *shared_program_library_name*. Функция может быть сразу откомпилирована сервером MySQL и будет доступна постоянно либо может быть динамически вызываемой программой. В следующем примере создается пользовательская функция в базе данных, работающей на Unix-сервере:

```
CREATE FUNCTION find_radius RETURNS INT SONAME "radius.so";
```

Oracle

Oracle поддерживает операторы *ALTER* и *CREATE* как для процедур, так и для функций. (Вам может быть интересно познакомиться с пакетами, также используемыми для создания хранимых процедур и функций. Читайте документацию Oracle.) Синтаксис оператора *CREATE PROCEDURE* следующий:

```
CREATE [OR REPLACE] {FUNCTION|PROCEDURE}
  [схема.]имя_объекта
[(параметр1 [IN | OUT | IN OUT] [NOCOPY] тип_данных[, ...])]
RETURN тип_данных
[DETERMINISTIC] [AUTHID {CURRENT_USER | DEFINER}]
[IS | AS] [EXTERNAL]
[PARALLEL_ENABLE
  [( PARTITION имя_секции BY {ANY | {HASH | RANGE}
  (столбец [, ...])} ) [{ORDER|CLUSTER} BY (столбец[, ...])]]]
{ {PIPELINED | AGGREGATE} [USING [схема.]тип_реализации] | [PIPELINED] {IS | AS} }
{блок_кода | LANGUAGE {JAVA NAME имя_внешней_программы |
  C [NAME имя_внешней_программы]}
  LIBRARY имя_библиотеки [AGENT IN (аргумент[, ...])]
    [WITH CONTEXT]
  [PARAMETERS ( параметры[, ...] )]};
```

Операторы *ALTER FUNCTION/PROCEDURE* используются для перекомпиляции невалидных хранимых процедур и функций.

```
ALTER {FUNCTION | PROCEDURE} [схема.]имя_объекта
COMPILE [DEBUG] [параметр_компилятора = значение [...]] [REUSE SETTINGS]
```

Значения параметров следующие:

CREATE [OR REPLACE] {FUNCTION|PROCEDURE} [схема.]имя_объекта

Создает новую хранимую процедуру или функцию. Используйте *OR REPLACE* для того, чтобы заменить уже существующую процедуру или функцию без ее предварительного удаления и затем повторной выдачи всех прав на нее.

Некоторые фразы используются только с функциями, включая фразы *RETURN*, *DETERMINISTIC* и *USING*.

IN | OUT | IN OUT

Указывает, является ли параметр функции входным, выходным, или и тем и другим одновременно.

NOCOPY

Повышает скорость работы с большими *OUT* и *IN OUT* параметрами, например типов *VARRAY* или *RECORD*.

AUTHID {CURRENT_USER | DEFINER}

Указывает, что процедура или функция должна выполняться с правами текущего пользователя (*CURRENT_USER*) или создателя (*DEFINER*).

AS EXTERNAL

Альтернативный способ указать, что процедура написана на языке C. В Oracle предпочтительнее использовать синтаксис *AS LANGUAGE C* за исключением случаев, когда параметры имеют типы PL/SQL.

PARALLEL_ENABLE

Делает возможным параллельное выполнение функции на симметричном мультипроцессорном (SMP) сервере. Фраза используется только для пользовательских функций. (Не используйте переменные пакетов или сессий, так как они не будут корректно использоваться при параллельном выполнении.) Параллельное выполнение управляется следующими дополнительными фразами:

PARTITION имя_секции BY {ANY | {HASH | RANGE} (столбец [, ...])}

Определяет секционирование для функции с входными параметрами типа *REF CURSOR*. Может быть особенно полезно для табличных функций. *ANY* секционирует данные случайным образом. Вы можете также выбрать секционирование по диапазону (*RANGE*) или по хеш-значению (*HASH*) по определенному набору столбцов.

{ORDER | CLUSTER} BY (столбец [, ...])

Упорядочивает или кластеризует данные по определенному набору столбцов при параллельной обработке. *ORDER BY* локально сортирует данные на параллельном сервере. *CLUSTER BY* ограничивает набор строк на параллельном сервере по ключевым значениям в указанном наборе столбцов.

{PIPELINED | AGGREGATE} [USING [схема.]тип_реализации]

PIPELINED позволяет итеративно возвращать значения табличной функции вместо обычного результата в виде *VARRAY* или вложенной таблицы. Эта

фраза используется только для пользовательских функций. Фраза *PIPELINED USING тип_реализации* используется для внешних функций, написанных на языках C++ или Java. При помощи *AGGREGATE USING тип_реализации* определяется агрегатная функция (обрабатывающая множество строк и возвращающая одно значение).

IS | AS

IS и *AS* эквивалентны и используются перед началом *блока_кода*.

блок_кода

В Oracle код хранимых процедур или пользовательских функций может быть написан на PL/SQL. Можно также использовать фразу *LANGUAGE* для хранимых процедур, написанных на Java и C.

LANGUAGE {JAVA NAME имя_внешней_программы | C [NAME имя_внешней_программы] LIBRARY имя_библиотеки [AGENT IN (аргумент[, ...])] [WITH CONTEXT] [PARAMETERS (параметры[, ...])] }

Определяет внешнюю реализацию хранимой процедуры или функции на языках C или Java. Параметры и семантика определения зависят от конкретного языка.

ALTER {FUNCTION | PROCEDURE} [схема.]имя_объекта

Повторно компилирует невалидную процедуру или функцию. Для изменения параметров, объявления или определения существующей процедуры или функции используйте *CREATE OR REPLACE*.

COMPILE [DEBUG] [REUSE SETTINGS]

Повторно компилирует процедуру или функцию. Ключевое слово *COMPILE* является обязательным. (При работе в SQL*Plus ошибки компиляции можно увидеть командой *SHOW ERRORS*.) Процедура становится валидной, если в процессе компиляции не возникло ошибок. При компиляции можно использовать следующие необязательные опции:

DEBUG

Создает и сохраняет код, используемый отладчиком PL/SQL.

параметр_компилятора = значение [...]

Определяет значения параметров компилятора. Множество параметров включает в себя *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS* и *NLS_LENGTH_SEMANTICS*. За подробностями обращайтесь к документации Oracle по компилятору PL/SQL.

REUSE SETTINGS

Сохраняет текущие настройки компилятора и использует их для рекомпиляции. В обычном режиме Oracle сбрасывает настройки компилятора, и требуется их повторная установка.

В Oracle хранимые процедуры и пользовательские функции очень похожи по структуре и использованию. Основное отличие состоит в том, что процедура не возвращает вызывающему процессу результирующее значение, а функция возвращает.

В качестве примера рассмотрим функцию, на вход которой дается название строительного проекта, а результатом является полученная прибыль:

```
CREATE OR REPLACE FUNCTION project_revenue
  (project IN varchar2)
RETURN NUMBER
AS
  proj_rev NUMBER(10,2);
BEGIN
  SELECT SUM(DECODE(action,'COMPLETED',amount,0)) -
         SUM(DECODE(action,'STARTED',amount,0)) +
         SUM(DECODE(action,'PAYMENT',amount,0))
  INTO proj_rev
  FROM construction_actions
  WHERE project_name = project;
  RETURN (proj_rev);
END;
```

В этом примере функция в качестве параметра принимает название проекта. Затем вычисляется прибыль проекта путем вычитания из завершающего платежа начальных затрат и прибавления прочих платежей. Строка *RETURN (proj_rev);* возвращает полученный результат вызывающему процессу.

В Oracle пользовательские функции нельзя использовать в следующих ситуациях:

- В ограничениях *CHECK* и значениях *DEFAULT* операторов *CREATE/ALTER TABLE*.
- В операторах *SELECT*, *INSERT*, *UPDATE* и *DELETE* пользовательская функция не может прямо или косвенно (то есть, будучи вызванной из другой функции) нарушать следующие ограничения:
  - Иметь параметры типа OUT или INOUT. (Хотя такие параметры допустимы в непрямых вызовах.)
  - Управлять транзакцией при помощи операторов *COMMIT*, *ROLLBACK*, *SAVEPOINT* или выполнять операторы *CREATE*, *ALTER*, *DROP*, неявно завершающие транзакцию.
  - Выполнять операторы управления сессией (*SET ROLE*) или системой (специальный оператор в Oracle *ALTER SYSTEM*).
  - Писать в базу данных (будучи вызванной из оператора *SELECT* или параллельных операторов *INSERT*, *UPDATE* и *DELETE*).
  - Писать данные в таблицу, модифицированную оператором, из которого вызывается функция.

При рекомпиляции при помощи оператора *ALTER* хранимая процедура или функция становится валидной, если не обнаружено ошибок компиляции. В противном случае хранимая подпрограмма становится невалидной. Но более важно то, что невалидными становятся все процедуры и функции, зависящие от рекомпилированной хранимой подпрограммы, вне зависимости от наличия или отсутствия ошибок. Вы можете перекомпилировать все зависимые объекты самостоятельно или позволить базе данных перекомпилировать их в момент обращения к ним.

В следующем примере показан оператор для рекомпиляции функции *project_revenue*, сохраняющий отладочную информацию:

```
ALTER FUNCTION project_revenue COMPILE DEBUG;
```

PostgreSQL

PostgreSQL поддерживает операторы *CREATE* и *ALTER FUNCTION*, но не поддерживает *CREATE* и *ALTER PROCEDURE*. Так сделано потому, что для процедурной обработки можно использовать тот же мощный механизм функций. Для создания функций используется следующий синтаксис:

```
CREATE [OR REPLACE] FUNCTION имя_функции
  ( [ параметр [{IN | OUT | INOUT}][, ...] ] )
[RETURNS тип_данных]
AS {блок_кода | объектный_файд, символьный_файл}
  [ LANGUAGE {C | SQL | internal} |
    {IMMUTABLE | STABLE | VOLATILE} |
    {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT |
    STRICT} | [EXTERNAL] SECURITY INVOKER |
    [EXTERNAL] SECURITY DEFINER ]
[WITH {[ISCACHABLE][,] [ISSTRICT]}]
```

Синтаксис *ALTER FUNCTION* следующий:

```
ALTER FUNCTION имя_функции
( [параметр [{IN | OUT | INOUT}][, ...] ] )
[RESTRICT]
[RENAME TO новое_имя_функции]
[OWNER TO новый_владелец]
[SET SCHEMA новая_схема]
```

Значение параметров следующее:

CREATE [OR REPLACE] FUNCTION имя_функции

Создает новую функцию с заданным именем или изменяет существующую функцию. *OR REPLACE* не позволяет менять имя, входные параметры или тип возвращаемого результата существующей функции; для изменения любого из этих значений нужно удалить функцию и затем создать ее заново.

RETURNS тип_данных

Определяет тип данных результата, возвращаемого функцией. Не используется для процедур.

блок_кода | объектный_файл, символьный_файл

Определяет структуру функции. *Блок_кода* представляет собой (в зависимости от параметра LANGUAGE) вызов внутренней функции, путь и имя объектного файла, SQL-запрос или набор команд на процедурном языке. Также функция может быть определена через объектный и символьный файл для вызова функции, написанной на C.

LANGUAGE {C | SQL | internal}

Определяет вызов внешней программы или внутренней SQL-программы. Так как все опции, кроме SQL, подразумевают программы, написанные на других

языках, то они не входят в рамки этой книги. Для написания функций на SQL нужно использовать фразу *LANGUAGE SQL*.

Вы можете добавить в PostgreSQL новый язык, не установленный по умолчанию, используя оператор *CREATE LANGUAGE*.

IMMUTABLE | STABLE | VOLATILE

Описывает поведение функции. *IMMUTABLE* означает, что функция детерминирована, не использует и не модифицирует данные (то есть не использует операторы *SELECT*). *STABLE* означает, что функция детерминирована и может читать или модифицировать данные в таблицах (то есть могут использоваться операторы *SELECT*, возвращающие одинаковые результаты при одинаковых значениях входных параметров). Если же не используются описанные параметры, то функция имеет тип *VOLATILE*, что значит, что функция недетерминирована (может возвращать различный результат при одних и тех же значениях входных параметров).

CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT

STRICT является синонимом для *RETURNS NULL ON NULL INPUT*. Значения *CALLED ON NULL INPUT* и *RETURNS NULL ON NULL INPUT* являются частью стандарта ANSI и описаны в соответствующем разделе.

[EXTERNAL] SECURITY {INVOKER | DEFINER}

Ключевое слово *INVOKER* указывает, что функция должна выполняться с правами вызывающего ее пользователя, а *DEFINER* указывает, что функция должна выполняться с правами создателя. Ключевое слово *EXTERNAL* не несет дополнительного смысла и добавлено только для совместимости с ANSI.

[WITH {[ISCACHABLE][,] [ISSTRICT]}]

Указывает, что функция возвращает одинаковый результат при одних и тех же значениях входных параметров, что позволяет оптимизировать производительность PostgreSQL. *WITH ISCACHABLE* аналогично настройке *DETERMINISTIC* из ANSI, за исключением того, что позволяет заранее вычислить значение функции. *WITH ISSTRICT* аналогично настройке *RETURNS NULL ON NULL INPUT* из ANSI. Поведение по умолчанию совпадает с поведением при *CALLED ON NULL INPUT*. Обратите внимание, что при объявлении функции можно использовать оба ключевых слова одновременно.

[RESTRICT] [RENAME TO новое_имя_функции] [OWNER TO новый владелец] [SET SCHEMA новая схема]

Устанавливает для процедуры новое имя, владельца или схему. Использование слова *RESTRICT* необязательно.

В PostgreSQL возможна *перегрузка* функций, то есть можно использовать одно и то же имя для функций, принимающих на вход разные значения параметров.

При удалении существующей функции все зависимые объекты становятся невалидными и требуют рекомпиляции.

Вот пример простой функции в PostgreSQL:

```
CREATE FUNCTION max_project_nbr
RETURNS int4
AS "SELECT MAX(title_ID) FROM titles AS RESULT"
LANGUAGE 'sql';
```

В этом примере мы создаем пользовательскую функцию, возвращающую максимальное значение **title_ID** из таблицы **titles**.

В PostgreSQL оператор *CREATE FUNCTION* используется как замена для *CREATE PROCEDURE* и для *CREATE TRIGGER*.

SQL Server

SQL Server поддерживает операторы *CREATE* и *ALTER* как для процедур, так и для функций. По умолчанию хранимые процедуры могут возвращать результирующие наборы данных, а пользовательские функции могут возвращать наборы из одной или множества записей, используя возвращаемое значение типа *TABLE*, что не предусмотрено стандартом ANSI. Тем не менее, это делает хранимые процедуры и функции более гибкими и мощными. Для создания процедуры или функции используется следующий синтаксис:

```
CREATE {FUNCTION | PROCEDURE}
  [владелец.]имя_объекта[;целое_число]
( [ {@параметр тип_данных [VARYING] [=default] [OUTPUT]}
  [READONLY][, ...] ] )
[RETURNS {тип_данных | TABLE]]
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE |
  RECOMPILE, ENCRYPTION |
  RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT |
  EXEC[UTE] AS {CALLER | SELF | OWNER |'имя_пользователя'}}]
[FOR REPLICATION]
[AS]
  блок_кода
```

Для изменения процедуры или функции используется следующий синтаксис:

```
ALTER {FUNCTION | PROCEDURE}
  [владелец.]имя_объекта[;целое_число]
([{@параметр тип_данных[VARYING][=default][OUTPUT]}[, ...]])
[RETURNS { тип_данных | TABLE}]
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE |
  RECOMPILE, ENCRYPTION |
  RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT |
  EXEC[UTE] AS {CALLER | SELF | OWNER |'имя_пользователя'}}]
[FOR REPLICATION]
[AS]
  блок_кода
```

Значения параметров следующие:

CREATE {FUNCTION | PROCEDURE} [владелец.]имя_объекта [;целое_число]

Создает новую хранимую процедуру или пользовательскую функцию в текущей базе данных. Для хранимых процедур вы дополнительно можете указать номер версии в формате *имя_процедуры; 1*, где 1 – это целочисленный номер вер-

сии. Этот параметр позволяет иметь в базе данных несколько версий одной процедуры.

{@параметр тип_данных [VARYING] [=значение по умолчанию] [OUTPUT]} [READONLY][, ...]

Определяет один или несколько параметров. В SQL Server имена параметров всегда начинаются с символа @.

VARYING

Используется в хранимых процедурах с параметрами типа *CURSOR*. Означает, что набор строк создается динамически.

=значение по умолчанию

Устанавливает для параметра значение по умолчанию. Это значение используется в тех случаях, когда процедура или функция вызывается без явного указания значения для параметра.

OUTPUT

Используется в хранимых процедурах, функциональный эквивалент описанного в ANSI параметра *OUT* оператора *CREATE FUNCTION*. Значения возвращаемых параметров передаются в вызывающий процесс через переменные команды *EXEC[UTE]*. Выходные параметры могут относиться к любому типу данных за исключением *TEXT* и *IMAGE*.

READONLY

Используется в функциях для указания того, что параметр не модифицируется внутри функции. Это особенно полезно для пользовательских типов данных *TABLE*.

RETURNS {тип_данных | TABLE}

Позволяет функции возвращать одно значение указанного типа либо возвращать набор значений через тип данных *TABLE*. Функция называется *встроенной* (inline), если она определяется одним единственным оператором *SELECT* и для нее не приводится перечень столбцов. Если во фразе *RETURNS* используется тип *TABLE* с определенным перечнем столбцов и их типов, то функция называется *многооператорной табличной функцией* (*multi-statement table-valued function*).

WITH

Позволяет задать дополнительные характеристики пользовательской функции или хранимой процедуры.

ENCRYPTION

Шифрует значение в столбце системной таблицы SQL Server, в которой хранится текст процедуры или функции, защищая таким образом от нежелательного просмотра кода. Используется с процедурами и с функциями.

SCHEMABINDING

Указывает, что функция привязывается к определенному объекту базы данных, такому как таблица или представление. Этот объект базы данных не может быть изменен или удален, пока существует функция (и пока установлена опция *SCHEMABINDING*). Используется только с функциями.

RECOMPILE

Запрещает серверу кэшировать план хранимой процедуры. Вместо этого план выполнения создается каждый раз заново. Эта возможность очень полезна при использовании нетипичных или временных значений в процедуре, но может привести к серьезному падению производительности. Используется только в хранимых процедурах. Обратите внимание, что опции *RECOMPLE* и *ENCRYPTION* можно использовать одновременно.

EXEC[UTE] AS {CALLER|SELF|OWNER|'имя_пользователя'}

Необязательная опция, определяющая, с какими привилегиями выполняется процедура или функция. *CALLER* означает, что при выполнении используются привилегии вызывающего пользователя. *CALLER* является значением по умолчанию. *SELF* обозначает, что выполнение производится с привилегиями создателя. *OWNER* обозначает выполнение с правами владельца. Наконец, можно явно указать пользователя, с правами которого будет выполняться процедура или функция.

FOR REPLICATION

Отключает выполнение хранимой процедуры на сервере-подписчике. Эта фраза в основном используется для создания фильтрующей хранимой процедуры, используемой встроенным в SQL Server механизмом репликации. Эта опция несовместима с опцией *WITH RECOMPILE*.

Как и в случае с таблицами (смотрите *CREATE TABLE*), в SQL Server можно создавать локальные и глобальные временные хранимые процедуры, их названия должны, соответственно, начинаться с одного или двух символов #. Временные процедуры существуют только в рамках создавшей их сессии. Когда сессия завершается, все ее временные процедуры автоматически удаляются.

В SQL Server хранимая процедура или пользовательская функция может иметь до 1024 параметров, обозначаемых символами @. Параметры должны иметь стандартный тип данных SQL Server. (Параметры типа *CURSOR* должны быть объявлены с опциями *VARYING* и *OUTPUT*.) Пользователь или вызывающий процесс должны предоставить значения параметров. Для параметров можно использовать и значения по умолчанию. Значение по умолчанию должно быть константой или NULL и может содержать обобщающие символы (wildcard).

В SQL Server требуется, чтобы функция имела один или более параметров, задаваемых пользователем. Для параметров можно использовать любой тип данных, за исключением *TIMESTAMP*. Значение, возвращаемое функцией, может иметь любой тип данных, за исключением *TIMESTAMP*, *TEXT*, *NTEXT* и *IMAGE*. Для встроенных функций список столбцов возвращаемого типа *TABLE* указывать не нужно.

Операторы *ALTER FUNCTION* и *ALTER PROCEDURE* поддерживают полностью тот же синтаксис, что и соответствующие операторы *CREATE*. Таким образом, с помощью оператора *ALTER* можно изменить любой атрибут существующей процедуры или функции без влияния на зависимые объекты или привилегии.

Для встроенных функций *блок_кода* представляет собой один-единственный оператор *SELECT* в формате *RETURN(SELECT...)*, либо слово *AS* и следующий за

ним набор операторов Transact-SQL. При использовании *RETURN(SELECT)* слово *AS* необязательно. Вот еще некоторые правила для пользовательских функций в SQL Server:

- При использовании фразы *AS* *блок_кода* должен быть заключен в *BEGIN*... *END*.
- Функции не могут менять данные или вызывать любые побочные эффекты. Это приводит к некоторым ограничениям. Например, операторы *INSERT*, *UPDATE* и *DELETE* могут менять данные только в локальных переменных типа *TABLE*.
- Если функция возвращает скалярное значение, в ней должна содержаться фраза *RETURN* и возвращаемое значение должно иметь тот же тип, что указан во фразе *RETURNS* при создании функции.
- Последним оператором в *блоке_кода* должен быть безусловный *RETURN*, возвращающий скалярное значение или значение типа *TABLE*.
- В *блоке_кода* не могут использоваться глобальные переменные с постоянно меняющимися значениями, такие как *@@CONNECTIONS* или *GETDATE*. Тем не менее, можно использовать переменные, возвращающие одно неизменное значение, например *@@SERVERNAME*.

Далее приводится пример *скалярной* функции, возвращающей единственное значение. Созданную функцию можно использовать в запросах так же, как и любую системную функцию:

```
CREATE FUNCTION metric_volume
-- На вход даются размеры в сантиметрах.
  (@length decimal(4,1),
  @width decimal(4,1),
  @height decimal(4,1) )
RETURNS decimal(12,3) - Кубические сантиметры.
AS BEGIN
    RETURN ( @length * @width * @height )
  END
GO
SELECT project_name,
  metric_volume(construction_height,
    construction_length,
    construction_width)
FROM housing_construction
WHERE metric_volume(construction_height,
  construction_length,
  construction_width) >= 300000
GO
```

Встроенные табличные функции возвращают результат с помощью одного оператора *SELECT*, используя фразу *AS RETURN*. Например, мы можем по идентификатору магазина получить список всех изданий:

```
CREATE FUNCTION stores_titles(@stor_id varchar(30))
RETURNS TABLE
AS
RETURN (SELECT title, qty
```

```
FROM sales AS s
JOIN titles AS t ON t.title_id = s.title_id
WHERE s.stor_id = @storeid )
```

Теперь давайте изменим тип параметра функции и добавим условие в *WHERE* (изменения выделены жирным):

```
ALTER FUNCTION stores_titles(@stor_id VARCHAR(4))
RETURNS TABLE
AS
RETURN (SELECT title, qty
FROM sales AS s
JOIN titles AS t ON t.title_id = s.title_id
WHERE s.stor_id = @storeid
AND s.city = 'New York')
```

Пользовательские функции, возвращающие набор строк через тип данных *TABLE*, часто используются во фразе *FROM* оператора *SELECT* точно так же, как и обычные таблицы. Многооператорные табличные функции могут быть достаточно замысловатыми, так как состоят из набора операторов, заполняющих возвращаемую переменную типа *TABLE*.

Вот пример использования табличной функции во фразе *FROM*. Обратите внимание, что вызываемой функции присваивается псевдоним, как и обычной таблице:

```
SELECT co.order_id, co.order_price
FROM   construction_orders AS co,
       fn_construction_projects('Cancelled') AS fcp
WHERE  co.construction_id = fcp.construction_id
ORDER BY co.order_id
GO
```

В хранимых процедурах *блок_кода* может состоять из одного или нескольких операторов Transact-SQL, заключенных в *BEGIN*...*END* и занимающих объем до 128 Мбайт. Вот некоторые правила для хранимых процедур:

- В хранимых процедурах можно использовать большинство операторов Transact-SQL за исключением операторов *SET SHOWPLAN_TEXT* и *SET SHOWPLAN_ALL*.
- Некоторые операторы при использовании из хранимых процедур имеют ограниченную функциональность. Это такие операторы, как *ALTER TABLE*, *CREATE INDEX*, *CREATE TABLE*, все операторы *DBCC*, *DROP TABLE*, *DROP INDEX*, *TRUNCATE TABLE* и *UPDATE STATISTICS*.
- В SQL Server используется отложенное разрешение имен, то есть хранимая процедура может быть откомпилирована, даже если в ней есть ссылки на еще не существующие объекты. SQL Server создает план выполнения, и процедура завершается с ошибкой, только если в момент вызова объект все еще не существует.
- Допускаются вложенные вызовы хранимых процедур. Если одна хранимая процедура вызывает другую, то значение системной переменной *@@NESTLEVEL* увеличивается на 1. Значение уменьшается на 1 при завершении вызванной процедуры. Для получения текущей глубины вложенности можно из хранимой процедуры использовать команду *SELECT @@NESTLEVEL*.

В следующем примере процедура для SQL Server создает уникальное значение длиной 22 цифры (основанное на системной дате и времени) и возвращает это значение вызывающему процессу:

```
-- Хранимая процедура для Microsoft SQL Server
CREATE PROCEDURE get_next_nbr
  @next_nbr CHAR(22) OUTPUT
AS
BEGIN
  DECLARE @random_nbr INT
  SELECT @random_nbr = RAND( ) * 1000000
  SELECT @next_nbr =
    RIGHT('000000'+CAST(ROUND(RAND(@random_nbr)*1000000,0))
      AS CHAR(6),6) +
    RIGHT('0000' + CAST(DATEPART (yy, GETDATE( ) )
      AS CHAR(4)), 2) +
    RIGHT('000' + CAST(DATEPART (dy, GETDATE( ) )
      AS CHAR(3)), 3) +
    RIGHT('00' + CAST(DATEPART (hh, GETDATE( ) )
      AS CHAR(2)), 2) +
    RIGHT('00' + CAST(DATEPART (mi, GETDATE( ) )
      AS CHAR(2)), 2) +
    RIGHT('00' + CAST(DATEPART (ss, GETDATE( ) )
      AS CHAR(2)), 2) +
    RIGHT('000' + CAST(DATEPART (ms, GETDATE( ) )
      AS CHAR(3)), 3)
END
GO
```

В SQL Server поддерживаются процедуры и функции, написанные на языках, исполняемых в CLR (Common Language Runtime) Microsoft .NET Framework. Синтаксис объявления таких процедур и функций похож на синтаксис, используемый для обычных процедур и функций, но вместо кода используются внешние сборки. Если вы хотите узнать о программировании для SQL Server с использованием CLR, то обратитесь к документации.

См. также

CALL
RETURN

CREATE/ALTER INDEX

Индексы – это специальные объекты, созданные поверх таблиц, позволяющие значительно ускорить такие операции манипуляции данными, как *SELECT*, *UPDATE* и *DELETE*. Селективность фразы *WHERE* и набор планов выполнения, доступный оптимизатору, обычно зависят от качества индексов, построенных над конкретной таблицей базы данных.

Оператор CREATE INDEX не является частью стандарта ANSI, и поэтому реализация оператора значительно отличается на разных платформах.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Общий синтаксис

```
CREATE [UNIQUE] INDEX имя_индекса ON имя_таблицы
(имя_столбца[, ...])
```

Ключевые слова

CREATE [UNIQUE] INDEX *имя_индекса*

Создает новый индекс с заданным именем в текущей базе данных и схеме. Так как индексы связаны с конкретными таблицами (или иногда с представлениями), то имя индекса должно быть уникально в пределах таблицы, по которой он построен. Ключевое слово UNIQUE указывает, что индекс является уникальным ограничением таблицы и запрещает повторяющиеся значения в столбце или наборе столбцов, на которых он построен.

имя_таблицы

Указывает существующую таблицу, по которой строится индекс. Индекс зависит от таблицы: если удаляется таблица, то автоматически удаляется и индекс.

имя_столбца[, ...]

Определяет список индексируемых столбцов таблицы. Указатели на данные, построенные по индексируемым столбцам, позволяют оптимизатору запросов значительно увеличить скорость выполнения таких операций, как *SELECT* и *DELETE*. Большинство баз данных поддерживают *составные индексы*, также называемые *конкатенированными индексами*, построенные по двум или более столбцам, которые в поиске обычно используются вместе (например, фамилия и имя).

Общие правила

Индексы строятся над таблицами для ускорения операций, использующих эти таблицы, в частности, при использовании фраз *WHERE* и *JOIN*. Индексы могут также ускорять некоторые другие операции:

- Нахождение минимального или максимального значения в индексированном столбце.
- Сортировку или группировку столбца таблицы.
- Поиск по условиям *IS NULL* или *IS NOT NULL*.
- Быструю выборку данных из индексированных столбцов в случаях, когда все необходимые для выполнения запроса данные находятся в индексе. Оператор *SELECT*, извлекающий данные только из столбцов индекса и не затрагивающий другие столбцы таблицы, называется *покрывающим запросом*. Соответствующий индекс называется *покрывающим индексом*.

После создания таблицы по некоторым ее столбцам можно создать индексы. Хорошей практикой является создание индексов по столбцам, наиболее часто используемым во фразах *WHERE* и *JOIN*. Например, следующий оператор создает индекс по столбцу таблицы **sales**, часто используемому во фразе *WHERE*:

```
CREATE INDEX ndx_ord_date ON sales(ord_date);
```

В следующем примере мы создаем уникальный индекс по столбцам **pub_name** и **country** таблицы **publishers**:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country);
```

Так как индекс является уникальным, каждая запись в таблице должна иметь уникальную комбинацию значений имени издателя и страны.

В некоторых СУБД индексы можно создавать не только по таблицам, но и по представлениям.

Советы и хитрости

Составные индексы наиболее полезны, если запросы чаще всего запрашивают столбцы из левой части списка столбцов, по которым построен индекс. Если в запросе не используются столбцы из левой части индекса, то запрос может не отработать лучшим образом. Предположим, что у нас имеется индекс по столбцам с фамилией и именем. Если в запросе мы используем только имя, то индекс, начинающийся с фамилии, вряд ли окажется нам полезным. Хотя механизмы запросов в некоторых СУБД уже усовершенствованы таким образом, что подобная ситуация не является проблемой.

При создании индекса таблица временно может увеличиться на 20–50 процентов от текущего размера. Убедитесь, что у вас есть достаточно свободного пространства. Большая часть этого пространства освободится после того, как индекс будет создан.

Следует знать, что бывают ситуации, когда слишком большое количество индексов только ухудшит производительность системы. В целом индексы ускоряют операции поиска в таблицах, например оператор *SELECT*. Однако каждый индекс – это дополнительные накладные расходы при выполнении операций *UPDATE*, *DELETE* и *INSERT*, потому что при изменении таблицы следует обновить соответствующие индексы. Как правило, по одной таблице требуется не более 6–12 индексов.

Также индексы требуют дополнительного места для хранения. Чем больше столбцов в индексе, тем больше места он требует. Обычно это не представляет большой проблемы, но иногда застает врасплох новичков.

В большинстве СУБД индексы используются для создания статистических выборок (называемых просто *статистиками*), позволяющих оптимизатору запросов определять, какой индекс или комбинация индексов лучше всего подходят для выполнения запроса. Эти статистики после создания индекса всегда актуальны, но могут устаревать по мере вставки и удаления записей в таблице. По-

этому и индексы по мере старения могут быть уже не столь полезными. Вам следует внимательно следить за поддержкой статистик в актуальном состоянии.

MySQL

MySQL поддерживает оператор *CREATE INDEX*, но не поддерживает *ALTER INDEX*. В зависимости от используемого механизма хранения данных (engine) вы можете создавать разные типы индексов, не обязательно это будет индекс, хранящийся в виде B-дерева в файле. У строк, используемых в индексах, автоматически убираются пробелы в начале и конце. Синтаксис оператора *CREATE INDEX* следующий:

```
CREATE [ONLINE|OFFLINE] [UNIQUE|FULLTEXT] INDEX имя_индекса
[USING {BTREE | HASH | RTREE}]
ON таблица(столбец(длина)[, ...])
[KEY BLOCK SIZE целое_число]
[WITH PARSER имя_парсера]
```

где:

FULLTEXT

Создает по столбцу индекс для полнотекстового поиска. Этот тип индекса поддерживается только механизмом MyISAM для типов данных *CHAR*, *VARCHAR* и *TEXT*. Опция *(длина)* при этом не используется.

ONLINE|OFFLINE

ONLINE означает, что для создания индекса не требуется предварительно копировать таблицу. *OFFLINE* копирует таблицу перед созданием индекса.

USING {BTREE|HASH|RTREE}

Определяет тип создаваемого индекса. Индексы типа *RTREE* используются только для пространственных (*SPATIAL) индексов*. Используйте эту опцию аккуратно, так как в разных механизмах хранения используются разные типы индексов: в MyISAM поддерживаются *BTREE* и *RTREE*, InnoDB поддерживает только *BTREE*, NDB поддерживает только *HASH* (а фразу *USING* только для уникальных и первичных ключей), MEMORY/HEAP поддерживает *HASH* и *BTREE*. Этот параметр замещает использовавшийся в версиях MySQL до 5.1.10 устаревший параметр *TYPE*.

KEY BLOCK SIZE целое_число

Рекомендует СУБД использовать указанный размер блока индекса. Значение задается в килобайтах. Значение 0 применяется, когда следует использовать размер блока по умолчанию для данного механизма хранения.

WITH PARSER имя_парсера

Используется только для индексов типа *FULLTEXT* для указания модуля парсера, используемого с индексом. Модули подробно описаны в документации MySQL.

MySQL поддерживает общепринятый синтаксис оператора *CREATE INDEX*. Интересной особенностью является возможность построить индекс по нескольким первым символам столбца типа *CHAR* или *VARCHAR* (для столбцов типа *BLOB* и *TEXT* указание количества индексируемых символов обязательно). Это возможность очень полезна, когда первые несколько символов значения уже

обеспечивают хорошую селективность и при этом важно сэкономить пространство для хранения индекса. В следующем примере мы создаем индекс по первым 25 символам столбца **pub_name** и 10 символам столбца **country**:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name(25),
  country(10))
```

Все механизмы хранения MySQL поддерживают не менее 16 индексов по одной таблице с общей длиной ключа не менее 256 байт. Однако конкретные значения будут зависеть от используемого механизма хранения.

Oracle

В Oracle оператор *CREATE INDEX* используется для создания индексов по таблицам, секционированным таблицам, кластерам, индексно-организованным таблицам, скалярным атрибутам объектных типов, используемых в таблицах, и столбцам типа «вложенная таблица». В Oracle используются разные типы индексов, включая обычные индексы в виде B-деревьев, *BITMAP*-индексы (индексы на основе битовых карт, полезные в ситуациях, когда каждое значение столбца повторяется 100 и более раз), секционированные индексы, индексы на базе функций (построенные не по значению столбца, а по выражению) и предметные (domain) индексы.

В Oracle имя индекса должно быть уникально в пределах схемы, а не в пределах таблицы, по которой индекс создан.

Oracle также поддерживает оператор *ALTER INDEX*, используемый для изменения или перестроения существующего индекса без его предварительного удаления. Синтаксис оператора *CREATE INDEX* следующий:

```
CREATE [UNIQUE | BITMAP] INDEX имя_индекса
{ON
  {имя_таблицы ({столбец | выражение} [ASC | DESC][, ...])
    [{INDEXTYPE IS индексируемый_тип
    [PARALLEL [целое_число] | NOPARALLEL]
    [PARAMETERS ('параметры')] }] |
  CLUSTER имя_кластера |
  FROM имя_таблицы WHERE условие [LOCAL секционирование]}
[ {LOCAL секционирование | GLOBAL секционирование } ]
[физические_атрибуты] [{LOGGING | NOLOGGING}] [ONLINE]
[COMPUTE STATISTICS]
[{TABLESPACE табличное_пространство | DEFAULT}]
[{COMPRESS целое_число | NOCOMPRESS}]
[{NOSORT | SORT}] [REVERSE] [{VISIBLE | INVISIBLE}]
[{PARALLEL [целое_число] | NOPARALLEL}] }
```

Синтаксис для *ALTER INDEX* следующий:

```
ALTER INDEX имя_индекса
{ {ENABLE | DISABLE} | UNUSABLE | {VISIBLE | INVISIBLE} |
  RENAME TO новое_имя_индекса | COALESCE |
  [NO]MONITORING USAGE | UPDATE BLOCK REFERENCES |
  PARAMETERS ('параметры_ODCI') | изменение_секционирования
```

```
| перестроение |
  [DEALLOCATE UNUSED [KEEP целое_число [K | M | G | T]]]
  [ALLOCATE EXTENT ( [SIZE целое_число [K | M | G | T]]
    [DATAFILE 'имя_файла'] [INSTANCE целое_число] )]
  [SHRINK SPACE [COMPACT] [CASCADE]]
  [{PARALLEL [целое_число] | NOPARALLEL}]
  [{LOGGING | NOLOGGING}]
  [физические_атрибуты] }
```

Значения параметров следующие:

BITMAP

Вместо индексирования каждой отдельной записи таблицы создается битовая карта для каждого индексируемого значения. Битовые карты лучше использовать для таблиц, где данные редко изменяются несколькими параллельно работающими сеансами (например, для таблиц, которые в основном читаются). *BITMAP*-индексы несовместимы с глобальными секционированными индексами, фразой *INDEXTYPE* и «индекс»-таблицами (index-organized tables) без таблицы соответствия логических и физических идентификаторов строк.

ASC | DESC

Определяет порядок хранения значений в индексе по возрастанию (*ASC*) или по убыванию (*DESC*). *ASC* используется как значение по умолчанию. Имейте в виду, что в Oracle индексы *DESC* используются как индексы на базе функций, поэтому есть определенная разница в работе *ASC* и *DESC* индексов. При использовании *INDEXTYPE* указание *ASC* и *DESC* запрещено. *DESC* игнорируется для индексов на основе битовых карт.

INDEXTYPE IS индексируемый_тип [PARAMETERS ('параметры')]

Строит индекс по пользовательскому типу данных. Для создания предметного индекса требуется, чтобы пользовательский тип уже существовал. Если пользовательский тип требует параметров, то передавайте их с помощью фразы *PARAMETERS*. Также вы можете распараллелить построение индекса при помощи фразы *PARALLEL* (рассматривается далее).

CLUSTER имя_кластера

Объявляет кластерный индекс на основе указанного кластера. В Oracle кластерный индекс физически хранит рядом две таблицы, обращение к которым в запросах идет по одинаковым столбцам, обычно по первичному и внешнему ключу. (Кластеры создаются при помощи специального оператора *CREATE CLUSTER*.) При создании кластерного индекса не требуется указание таблицы или индексируемых столбцов, потому что они указываются в операторе *CREATE CLUSTER*.

GLOBAL секционирование

Полный синтаксис следующий:

```
GLOBAL PARTITION BY
  {RANGE (список_столбцов) ( PARTITION [имя_секции]
    VALUE LESS THAN (список_значений)
    [физические_атрибуты]
    [TABLESPACE табличное_пространство]
```

```
  [LOGGING | NOLOGGING][, ...] )} |
{HASH (список_столбцов) ( PARTITION [имя_секции] )
  {[TABLESPACE табличное_пространство]
  [[OVERFLOW] TABLESPACE табличное_пространство]
    [VARRAY столбец_VARRAY STORE AS LOB имя_сегмента]
    [LOB (столбец_LOB) STORE AS [имя_сегмента]]
  [TABLESPACE табличное_пространство]} |
  [STORE IN (табличное_пространство [, ...])]
  [OVERFLOW STORE IN (табличное_пространство ,...])]}
  [, ...]
```

Фраза *GLOBAL PARTITION* позволяет вручную секционировать индекс по диапазонам значений или по хеш-значениям. (По умолчанию индекс следует секционировать так же, как секционирована таблица, по которой этот индекс построен.) Вы можете указать в списке до 32 столбцов, столбец *ROWID* использовать нельзя. Также вы можете использовать фразы *[NO]LOGGING* и *TABLESPACE* и определить физические атрибуты хранения. В списке через запятую вы можете указать несколько секций:

RANGE

Секционирует индекс по диапазону значений в заданном списке столбцов.

VALUES LESS THAN (список_значений)

Устанавливает верхнюю границу значений для секции. Значения в *списке_значений* соответствуют столбцам из *списка_столбцов*. Оба списка – префикс-зависимы, что в данном случае означает следующее: если индекс состоит из столбцов **a**, **b**, **c**, то вы можете секционировать его по набору столбцов **(a, b)** или **(a, b, c)**, но не по **(b,c)**. Последнее значение в списке должно быть ключевым словом *MAXVALUE*.

HASH

Секционирует индекс по хеш-значениям. Каждая строка индекса помещается в ту или иную секцию в зависимости от значения хеш-функции для заданного набора столбцов.

Вы можете указать специальное табличное пространство для хранения объектов типа *LOB* и *VARRAY*, а также табличное пространство переполнения (*OVERFLOW*).

LOCAL секционирование

Используется для локального секционирования индексов по диапазону значений, списку значений, хеш-значениям или для составного (composite) секционирования. Вы можете указать через запятую несколько секций с соответствующими атрибутами. Если не использовать эту фразу, то Oracle создает секции индекса в соответствии с секциями таблицы. Секционирование индексов делается одним из трех следующих способов:

Секционирование по диапазону или списку значений

Применяется с обычными или эквисекционированными таблицами. Индексы, секционированные по диапазону или списку значений (синтаксис одинаковый), требуют следующего синтаксиса:

```
LOCAL [ (PARTITION [имя_секции]
  { [физические_атрибуты]
```

```
[TABLESPACE табличное_пространство]
[LOGGING | NOLOGGING] |
[COMPRESS | NOCOMPRESS] }[, ...]) ]
```

Значение всех параметров то же самое, что и для *GLOBAL PARTITION* (смотрите ранее), за исключением того, что индекс локальный.

Секционирование по хеш-значениям

Применяются с таблицами, секционированными по хеш-значениям. В этом случае вы можете использовать синтаксис, приведенный ранее, либо для хранения секций индекса в определенных табличных пространствах использовать следующий синтаксис:

```
LOCAL {STORE IN (табличное_пространство [, ...]) |
  (PARTITION [имя_секции]
  [TABLESPACE табличное_пространство])}
```

Если вы укажете больше табличных пространств, чем секций индекса, то Oracle при секционировании будет циклически переходить между табличными пространствами.

Составное секционирование индексов

Применяется с таблицами с составным (composite) секционированием. Используется следующий синтаксис:

```
LOCAL [STORE IN (табличное_пространство [, ...])]
PARTITION [имя_секции]
  {[ физические_атрибуты]
   [TABLESPACE табличное_пространство]
   [LOGGING | NOLOGGING] |
   [COMPRESS | NOCOMPRESS]}
     [ {STORE IN (табличное_пространство [, ...]) |
     (SUBPARTITION [имя_подсекции]
       [TABLESPACE табличное_пространство])} ]
```

Вы можете использовать фразу *LOCAL STORE*, используемую при секционировании по хеш-значениям, или фразу *LOCAL*, используемую при секционировании по диапазону или списку значений. (При использовании фразы *LOCAL* замените ключевое слово *SUBPARTITION* на *PARTITION*.)

физические_атрибуты

Устанавливает значения для параметров *PCTFREE*, *PCTUSED* и *INITRANS*. По умолчанию используются значения *PCTFREE 10*, *PCTUSED 40*, *INITRANS 2*.

PCTFREE целое_число

Устанавливает процент свободного пространства, оставляемого в каждом блоке при создании индекса. Это ускоряет добавление и изменение записей в таблице. Однако параметр *PCTFREE* учитывается только при создании индекса. Поэтому свободное пространство может уменьшаться по мере вставки, обновления и удаления записей.

PCTUSED целое_число

Говорит о том, когда блок следует помещать в список свободных блоков. Если использованное пространство блока становится меньше указанного

значения, то блок становится доступным для вставки данных. Сумма значений *PCTFREE* и *PCTUSED* должна быть меньше или равна 100. Эта фраза не используется с индексно-организованными таблицами.

INITRANS *целое_число*

Устанавливает для блока данных начальное значение допустимых параллельно работающих транзакций. Значение должно быть в пределах от 1 до 255.

В версиях Oracle до 11g использовался параметр *MAXTRANS* для ограничения максимального числа транзакций для блока данных, но сейчас этот параметр устарел. Oracle 11g автоматически устанавливает для *MAXTRANS* значение 255, вне зависимости от того, какое значение вы указали явно (хотя существующие объекты сохраняют установленное значение *MAXTRANS*).

LOGGING | NOLOGGING

Включает (*LOGGING*) или выключает (*NOLOGGING*) режим журналирования при создании индекса. Эта фраза также определяет поведение по умолчанию при массовых вставках с использованием утилиты SQL*Loader. В случае секционированных индексов эта фраза устанавливает значение по умолчанию для всех секций и соответствующих сегментов, а также для всех секций и подсекций, добавляемых позднее оператором *ALTER TABLE...ADD PARTITION*. (При работе в режиме *NOLOGGING* мы рекомендуем делать полную резервную копию после построения индексов, чтобы не пришлось перестраивать индексы в случае сбоев.)

ONLINE

Разрешает манипуляции с данными таблицы в процессе создания индекса. Даже в режиме *ONLINE* в конце процесса создания индекса таблица будет на очень короткое время заблокирована. Все изменения в таблице, выполненные в момент построения индекса, будут отражены в созданном индексе. Режим *ONLINE* несовместим с битовыми и кластерными индексами, а также с параллельным созданием индекса. Также его нельзя использовать с индексами по столбцам *UROWID* и индексно-организованными таблицами с первичным ключом из более чем 32 полей.

COMPUTE [STATISTICS]

Собирает статистику в момент создания индекса, когда это можно сделать с минимальными затратами. Иначе статистику придется собирать после создания индекса.

TABLESPACE {табличное_пространство | DEFAULT}

Определяет табличное пространство, в котором хранится индекс. Если табличное пространство не указано или явно выбрана опция *DEFAULT*, то индекс хранится в табличном пространстве, используемом по умолчанию. (Локальные секционированные индексы, для которых указано *TABLESPACE DEFAULT*, хранятся в тех же табличных пространствах, в которых хранятся соответствующие секции таблицы.)

COMPRESS [целое_число] | NOCOMPRESS

Включает или выключает сжатие ключей индекса. При компрессии удаляются повторяющиеся значения в столбцах, что приводит к уменьшению занимаемого места и увеличению скорости работы. *Целое_число* определяет количество ключей для сжатия. Значение может варьироваться от 1 до n (здесь n – количество столбцов в индексе) для неуникального индекса и от 1 до *n-1* для уникального индекса. По умолчанию используется *NOCOMPRESS*, но если вы укажете *COMPRESS* без указания количества столбцов, то будет использоваться *COMPRESS n* для неуникальных индексов и *COMPRESS n-1* для уникальных индексов, где *n* – число столбцов в индексе. *COMPRESS* не может использоваться с секционированными или битовыми индексами.

NOSORT | SORT

NOSORT позволяет быстро создать индекс по столбцу, который уже отсортирован по возрастанию. Если при этом обнаруживается, что значения не отсортированы, то создание индекса прерывается с ошибкой. *NOSORT* обычно используется для построения индекса сразу после загрузки отсортированного набора данных.

REVERSE

REVERSE указывает, что байты в блоке индекса должны храниться в обратном порядке (за исключением *ROWID*). *REVERSE* исключает *NOSORT* и не может использоваться для битовых индексов и индексно-организованных таблиц.

VISIBLE | INVISIBLE

Делает индекс видимым или невидимым для оптимизатора запросов. Невидимые индексы поддерживаются в актуальном состоянии, но не используются оптимизатором для выполнения запросов. Эта опция полезна в тех случаях, когда вы не можете удалить индекс, но вам очень нужно, чтобы оптимизатор его не использовал.

PARALLEL [целое_число] | NOPARALLEL

Позволяет выполнять создание индекса, параллельно используя несколько серверных процессов, каждый из которых работает над определенным подмножеством индекса. По желанию вы можете указать точное число используемых процессов, иначе Oracle самостоятельно вычислит степень параллелизма. По умолчанию используется значение *NOPARALLEL*, и индекс создается последовательно.

ENABLE | DISABLE

Делает индекс на основе функции активным или неактивным. При использовании *ENABLE* или *DISABLE* вы не можете использовать другие опции оператора *ALTER INDEX*.

UNUSABLE

Помечает индекс (или его секцию или подсекцию) как неиспользуемый. Неиспользуемый индекс нужно перестроить или удалить и пересоздать, чтобы его опять можно было использовать.

RENAME TO новое_имя_индекса

Переименовывает индекс.

COALESCE

Соединяет содержимое блоков индекса, используемых для поддержки индексно-организованных таблиц, с целью повторного использования блоков. Опция *COALESCE* похожа на *SHRINK*, но *COALESCE* менее плотно сжимает блоки и не освобождает неиспользуемое пространство.

[NO]MONITORING USAGE

Начинает мониторинг использования индекса, предварительно очищая предыдущие результаты. Информация об использовании индекса отражается в системном представлении *V$OBJECT_USAGE*. Для завершения мониторинга необходимо явно выполнить оператор *ALTER INDEX...NOMONITORING USAGE*.

UPDATE BLOCK REFERENCES

Обновляет все устаревшие указатели (guess data blocks) на блоки данных таблицы в обычных и предметных индексах, созданных для индексно-организованных таблиц. Эта фраза не может использоваться одновременно с любыми другими фразами *ALTER INDEX*.

PARAMETERS ('параметры ODCI')

Определяет строку параметров, передаваемую без изменений в подпрограмму ODCI предметного индекса. Строка параметров может иметь длину до 1000 символов. За дополнительной информацией обращайтесь к документации производителя.

изменение_секционирования

Детальную информацию об этой опции читайте в разделе «Секционированные и подсекционированные таблицы», приведенном в описании оператора *CREATE/ALTER TABLE* для Oracle.

перестроение

Перестраивает индекс или отдельную секцию или подсекцию индекса. После успешного перестроения индекс из состояния *UNUSABLE* переводится в состояние *USABLE*. Для перестроения используется следующий синтаксис:

```
REBUILD {[NO]REVERSE | [SUB]PARTITION имя_секции}
  [{PARALLEL [int] | NOPARALLEL}]
  [TABLESPACE    tablespace_name]
  [PARAMETERS ('ODCI_params')] [ONLINE]
  [COMPUTE STATISTICS]
  [COMPRESS int | NOCOMPRESS] [[NO]LOGGING]
  [physical_attributes_clause]
```

где:

[NO]REVERSE

Хранит байты индексного блока за исключением ROWID в обратном порядке при перестроении индекса (*REVERSE*), либо хранит байты индексного блока в обычном порядке (*NOREVERSE*).

DEALLOCATE UNUSED [KEEP целое_число [K|M|G|T]]

Освобождает неиспользуемое пространство в конце индекса (или каждой секции секционированного индекса) и делает это пространство доступным для

других сегментов табличного пространства. Необязательное слово *KEEP* определяет, сколько (в байтах) свободного пространства сверх маркера максимального уровня заполнения (high-water mark) останется в индексе. Вы можете добавить суффикс для указания размера в килобайтах (*K*), мегабайтах (*M*), гигабайтах (*G*) или терабайтах (*T*). Если *KEEP* не используется, то освобождается все неиспользуемое пространство.

ALLOCATE EXTENT ([SIZE целое_число [K|M|G|T]] [DATAFILE 'имя_файла'] [INSTANCE целое_число])

Выделяет индексу новый экстент с указанными параметрами. Вы можете указывать параметры в различных сочетаниях. *SIZE* используется для указания размера экстента в байтах (без суффикса), в килобайтах (*K*), мегабайтах (*M*), гигабайтах (*G*) или терабайтах (*T*). *DATAFILE* указывает файл данных, в котором выделяется экстент. Параметр *INSTANCE* используется только в RAC и делает экстент доступным для группы списков свободных блоков, связанной с конкретным экземпляром базы данных.

SHRINK SPACE [COMPACT] [CASCADE]

Уменьшает сегменты индекса. Уменьшить можно только сегменты, расположенные в табличных пространствах с автоматическим управлением сегментами. Если не указаны ключевые слова *COMPACT* или *CASCADE*, то Oracle сжимает сегменты, освобождает неиспользуемое пространство и уменьшает значение маркера максимального уровня заполнения. Ключевое слово *COMPACT* используется только для дефрагментации индекса, маркер максимального уровня заполнения не уменьшается и свободное пространство не освобождается. *CASCADE* выполняет операцию уменьшения (с некоторыми ограничениями и исключениями) для всех зависимых объектов. Оператор *ALTER INDEX...SHRINK SPACE COMPACT* функционально эквивалентен оператору *ALTER INDEX...COALESCE*.

По умолчанию в Oracle создаются неуникальные индексы. Также важно знать, что индексы в виде B-деревьев не включают записи, имеющие значения NULL во всех ключевых полях.

Oracle не поддерживает индексы по столбцам следующих типов: *LONG*, *LONG RAW*, *REF* (с атрибутом *SCOPE*) и пользовательских типов. Вы можете создавать индексы по функциям (кроме агрегатных) и выражениям, но они не должны возвращать NULL. При создании индекса на базе функций для функций без параметров следует использовать пустой список (то есть, например, *function_name( )*). Если это пользовательская функция, то она должна быть детерминированной.

Oracle поддерживает специальную индексную структуру, называемую *индексно-организованной таблицей* (index-organized table, IOT). В индексно-организованной таблице атрибуты первичного ключа и неключевые атрибуты хранятся в единой физической структуре вместо обычно используемых отдельных структур для таблицы и для индекса. Индексно-организованные таблицы создаются при помощи оператора *CREATE TABLE...ORGANIZATION INDEX*. Дополнительная информация об индексно-организованных таблицах приводится в разделе, посвященном оператору *CREATE/ALTER TABLE*.

Дополнительные индексы для индексно-организованных таблиц создаются как вторичные. Вторичные индексы не поддерживают опцию *REVERSE*.

Обратите внимание, что при каждом использовании в синтаксических диаграммах имен объектов вы можете добавлять к этим именам имя схемы. Это относится к таблицам, индексам и др. (за исключением табличных пространств). Для создания индекса не в текущей схеме вы должны иметь соответствующие привилегии.

В качестве примера рассмотрим параллельное создание без журналирования сжатого индекса, с собранной статистикой:

```
CREATE UNIQUE INDEX unq_pub_id
    ON publishers(pub_name, country)
COMPRESS 1 PARALLEL NOLOGGING COMPUTE STATISTICS;
```

Для индексов, как и для других объектов базы данных, можно контролировать занимаемое индексом пространство и размер, на который он увеличивается. В следующем примере создается индекс в определенном табличном пространстве и с заданными параметрами хранения:

```
CREATE UNIQUE INDEX unq_pub_id
    ON publishers(pub_name, country)
STORAGE (INITIAL 10M NEXT 5M PCTINCREASE 0)
TABLESPACE publishers;
```

Если, например, таблица **housing_construction** секционирована, то вы можете создать секционированный индекс со своими секциями:

```
CREATE UNIQUE CLUSTERED INDEX project_id_ind
ON housing_construction(project_id)
GLOBAL PARTITION BY RANGE (project_id)
  (PARTITION part1 VALUES LESS THAN ('H')
    TABLESPACE construction_part1_ndx_ts,
  PARTITION part2 VALUES LESS THAN ('P')
    TABLESPACE construction_part2_ndx_ts,
  PARTITION part3 VALUES LESS THAN (MAXVALUE)
    TABLESPACE construction_part3_ndx_ts);
```

Если для таблицы **housing_construction** используется составное секционирование, то мы можем создать соответствующий индекс:

```
CREATE UNIQUE CLUSTERED INDEX project_id_ind
ON housing_construction(project_id)
STORAGE (INITIAL 10M MAXEXTENTS UNLIMITED)
LOCAL (PARTITION part1 TABLESPACE construction_part1_ndx_ts,
       PARTITION part2 TABLESPACE construction_part2_ndx_ts
          (SUBPARTITION subpart10, SUBPARTITION subpart20,
           SUBPARTITION subpart30, SUBPARTITION subpart40,
           SUBPARTITION subpart50, SUBPARTITION subpart60),
PARTITION part3 TABLESPACE construction_part3_ndx_ts);
```

В следующем примере мы перестраиваем существующий индекс **project_id_ind**. Мы делаем индекс инвертированным и перестраиваем его с использованием параллельных процессов:

```
ALTER INDEX project_id_ind
REBUILD REVERSE PARALLEL;
```

Мы также можем разбить секцию индекса на две:

```
ALTER INDEX project_id_ind
SPLIT PARTITION part3 AT ('S')
INTO (PARTITION part3_a TABLESPACE constr_p3_a LOGGING,
      PARTITION part3_b TABLESPACE constr_p3_b);
```

PostgreSQL

В PostgreSQL можно создавать индексы, отсортированные оп возрастанию, а также уникальные индексы. Также реализованы некоторые возможности для повышения производительности, настраиваемые фразой *USING*. Оператор *CREATE INDEX* имеет следующий синтаксис:

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] имя_индекса
    ON имя_таблицы
[USING {BTREE | RTREE | HASH | GIST}]
{имя_функции | (столбец [, ...])}
[WITH FILLFACTOR = целое_число]
[TABLESPACE табличное_пространство]
[WHERE условие]
```

Синтаксис *ALTER INDEX* следующий:

```
ALTER INDEX имя_индекса
[RENAME TO новое_имя_индекса]
[SET TABLESPACE табличное_пространство]
[SET FILLFACTOR = целое_число]
[RESET FILLFACTOR = целое_число]
```

где:

CONCURRENTLY

Строит индекс без установки блокировок, предотвращающих вставку, обновление и удаление записей таблицы. В обычном режиме PostgreSQL блокирует запись в таблицу (но не чтение) до окончания операции. Построение индекса с опцией *CONCURRENTLY* не рекомендуется в том случае, если вы не уверены, что таблица не будет модифицироваться.

USING {BTREE | RTREE | HASH | GIST}

Определяет один из трех (*RTREE* - устарело) динамических методов доступа для улучшения производительности. Особенно важно, что индексы являются полностью динамическими и не требуют периодического обновления статистики.[1] *USING* имеет следующие опции:

BTREE

Используется алгоритм B-деревьев Лемана-Яо с высокой степенью параллельности (high concurrency). Этот алгоритм используется по умолчанию. Индексы на основе B-деревьев могут быть использованы для операций сравнения, таких как =, <, <=, >, >=. Индексы на основе B-деревьев могут быть построены по нескольким столбцам.

---

[1] При настройке PostgreSQL по умолчанию специальный демон autovacuum заботится об автоматическом анализе таблиц при их первоначальной загрузке, а также о периодической актуализации собранной статистики. Когда autovacuum отключен, необходимо периодически вручную обновлять статистику командой ANALYZE . – *Прим. науч. ред.*

RTREE

Используется алгоритм R-деревьев Гутмана. Индексы на основе R-деревьев могут быть использованы для операций сравнения <<, &<, &>, >>, @, ~= и &&. Эти индексы должны быть одностолбцовыми. На текущий момент это ключевое слово устарело и убрано из последней версии документации PostgreSQL, а в случае использования заменяется на *USING GIST*.

HASH

Используется алгоритм линейного хеширования Литвина. Хеш-индексы используются только при операциях = и должны быть одностолбцовыми.

GIST

Используется алгоритм GIST (Generalized Index Search Trees). *GIST*-индексы могут быть многостолбцовыми.

имя_функции

Указывает имя функции, используемой в качестве значений для построения индекса. Индекс по функции и обычный индекс по столбцам взаимно исключают друг друга.

WITH FILLFACTOR = целое_число

Определяет процент заполнения каждой страницы индекса при его создании. Для индексов на основе B-деревьев этот параметр используется как при создании индекса, так и при его расширении. По умолчанию используется значение 90. PostgreSQL не поддерживает заполнение блоков на указанном уровне в процессе работы, поэтому во избежание излишней фрагментации и расщепления страниц рекомендуется периодически перестраивать индекс.

TABLESPACE табличное_пространство

Указывает, в каком табличном пространстве создается индекс.

WHERE условие

Определяет условие отбора для построения частичного индекса. Частичный индекс содержит записи не для всех строк таблицы, а только для определенного множества. Используя эту фразу, можно получить интересные эффекты. Комбинацией *UNIQUE* и *WHERE* можно обеспечить уникальность значений на некотором множестве записей таблицы, а не на всей таблице. При использовании *WHERE* необходимо:

- Ссылаться на столбцы таблицы (не обязательно только на те, по которым построен индекс).
- Не использовать агрегатные функции.
- Не использовать подзапросы.

[RENAME TO новое_имя_индекса] [SET TABLESPACE табличное_пространство] [SET FILLFACTOR = целое_число] [RESET FILLFACTOR]

Позволяет изменить свойства существующего индекса, например переименовать его, использовать другое табличное пространство, установить новое значение для процента заполнения блоков. Мы рекомендуем вместо опций *SET FILLFACTOR* и *RESET FILLFACTOR* перестраивать индекс целиком командой *REINDEX* для немедленного вступления изменений в силу.

В PostgreSQL столбец на основании его типа связывается с определенным *классом операторов*. Класс операторов определяет набор операторов, допустимых для определенного индекса. Хотя пользователи могут свободно менять класс операторов для каждого столбца, по умолчанию используется класс операторов, соответствующий типу данных столбца.

В следующем примере мы создаем уникальный индекс типа *GIST* на подмножестве издателей, расположенных вне США:

```
CREATE UNIQUE INDEX unq_pub_id
  ON publishers(pub_name, country)
USING GIST
WHERE country <> 'USA';
```

SQL Server

В SQL Server для оператора *CREATE INDEX* используется следующий синтаксис:

```
CREATE [UNIQUE] [[NON]CLUSTERED] INDEX имя_индекса
ON {таблица | представление} (столбец [ASC | DESC][, ...])
[INCLUDE (столбец [ASC | DESC][, ...])]
[WITH [PAD_INDEX = {ON | OFF}] [FILLFACTOR =целое_число] [IGNORE_DUP_KEY = {ON | OFF}]
    [STATISTICS_NORECOMPUTE = {ON | OFF}]
    [DROP_EXISTING = {ON | OFF}]
    [ONLINE = {ON | OFF}] [SORT_IN_TEMPDB = {ON | OFF}]
    [ALLOW_ROW_LOCKS = {ON | OFF}]
    [ALLOW_PAGE_LOCKS = {ON | OFF}]
    [MAXDOP = целое_число][, ...]]
[ON {файловая_группа | секция (столбец) | DEFAULT}]
[FILESTREAM_ON { имя_файлового_потока | секция | "NULL"}]
```

Синтаксис оператора *ALTER INDEX* следующий:

```
ALTER INDEX {имя_индекса | ALL} ON {имя_объекта}
{ DISABLE |
  REBUILD [PARTITION = номер_секции] [WITH
    ( [ SORT_IN_TEMPDB = {ON | OFF} ]
        [MAXDOP = целое_число][, ...] )]
    [WITH [PAD_INDEX = {ON | OFF}][FILLFACTOR = целое_число]
        [IGNORE_DUP_KEY = {ON | OFF}]
        [STATISTICS_NORECOMPUTE = {ON | OFF}]
        [SORT_IN_TEMPDB = {ON | OFF}]
        [ALLOW_ROW_LOCKS = {ON | OFF}]
        [ALLOW_PAGE_LOCKS = {ON | OFF}]
        [MAXDOP = целое_число][, ...]] |
  REORGANIZE [PARTITION = номер_секции]
  [WITH (LOB_COMPACTION = {ON | OFF})] |
  SET [ALLOW_ROW_LOCKS = {ON | OFF}]
    [ALLOW_PAGE_LOCKS = {ON | OFF}]
    [IGNORE_DUP_KEY = {ON | OFF}]
    [STATISTICS_NORECOMPUTE = {ON | OFF}][, ...] }
```

где:

[NON]CLUSTERED

Индекс типа *CLUSTERED* определяет физическое упорядочивание записей при хранении на диске. Столбцы кластерного (*CLUSTERED*) индекса исполь-

зуются для сортировки записей таблицы. То есть, если в таблице **Foo** создан возрастающий *CLUSTERED*-индекс по столбцу **A**, то записи таблицы буду записываться на диск в алфавитном порядке. Опция *NONCLUSTERED* используется для создания вторичного индекса, содержащего только указатели на данные и не определяющего физический порядок строк на диске.

ASC | DESC

Указывает, что значения в индексе хранятся по возрастанию (*ASC*) или по убыванию (*DESC*). Если не указано ни то ни другое, используется возрастающий порядок.

WITH

Используется для определения дополнительных атрибутов индекса.

PAD_INDEX = {ON | OFF}

Указывает, что в конце каждой 8-килобайтной страницы индекса следует оставлять неиспользуемое пространство в соответствии со значением параметра *FILLFACTOR*.

FILLFACTOR = целое_число

Определяет процент заполнения каждой 8-килобайтной страницы индекса при его создании. Это полезно для уменьшения конкуренции за страницы и расщепления страниц при их заполнении. Создание кластерного индекса с явно указанным значением *FILLFACTOR* увеличивает размер индекса, но ускоряет работу с ним в определенных ситуациях.

IGNORE_DUP_KEY = {ON | OFF}

Определяет, что происходит, когда операция вставки пытается вставить в уникальный индекс повторяющиеся значения ключа. Параметр *IGNORE_DUP_KEY* применяется только к операциям вставки, производимым после создания или перестроения индекса. Параметр не работает во время выполнения инструкции *CREATE INDEX*, *ALTER INDEX* или *UPDATE*. Значение по умолчанию – *OFF*.

Если установлено *ON*, то выводится предупреждающее сообщение. С ошибкой завершаются только строки, нарушающие ограничение уникальности.

Если установлено *OFF*, то выводится сообщение об ошибке. Будет выполнен откат всей операции INSERT.

DROP_EXISTING = {ON | OFF}

Указывает, что названный существующий кластеризованный или некластеризованный индекс удаляется и перестраивается.

STATISTICS_NORECOMPUTE = {ON | OFF}

Позволяет не собирать статистику при создании индекса. Это дает возможность ускорить выполнение *CREATE INDEX*, но может сделать работу оптимизатора менее эффективной.

ONLINE = {ON | OFF}

Определяет возможность использования таблицы для запросов и изменения данных в момент создания индекса. По умолчанию установлено значение *OFF*. В режиме *ON* создаются не долговременные блокировки на запись, а только разделяемые блокировки.

SORT_IN_TEMPDB = {ON | OFF}

Сохраняет в системной базе данных *TEMPDB* промежуточные данные, используемые при построении индекса. Это увеличивает пространство, необходимое для построения индекса, но может ускорить работу с базой данных, если база *TEMPDB* расположена на ином диске, чем таблица и индекс.

ALLOW_ROW_LOCKS = {ON | OFF}

Указывает допустимость блокировок на уровне строк. По умолчанию используется значение *ON*.

ALLOW_PAGE_LOCKS = {ON | OFF}

Указывает допустимость блокировок на уровне страниц. По умолчанию используется значение *ON*.

MAXDOP = целое_число

Устанавливает степень параллелизма операции по созданию индекса. Значение 1 отключает параллелизм. Любое значение больше 1 ограничивает степень параллелизма указанным числом процессоров. 0, значение по умолчанию, оставляет определение степени параллелизма на усмотрение сервера.

ON файловая_группа

Создает индекс в заданной файловой группе. Это позволяет расположить индекс на отдельном диске или RAID-массиве. Оператор *CREATE CLUSTERED INDEX...ON FILEGROUP* позволяет эффективно перенести индекс в новую файловую группу, так как листовые элементы индекса являются страницами с данными.

DISABLE

Отключает индекс, делая его недоступным для запросов. Отключение некластерного индекса не влияет на данные таблицы, по которой индекс построен. Отключение кластерного индекса делает недоступной для запросов всю таблицу целиком. Вы можете вновь включить индекс при помощи команд *ALTER INDEX REBUILD* или *CREATE INDEX WITH DROP_EXISTING*.

REBUILD [PARTITION = номер_секции]

Перестраивает индекс с использованием существующих параметров, включая используемый набор столбцов, тип индекса, уникальность и порядок сортировки. По желанию вы можете указать новую секцию. Эта команда *не* перестраивает автоматически связанные некластерные индексы, если только не указано ключевое слово *ALL*. При использовании этой фразы для перестроения индекса по XML или пространственным данным нельзя использовать *ONLINE=ON* и *IGNORE_DUP_KEY=ON*. Эквивалентно *DBCC DBREINDEX*.

REORGANIZE [PARTITION = номер_секции]

Выполняет оперативную реорганизацию элементов индекса листового уровня (то есть не накладываются долговременные блокировки и не блокируются запросы и обновления). Можно дополнительно указать новую секцию. Нельзя использовать совместно с *ALLOW_PAGE_LOCKS=OFF*. Эквивалентно *DBCC INDEXDEFRAG*.

WITH (LOB_COMPACTION = {ON | OFF})

Сжимает все страницы, содержащие большие объекты (*LOB*), включая *IMAGE*, *TEXT*, *NTEXT*, *VARCHAR(MAX)*, *NVARCHAR(MAX)*, *VARBINARY(MAX)* и *XML*. По умолчанию используется *ON*. Фраза игнорируется, если не используются столбцы указанных типов. При указании *ALL* реорганизуются все индексы таблицы или представления.

SET

Устанавливает параметры индекса без его перестроения или реорганизации. *SET* нельзя использовать с отключенными индексами.

SQL Server позволяет строить уникальный кластерный индекс по представлению для материализации представления. Это может значительно ускорить извлечение данных из представления. После создания кластерного индекса по представлению можно также построить дополнительные некластерные индексы. Имейте в виду, что представление должно быть создано с опцией *SCHEMABINDING*. Индексирование представлений возможно только в редакции SQL Server Enterprise Edition, если только в представлении не используется подсказка *NOEXPAND*. Через индексированные представления возможно извлечение данных, но не модификация.

В SQL Server можно создать до 249 некластеризованных индексов (уникальных и неуникальных) по одной таблице, включая один индекс для первичного ключа. Нельзя индексировать столбцы типов *NTEXT*, *TEXT* и *IMAGE*.

SQL Server автоматически распараллеливает процесс создания индекса в соответствии с параметром конфигурации *максимальная степень параллелизма*.

Часто требуется построить индекс по нескольким столбцам – составной индекс. Ключ составного индекса может содержать до 16 столбцов, а общая длина ключа должна составлять не более 900 байт по всем столбцам с фиксированной длиной. Вот пример создания такого индекса:

```
CREATE UNIQUE INDEX project2_ind
ON housing_construction(project_name, project_date)
WITH PAD_INDEX, FILLFACTOR = 80
ON FILEGROUP housing_fg
GO
```

Добавление параметра *PAD_INDEX* и *FILLFACTOR = 80* создает индекс, страницы которого заполнены на 80%, а не на все 100%. Также указывается, что индекс нужно создать не в файловой группе по умолчанию, а в группе **housing_fg**.

См. также

CREATE/ALTER TABLE
DROP

CREATE/ALTER METHOD

Операторы *CREATE/ALTER METHOD* создают и изменяют методы в базе данных. Если попытаться объяснить просто (но не очень точно), то метод – это пользовательская функция, связанная с пользовательским типом данных.

Например, метод *getBonus* типа *Emp* мог бы принимать входной параметр с процентной ставкой и возвращать для сотрудника его бонус по формуле salary*rate (пример для DB2):

```
CREATE METHOD getBonus (rate DOUBLE)
  FOR Emp
  RETURN SELF..salary * rate
```

Неявный метод создается для каждого пользовательского типа. С помощью комбинации операторов *CREATE TYPE* и *CREATE METHOD* создаются пользовательские методы.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Не поддерживается
PostgreSQL	Не поддерживается
SQL Server	Не поддерживается

Синтаксис SQL2003

```
{CREATE | ALTER} [INSTANT | STATIC] METHOD имя_метода
    ( [{IN | OUT | INOUT}] параметр тип_данных
        [AS LOCATOR] [RESULT][, ...] )
RETURNS тип_данных
FOR пользовательский_тип_данных
[SPECIFIC специальное_имя] блок_кода
```

Ключевые слова

{CREATE|ALTER} [INSTANT|STATIC] `имя_метода`

Создает новый метод или изменяет существующий. Для метода можно указать опции *INSTANT* и *STATIC*.

([{IN|OUT|INOUT}] `параметр тип_данных`*[, ...])*

Определяет один или несколько (через запятую, в скобках) параметров, передаваемых в метод. Параметры могут быть входными (*IN*), выходными (*OUT*) или двунаправленными (*INOUT*). Синтаксис для объявления параметров следующий:

```
[{IN | OUT | INOUT}] параметр1 тип_данных,
[{IN | OUT | INOUT}] параметр2 тип_данных,[...]
```

Имя параметра должно быть уникально в пределах метода. С помощью оператора *ALTER* можно добавить новые параметры. Информация о типах данных приводится в главе 2.

AS LOCATOR

Используется для валидации внешних подпрограмм, возвращающих параметры типов *BLOB*, *CLOB*, *NCLOB*, *ARRAY* и пользовательских типов. Другими словами, возвращается указатель на большой объект, а не сам объект.

RESULT

Обозначает пользовательский тип данных, не используется для стандартных типов.

RETURNS тип_данных

Определяет тип данных результата, возвращаемого методом. Основным назначением пользовательского метода является возврат результата. Если нужно на лету менять тип возвращаемого результата, то используйте фразу *CAST* (обратитесь к описанию функции *CAST*), например *RETURNS VARCHAR(12) CAST FROM DATE*.

FOR пользовательский_тип_данных

Связывает метод с уже существующим пользовательским типом данных, созданным с помощью оператора *CREATE TYPE*.

SPECIFIC специальное_имя

Уникально идентифицирует функцию, обычно используется с пользовательскими типами данных.

Общие правила

Методы – это просто другой подход для получения того же результата, что и при применении пользовательских функций. Например, рассмотрим два следующих фрагмента кода:

```
CREATE FUNCTION my_fcn (order_udt)
RETURNS INT;
CREATE METHOD my_mthd ( )
RETURNS INT
FOR order_udt;
```

Хотя код функции и метода отличается, они делают одно и то же. Правила использования и вызова методов такие же, как для функций.

Советы и хитрости

Основной сложностью операторов *CREATE METHOD* является то, что они используют объектно-ориентированный подход для той же функциональности, что и обычные пользовательские функции. При наличии двух различных подходов бывает сложно выбрать, какой из них использовать.

MySQL

Не поддерживается.

Oracle

Не поддерживается.

PostgreSQL

Не поддерживается.

SQL Server

Не поддерживается.

См. также

CREATE/ALTER TYPE

CREATE ROLE

Оператор CREATE ROLE позволяет создавать именованный набор привилегий, которые затем выдаются пользователям базы данных. Если пользователю назначена роль, то он получает все привилегии и разрешения этой роли. Роли являются общепринятым механизмом для поддержки безопасности и контроля привилегий в базе данных.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
CREATE ROLE имя_роли[WITH ADMIN {CURRENT_USER|CURRENT_ROLE}]
```

Ключевые слова

CREATE ROLE имя_роли

Создает новую роль. Привилегии можно выдавать непосредственно пользователю, а можно того же результата добиться, выдавая привилегии роли, а уже роль – пользователю. Важное ротличие состоит в том, что роль можно назначить одному или нескольким пользователям, и каждый из них получит все привилегии этой роли.

WITH ADMIN {CURRENT_USER|CURRENT_ROLE}

Немедленно назначает роль текущему пользователю или текущей активной роли с возможностью назначать роль другим пользователям. По умолчанию используется *WITH ADMIN CURRENT_USER*.

Общие правила

Управление безопасностью с помощью ролей сильно упрощает администрирование и поддержку пользователей. Для использования ролей необходимо выполнить следующие шаги:

1. Определите необходимые роли и выберите для них названия (например, *administrator*, *manager*, *data_entry*, *report_writer* и т. д.).
2. Используя оператор *GRANT*, выдайте каждой роли необходимые привилегии, как если бы это был обычный пользователь. Например, роли *manager* могли бы быть предоставлены права на чтение и запись любой таблицы в базе данных, а роли *report_writer* – права на чтение таблиц, используемых для построения отчетов.

3. Используйте оператор *GRANT*, чтобы назначить пользователям роли, соответствующие типу выполняемой ими работы.

Роли и привилегии отключаются при помощи оператора *REVOKE*.

Советы и хитрости

Иногда администратор базы данных может одновременно выдать привилегию пользователю напрямую и через роль. Это может быть проблемой, так как если вам понадобиться забрать эту привилегию у пользователя, то это придется делать дважды: сперва отключить у пользователя роль, а затем отключить привилегию, выданную пользователю напрямую.

MySQL

Не поддерживается.

Oracle

В Oracle поддерживается оператор *ALTER ROLE*, не описанный в стандарте ANSI. Oracle поддерживает концепцию ролей, хотя реализация соответствующих операторов значительно отличается от ANSI:

```
{CREATE | ALTER} ROLE имя_роли
  [NOT IDENTIFIED |
  IDENTIFIED {BY пароль| EXTERNALLY | GLOBALLY |
    USING имя_пакета}]
```

где:

{CREATE|ALTER} ROLE *имя_роли*

Указывает имя создаваемой или изменяемой роли.

NOT IDENTIFIED

Указывает, что авторизация роли осуществляется базой данных и для ее включения не требуется пароль. Является настройкой по умолчанию.

IDENTIFIED

Указывает, что перед активацией роли оператором *SET ROLE* пользователь должен быть аутентифицирован указанным способом:

BY *пароль*

Указывает пароль, используемый для аутентификации. В качестве пароля используется строка однобайтных символов, даже если в базе данных используется многобайтная кодировка.

EXTERNALLY

Создает роль с аутентификацией, выполняемой операционной системой или сторонним программным обеспечением. В любом случае внешняя аутентификация вероятнее всего также потребует пароль.

GLOBALLY

Создает глобальную роль, аутентифицируемую с помощью корпоративной службы каталогов, например LDAP-каталогом.

USING имя_пакета

Создает роль, активируемую через указанный PL/SQL пакет. Если схема не указана, то подразумевается, что пакет находится в текущей схеме.

В Oracle роль сперва создается, затем с помощью оператора *GRANT* роли, как и обычному пользователю, выдаются необходимые разрешения и привилегии. Если пользователь хочет воспользоваться разрешениями роли, защищенной паролем, он должен выполнить команду *SET ROLE*, в которой необходимо указать пароль.

Oracle поставляется с несколькими предварительно сконфигурированными ролями. Роли *CONNECT*, *DBA* и *RESOURCE* имеются во всех версиях Oracle. *EXP_FULL_DATABASE* и *IMP_FULL_DATABASE* – новые роли, используемые для операций экспорта и импорта данных. Детальная информация о стандартных ролях Oracle приводится в описании оператора *GRANT*.

В следующем примере при помощи оператора *CREATE* создается новая роль, затем этой роли выдаются привилегии, далее оператором *ALTER ROLE* устанавливается пароль и, наконец, роль назначается нескольким пользователям:

```
CREATE ROLE boss;
GRANT ALL ON employee TO boss;
GRANT CREATE SESSION, CREATE DATABASE LINK TO boss;
ALTER ROLE boss IDENTIFIED BY le_grand_fromage;
GRANT boss TO emily, jake;
```

PostgreSQL

PostgreSQL поддерживает операторы *ALTER* и *CREATE ROLE*, а также предлагает практически идентичные операторы *ALTER/CREATE GROUP*. Синтаксис *CREATE ROLE* следующий:

```
{CREATE | ALTER} ROLE имя_роли
[ [WITH] [[NO]SUPERUSER] [[NO]CREATEDB] [[NO]CREATEUSER] [[NO]INHERIT]
[[NO]LOGIN]
  [CONNECTION LIMIT целое_число]
  [ {ENCRYPTED | UNENCRYPTED} PASSWORD 'пароль' ]
  [VALID UNTIL 'дата и время'] [IN ROLE имя_роли[, ...]]
  [IN GROUP имя_группы[, ...]] [ROLE имя_роли [, ...]]
  [ADMIN имя_роли [, ...]] [USER имя_роли [, ...]]
  [SYSID целое_число][...] ]
[RENAME TO новое_имя]
[SET параметр {TO | =} {значение | DEFAULT}]
[RESET параметр]
```

где:

{CREATE|ALTER} ROLE имя_роли

Указывает имя создаваемой или изменяемой роли.

[NO]SUPERUSER

Определяет, является роль привилегированной или нет. Привилегированная роль имеет приоритет над любыми ограничениями доступа в базе данных. По умолчанию используется *NOSUPERUSER*.

[NO]CREATEDB

Определяет, имеет ли роль права на создание баз данных. По умолчанию используется *NOCREATEDB*.

[NO]CREATEROLE

Определяет, имеет ли роль права на создание, модификацию и удаление других ролей. По умолчанию используется *NOCREATEROLE*.

[NO]CREATEUSER

Определяет, имеет ли роль права на создание пользователей. Это ключевое слово устарело, вместо него нужно использовать *[NO]SUPERUSER*.

[NO]INHERIT

Определяет, наследует ли роль привилегии тех ролей, членом которых она является. Если установлено значение *INHERIT*, то роль автоматически получает привилегии ролей, членом которых она является (напрямую или опосредованно). По умолчанию используется *NOINHERIT*.

[NO]LOGIN

Определяет, может ли роль подключаться к базе данных. Роль с опцией *LOGIN* является просто пользователем. Роль с *NOLOGIN* является набором привилегий в базе данных. По умолчанию используется *NOLOGIN*.

CONNECTION LIMIT *целое_число*

Определяет максимальное допустимое число одновременных подключений для роли с опцией *LOGIN*. По умолчанию используется значение -1, что означает отсутствие ограничения.

{ENCRYPTED | UNENCRYPTED} PASSWORD '*пароль*'

Устанавливает пароль для роли с опцией *LOGIN*. Пароль может храниться как обычный текст (*UNENCRYPTED*) или зашифрованный по алгоритму MD5 (*ENCRYPTED*). Старые клиенты могут не поддерживать аутентификацию посредством MD5, так что будьте осторожны.

VALID UNTIL '*дата и время*'

Для роли с опцией *LOGIN* устанавливает дату и время, когда истекает срок ее действия. По умолчанию срок действия роли не ограничен.

IN ROLE, IN GROUP

Указывает роли и группы (хотя группы устарели), членом которых является данная роль.

ROLE, GROUP

Указывает роли и группы (хотя группы устарели), автоматически становящиеся членами данной роли.

ADMIN *имя_роли*

Делает то же, что и опция *ROLE*, только роли добавляются с параметром *WITH ADMIN OPTION*, что позволяет им предоставлять членство в роли другим ролям.

SYSID целое_число, USER имя_пользователя

Устаревшие параметры, оставленные для обратной совместимости. *USER* эквивалентно *ROLE*, *SYSID* эквивалентно *GROUP*.

[RENAME TO новое_имя] [SET параметр {TO|=} {значение |DEFAULT}] [RESET параметр]

Переименовывает роль, устанавливает для параметра новое значение или значение по умолчанию. Конфигурационные параметры описаны в документации PostgreSQL.

Для удаления роли используйте *DROP ROLE*.

SQL Server

SQL Server поддерживает операторы *CREATE* и *ALTER ROLE*, а также аналогичную функциональность через системную хранимую процедуру **sp_add_role**. Используется следующий синтаксис:

```
CREATE ROLE имя_роли [AUTHORIZATION имя_владельца]
    [WITH NAME = новое_имя]
```

где:

AUTHORIZATION имя_владельца

Указывает для роли имя владельца. По умолчанию владельцем роли становится тот пользователь, который ее создает. Используется только в *CREATE ROLE*.

WITH NAME = новое_имя

Устанавливает для роли новое имя. Используется только в *ALTER ROLE*.

См. также

GRANT

REVOKE

CREATE SCHEMA

Этот оператор создает *схему*, то есть именованную группу связанных объектов. Схема – это набор таблиц, представлений и разрешений, выданных пользователям и ролям. В соответствии со стандартом ANSI разрешения на объекты не являются объектами сами по себе, и поэтому не принадлежат конкретной схеме. Однако роли являются наборами привилегий и принадлежат конкретным схемам.

СУБД	Уровень поддержки
MySQL	Поддерживается (как CREATE DATABASE)
Oracle	Поддерживается с вариациями
PostgreSQL	Не поддерживается
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

```
CREATE SCHEMA [имя_схемы] [AUTHORIZATION имя_владельца]
[DEFAULT CHARACTER SET кодировка]
[PATH имя_схемы[, ...]]
  [ ANSI CREATE операторы [...] ]
  [ ANSI GRANT операторы [...] ]
```

Ключевые слова

CREATE SCHEMA [имя_схемы]

Создает схему с заданным именем. Если имя не указано, то создается схема с именем пользователя, создавшего схему.

AUTHORIZATION имя_владельца

Указывает имя владельца схемы. По умолчанию владельцем схемы становится ее создатель. Стандарт ANSI позволяет не указывать либо имя схемы, либо фразу *AUTHORIZATION*, но можно указать оба параметра одновременно.

DEFAULT CHARACTER SET кодировка

Устанавливает кодировку, используемую по умолчанию для всех создаваемых в схеме объектов.

PATH имя_схемы[, ...]

Устанавливает список схем, в которых будет осуществляться поиск подпрограмм, если явно не задать схему (то есть хранимых процедур, пользовательских функций и методов).

ANSI CREATE операторы

Содержит один или несколько операторов *CREATE*. Между операторами запятые не ставятся.

ANSI GRANT операторы

Содержит операторы *GRANT*, применяющиеся к существующим объектам. Обычно это объекты, создаваемые в этом же операторе *CREATE SCHEMA*, но также это могут быть любые другие объекты. Между операторами не используются запятые.

Общие правила

Оператор *CREATE SCHEMA* является контейнером, который может содержать другие операторы *CREATE* и *GRANT*. Проще всего представлять схему как набор всех объектов, принадлежащих определенному пользователю. Например, пользователь *jake* может быть владельцем нескольких таблиц и представлений в своей схеме, включая таблицу **publishers**. В то же время пользователь *dylan* также может быть владельцем нескольких таблиц и представлений в своей схеме и также может иметь свою собственную таблицу **publishers**.

Стандарт ANSI требует, чтобы в операторе *CREATE SHEMA* можно было использовать любые другие операторы *CREATE*. Однако на практике большинство реализаций допускают применение только трех подчиненных операторов: *CREATE TABLE*, *CREATE VIEW* и *GRANT*. Порядок команд не важен, то есть вы можете написать операторы выдачи привилегий раньше операторов создания объектов, на которые выдаются привилегии (хотя такой подход не рекомендуется).

Советы и хитрости

Хорошей практикой является написание операторов *CREATE* и *GRANT* внутри *CREATE SCHEMA* в порядке их действительного выполнения. Другими словами, оператор *CREATE VIEW* должен идти после оператора *CREATE TABLE*, от которого он зависит, а соответствующий оператор *GRANT* должен идти последним.

Если в вашей базе данных используются схемы, то мы рекомендуем всегда указывать полные имена объектов с указанием схемы (например, **jake.publishers**). Если вы не укажете схему явно, то в большинстве случаев будет подразумеваться текущая схема.

Некоторые платформы не поддерживают явно оператор *CREATE SCHEMA*. Однако они неявно создают схему при создании пользователем объектов. Например, Oracle создает схему для каждого пользователя. Оператор *CREATE SCHEMA* – это просто возможность одним шагом создать таблицы, представления и другие объекты вместе с соответствующими привилегиями.

MySQL

В MySQL оператор *CREATE SCHEMA* поддерживается как синоним оператора *CREATE DATABASE*. За информацией обращайтесь к соответствующему разделу.

Oracle

В Oracle оператор *CREATE SCHEMA* на самом деле не создает схему, схема создается оператором *CREATE USER*. Что делает *CREATE SCHEMA*, так это выполняет в ранее созданной схеме набор операторов *CREATE* и *GRANT* одной командой:

```
CREATE SCHEMA AUTHORIZATION имя_схемы
    [ ANSI CREATE операторы [...] ]
    [ ANSI GRANT операторы [...] ]
```

Имейте в виду, что в операторе *CREATE SCHEMA* можно применить только *CREATE TABLE*, *CREATE VIEW* и *GRANT*, причем в этом случае *нельзя* использовать ни одно из нестандартных расширений указанных команд.

В следующем примере выдача привилегий на объекты указывается раньше, чем создаются сами объекты:

```
CREATE SCHEMA AUTHORIZATION emily
  GRANT SELECT, INSERT ON view_1 TO sarah
  GRANT ALL ON table_1 TO sarah
  CREATE VIEW view_1 AS
    SELECT column_1, column_2
    FROM table_1
    ORDER BY column_2
  CREATE TABLE table_1(column_1 INT, column_2 CHAR(20));
```

Этот пример показывает, что порядок операторов внутри *CREATE SCHEMA* не важен: Oracle фиксирует *CREATE SCHEMA* только в том случае, если успешно выполняются все вложенные операторы.

PostgreSQL

В PostgreSQL поддерживаются и *CREATE*, и *ALTER SCHEMA*, но не поддерживаются фразы *PATH* и *DEFAULT CHARACTER SET*. Синтаксис *CREATE SCHEMA* следующий:

```
CREATE SCHEMA { имя_схемы [AUTHORIZATION имя_пользователя] |
    AUTHORIZATION имя_пользователя }
[ ANSI CREATE операторы [...] ]
[ ANSI GRANT операторы [...] ]
```

Если не указать имя схемы, то в качестве имени схемы будет использовано имя пользователя-владельца схемы. Внутри оператора *CREATE SCHEMA* поддерживаются только следующие операторы *CREATE*: *CREATE TABLE*, *CREATE VIEW*, *CREATE INDEX*, *CREATE SEQUENCE* и *CREATE TRIGGER*. Другие операторы *CREATE* нужно выполнять отдельно.

Синтаксис *ALTER SCHEMA* следующий:

```
ALTER SCHEMA имя_схемы [RENAME TO новое_имя_схемы ]
    [OWNER TO новое_имя_пользователя]
```

Оператор *ALTER SCHEMA* позволяет переименовывать схему или присваивать ее новому владельцу.

SQL Server

SQL Server поддерживает базовую функциональность оператора *CREATE SCHEMA*, без фраз *PATH* и *DEFAULT CHARACTER SET*:

```
CREATE SCHEMA AUTHORIZATION имя_пользователя
    [ ANSI CREATE операторы [...] ]
    [ ANSI GRANT операторы [...] ]
```

Если возникает ошибка в любом из вложенных операторов, то оператор *CREATE STATEMENT* завершается с ошибкой.

В SQL Server не требуется, чтобы вложенные операторы *CREATE* и *GRANT* шли в каком-то определенном порядке, за исключением того, что в логическом порядке должны создаваться представления. То есть, если представление **view_100** ссылается на представление **view_10**, то создание **view_10** должно появиться в *CREATE SCHEMA* раньше, чем создание **view_100**.

Например:

```
CREATE SCHEMA AUTHORIZATION katie
  GRANT SELECT ON view_10 TO public
  CREATE VIEW view_10(col1) AS SELECT col1 FROM foo
  CREATE TABLE foo(col1 INT)
  CREATE TABLE foo
    (col1 INT PRIMARY KEY,
     col2 INT REFERENCES foo2(col1))
  CREATE TABLE foo2
    (col1 INT PRIMARY KEY,
     col2 INT REFERENCES foo(col1));
```

Синтаксис *ALTER SCHEMA* следующий:

```
ALTER SCHEMA имя_схемы WITH NAME = новое_имя_владельца;
```

ALTER SCHEMA позволяет лишь сменить владельца схемы.

См. также

CREATE/ALTER TABLE
CREATE/ALTER VIEW
GRANT

CREATE/ALTER TABLE

Управление таблицами является самой распространенной задачей, выполняемой администраторами и разработчиками при работе с объектами базы данных. В этом разделе рассказывается, как создавать и изменять таблицы.

Стандарт ANSI представляет в некотором роде наименьший общий знаменатель среди всех реализаций. Хотя не все производители полностью поддерживают стандартные версии операторов *CREATE TABLE* и *ALTER TABLE*, стандарт ANSI предлагает базовый формат, который можно использовать на всех платформах. Производители же наоборот предлагают множество расширений и дополнений к операторам *CREATE* и *ALTER TABLE*.

Обычно при проектировании и создании таблиц требуется принимать во внимание множество различных соображений. Этот алгоритм называется *проектированием баз данных*. Процесс анализа отношения таблицы к ее данным и к другим таблицам называется *нормализацией*. Мы рекомендуем разработчикам и администраторам внимательно изучить основы проектирования баз данных и принципы нормализации, прежде чем браться за оператор *CREATE TABLE*.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

Согласно SQL2003 оператор *CREATE TABLE* используется для создания в базе данных постоянных или временных таблиц. Используется следующий синтаксис:

```
CREATE [{LOCAL TEMPORARY| GLOBAL TEMPORARY}] TABLE
имя_таблицы
    (имя_столбца тип_данных атрибуты[, ...]) |
    [имя_столбца WITH OPTIONS опции] |
    [LIKE имя_столбца] |
    [REF IS имя_столбца
```

```
        {SYSTEM GENERATED | USER GENERATED | DERIVED}]
      [CONSTRAINT тип_ограничения [имя_ограничения][, ...]]
  [OF имя_типа [UNDER супертаблица] [определение_таблицы]]
  [ON COMMIT {PRESERVE ROWS | DELETE ROWS}
```

Оператор *ALTER TABLE* позволяет модифицировать таблицу без удаления существующих индексов, триггеров и разрешений. Для оператора *ALTER TABLE* используется следующий синтаксис:

```
ALTER TABLE имя_таблицы
[ADD [COLUMN] имя_столбца тип_данных атрибуты]
| [ALTER [COLUMN] имя_столбца SET DEFAULT значение]
| [ALTER [COLUMN] имя_столбца DROP DEFAULT]
| [ALTER [COLUMN] имя_столбца ADD SCOPE имя_таблицы]
| [ALTER [COLUMN] имя_столбца DROP
      SCOPE {RESTRICT | CASCADE}]
| [DROP [COLUMN] имя_столбца {RESTRICT | CASCADE}]
| [ADD ограничение]
| [DROP CONSTRAINT имя_ограничения {RESTRICT | CASCADE}]
```

Ключевые слова

CREATE [{LOCAL TEMPORARY|GLOBAL TEMPORARY}] TABLE

Создает постоянную или временную (*TEMPORARY*) таблицу. Временная таблица может быть локальной (*LOCAL*) или глобальной (*GLOBAL*). *Локальные временные таблицы* доступны только создавшей их сессии и автоматически удаляются после завершения этой сессии. *Глобальные временные таблицы* доступны всем активным сессиям, но также удаляются после завершения сессии, создавшей временную таблицу. Не добавляйте к имени глобальной временной таблицы имя схемы.

имя_столбца тип_данных атрибуты[, ...]

Один или несколько столбцов (в списке через запятую) с указанием их типов данных и дополнительных атрибутов. Для создания таблицы необходимо объявить как минимум один столбец, указав следующие параметры:

имя_столбца

Определяет имя столбца. Имя должно являться допустимым идентификатором. Убедитесь, что имя столбца осмысленно и понятно.

тип_данных

Определяет тип данных столбца. Для некоторых типов можно также дополнительно определить длину значения, например *VARCHAR(25)*. Тип данных должен быть допустим в используемой СУБД. Полное описание допустимых типов данных и их реализаций приводится в главе 2.

атрибуты

Определяет для столбца дополнительные атрибуты. При указании нескольких атрибутов запятые между ними не нужны. В ANSI описаны следующие основные атрибуты:

NOT NULL

Указывает, что столбец не допускает значения NULL (или допускает, если атрибут не указан). Любой оператор *INSERT* или *UPDATE*, который по-

пытается поместить в столбец значение NULL, завершится с ошибкой и будет откачен.

DEFAULT выражение

Определяет выражение, используемое для столбца в качестве значения по умолчанию в тех случаях, когда операторы UPDATE или INSERT не устанавливают значение явно. Выражение должно быть совместимо с используемым типом данных: например, для столбца типа INTEGER нельзя использовать алфавитные символы. Выражение может быть строковым или числовым литералом, но может также быть и пользовательской или системной функцией. SQL2003 допускает использование в качестве значения по умолчанию следующих системных функций: *NULL*, *USER*, *CURRENT_USER*, *SESSION_USER*, *SYSTEM_USER*, *CURRENT_PATH*, *CURRENT_DATE*, *CURRENT_TIME*, *LOCALTIME*, *CURRENT_TIMESTAMP*, *LOCALTIMESTAMP*, *ARRAY* и *ARRAY[ ]*.

COLLATE схема_упорядочения

Определяет для значения столбца схему упорядочения (то есть порядок сортировки). Названия схем упорядочивания зависят от платформы. По умолчанию будет использоваться стандартная схема упорядочения набора символов столбца.

REFERENCES ARE [NOT] CHECKED [ON DELETE {RESTRICT | SET NULL}]

Указывает, что необходимо проверять ссылки для столбцов, объявленных с атрибутом *REF*. Фраза *ON DELETE* определяет поведение при удалении записи, на которую ссылаются другие записи: такое удаление можно запретить (*RESTRICT*) либо проставить в ссылающихся записях значение NULL (*SET NULL*).

CONSTRAINT имя_ограничения [тип_ограничения [ограничение]]

Накладывает на столбец ограничение. Ограничения детально описываются в главе 2. Так как ограничение создается на уровне столбца, то в ограничении может использоваться только один этот столбец. После создания таблицы ограничение используется как ограничение уровня таблицы.

имя_столбца [WITH OPTIONS опции]

Определяет столбец с дополнительными опциями, такими как область видимости, значение по умолчанию, ограничение, схема упорядочения. В большинстве реализаций использование выражения *WITH OPTIONS* ограничено созданием типизированных таблиц.

LIKE имя_таблицы

Создает новую таблицу с таким же набором столбцов, что и указанная существующая таблица.

REF IS имя_столбца {SYSTEM GENERATED | USER GENERATED | DERIVED}

Объявляет для типизированной таблицы столбец, являющийся объектным идентификатором (*OID*). *OID* необходим для корневой таблицы в иерархии таблиц. В зависимости от указанной опции идентификатор может автоматически генерироваться системой (*SYSTEM GENERATED*), указываться пользователем при вставке записи (*USER GENERATED*) или получаться из друго-

го идентификатора (*DERIVED*). Также требуется, чтобы для указанного столбца был указан атрибут *REFERENCES*.

CONSTRAINT тип_ограничения [имя_ограничения][, ...]

Создает для таблицы одно или несколько ограничений. Эта опция отличается от *CONSTRAINT* для столбца, так как ограничение уровня столбца относится только к соответствующему столбцу. Ограничение уровня таблицы позволяет использовать в ограничении несколько столбцов. Например, в таблице **sales** вы могли бы создать уникальный ключ на наборе столбцов **store_id**, **order_id** и **order_date**. Это можно сделать только с помощью ограничения на уровне таблицы. Ограничения обсуждаются в главе 2.

OF имя_типа [UNDER супертаблица] [определение_таблицы]

Указывает, что таблица базируется на существующем пользовательском типе данных. В этом случае в таблице может быть один столбец на каждый атрибут типа данных и дополнительный столбец, объявленный с опцией *REF IS*. Эта фраза несовместима с фразой *LIKE*.

[UNDER супертаблица] [определение_таблицы]

Указывает для создаваемой таблицы непосредственную супертаблицу, созданную в этой же схеме. При этом создаваемая таблица становится субтаблицей. По желанию вы можете указать полное определение новой субтаблицы, со столбцами, ограничениями и т. д.

ON COMMIT {PRESERVE ROWS | DELETE ROWS}

ON COMMIT PRESERVE ROWS оставляет записи во временной таблице при фиксации транзакции. *ON COMMIT DELETE ROWS* при COMMIT удаляет все записи из временной таблицы.

ADD [COLUMN] имя_столбца тип_данных атрибуты

Добавляет в таблицу столбец с указанным именем, типом данных и атрибутами.

ALTER [COLUMN] имя_столбца SET DEFAULT значение

Устанавливает или изменяет для столбца значение, используемое по умолчанию.

ALTER [COLUMN] имя_столбца DROP DEFAULT

Удаляет для столбца значение по умолчанию.

ALTER [COLUMN] имя_столбца ADD SCOPE имя_таблицы

Устанавливает область действия для указанного столбца. Область действия – это ссылка на пользовательский тип данных.

ALTER [COLUMN] имя_столбца DROP SCOPE {RESTRICT | CASCADE}

Удаляет область действия указанного столбца. В конце можно указать *RESTRICT* или *CASCADE*.

DROP COLUMN имя_столбца {RESTRICT | CASCADE}

Удаляет из таблицы указанный столбец.

ADD ограничение

Добавляет в таблицу новое ограничение с указанным именем и характеристиками.

DROP CONSTRAINT имя_ограничения {RESTRICT | CASCADE}

Удаляет созданное ранее ограничение.

RESTRICT

Прерывает операцию в случае обнаружения зависимых объектов.

CASCADE

Удаляет все остальные объекты, зависимые от удаляемого объекта.

Общие правила

Типичный оператор *CREATE TABLE* очень прост. Обычно он просто именует таблицу и столбцы таблицы. Часто определения таблицы также содержат ограничения на пустые значения, как в следующем примере для SQL Server:

```
CREATE TABLE housing_construction
(project_number     INT         NOT NULL,
project_date        DATE        NOT NULL,
project_name        VARCHAR(50) NOT NULL,
construction_color  NCHAR(20)   ,
construction_height DECIMAL(4,1),
construction_length DECIMAL(4,1),
construction_width  DECIMAL(4,1),
construction_volume INT         )
```

В этом примере демонстрируется создание внешнего ключа:

```
-- Создаем ограничение на уровне столбца
CREATE TABLE favorite_books
   (isbn         CHAR(100)   PRIMARY KEY,
   book_name     VARCHAR(40) UNIQUE,
   category      VARCHAR(40) ,
   subcategory   VARCHAR(40) ,
   pub_date      DATETIME    NOT NULL,
   purchase_date DATETIME    NOT NULL,
      CONSTRAINT fk_categories FOREIGN KEY (category)
        REFERENCES category(cat_name));
```

Внешний ключ по столбцу **categories** ссылается на столбец **cat_name** таблицы **category**. Все рассматриваемые в книге платформы поддерживают этот синтаксис.

Примеры создания различных ограничений приводятся в главе 2.

Аналогичным образом может быть создан составной внешний ключ, содержащий столбцы **category** и **subcategory**:

```
ALTER TABLE favorite_books ADD CONSTRAINT fk_categories
  FOREIGN KEY (category, subcategory)
  REFERENCES category(cat_name, subcat_name);
```

Теперь мы можем использовать *ALTER TABLE* для удаления ограничения:

```
ALTER TABLE favorite_books
  DROP CONSTRAINT fk_categories RESTRICT;
```

Далее приводится фрагмент из базы данных **pubs**, поставляемой в качестве примера для Microsoft SQL Server и Sybase Adaptive Server:

```
-- Для Microsoft SQL Server
CREATE TABLE jobs
  (job_id  SMALLINT IDENTITY(1,1) PRIMARY KEY CLUSTERED,
  job_desc VARCHAR(50) NOT NULL DEFAULT 'New Position',
  min_lvl  TINYINT NOT NULL CHECK (min_lvl >= 10),
  max_lvl  TINYINT NOT NULL CHECK (max_lvl <= 250))
-- Для MySQL
CREATE TABLE employee
  (emp_id INT AUTO_INCREMENT
    CONSTRAINT PK_emp_id PRIMARY KEY,
  fname VARCHAR(20) NOT NULL,
  minit CHAR(1) NULL,
  lname VARCHAR(30) NOT NULL,
  job_id SMALLINT NOT NULL DEFAULT 1
    REFERENCES jobs(job_id),
  job_lvl TINYINT DEFAULT 10,
  pub_id CHAR(4) NOT NULL DEFAULT ('9952')
    REFERENCES publishers(pub_id),
  hire_date DATETIME NOT NULL DEFAULT (CURRENT_DATE( ));
CREATE TABLE publishers
  (pub_id char(4) NOT NULL
    CONSTRAINT UPKCL_pubind PRIMARY KEY CLUSTERED
    CHECK (pub_id IN ('1389', '0736', '0877', '1622',
      '1756') OR pub_id LIKE '99[0-9][0-9]'),
  pub_name varchar(40) NULL,
  city varchar(20) NULL,
  state char(2) NULL,
  country varchar(30) NULL DEFAULT('USA'))
```

Как только вы начинаете использовать специфические расширения, оператор *CREATE TABLE* перестает быть переносимым между платформами. Вот, например, оператор *CREATE TABLE* для Oracle, с большим набором параметров хранения:

```
CREATE TABLE classical_music_cds
  (music_id        INT,
  composition      VARCHAR2(50),
  composer         VARCHAR2(50),
  performer        VARCHAR2(50),
  performance_date DATE DEFAULT SYSDATE,
  duration         INT,
  cd_name          VARCHAR2(100),
CONSTRAINT pk_class_cds PRIMARY KEY (music_id)
  USING INDEX TABLESPACE index_ts
  STORAGE (INITIAL 100K NEXT 20K),
CONSTRAINT uq_class_cds UNIQUE
  (composition, performer, performance_date)
  USING INDEX TABLESPACE index_ts
  STORAGE (INITIAL 100K NEXT 20K))
TABLESPACE tabledata_ts;
```

Мы рекомендуем, чтобы оператор *ALTER* или *CREATE* был единственным в транзакции. Например, не пытайтесь создать таблицу, а затем сразу же выбрать из нее данные. Вместо этого создайте таблицу, убедитесь, что все сделано верно, выполните *COMMIT* и только затем выполняйте последующие операции.

Имена таблиц должны начинаться с буквы и в общем случае не должны содержать специальных символов, за исключением подчеркивания. Правила, касающиеся длины и содержимого имени таблицы, отличаются для разных платформ.

При создании или изменении таблицы список столбцов следует указывать в скобках, разделяя столбцы запятыми.

Советы и хитрости

Для выполнения команды *CREATE TABLE* пользователь должен иметь соответствующие привилегии. Аналогично, любой пользователь, который хочет изменить или удалить таблицу, может сделать это либо при наличии соответствующих привилегий, либо, если таблица содержится в его собственной схеме. Так как стандарт ANSI не определяет точный набор привилегий, то существуют некоторые отличия среди производителей.

Операторы *CREATE TABLE* и *ALTER TABLE* можно инкапсулировать в транзакцию, используя операторы *COMMIT* или *ROLLBACK* для явного завершения транзакции. Мы рекомендуем, чтобы *CREATE/ALTER TABLE* был единственным оператором в транзакции.

Расширения стандарта ANSI позволяют вам контролировать физический порядок строк на диске. В SQL Server для этого используются *кластерные индексы*. В Oracle есть функционально аналогичная структура под названием *индексно-организованная таблица*.

В некоторых СУБД модифицируемая оператором *ALTER TABLE* таблица блокируется. Поэтому можно выполнять этот оператор только для тех таблиц, которые не используются в данный момент. Более того, некоторые СУБД при выполнении оператора *CREATE TABLE ... LIKE* блокируют и создаваемую таблицу, и таблицу-источник.

MySQL

В MySQL оператор *CREATE TABLE* создает обычную или временную таблицу в той базе данных, в которой он выполняется:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] имя_таблицы
{( имя_столбца тип_данных атрибуты
  тип_ограничения имя_ограничения [, ...])
 [тип_ограничения [имя_ограничения][, ...]]
    [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
    [ON {DELETE | UPDATE} {RESTRICT | CASCADE | SET NULL |
       NO ACTION}]
LIKE имя_таблицы }
 {[TABLESPACE имя_табличного_пространства STORAGE DISK] |
  [ENGINE = {ISAM | MyISAM | HEAP | BDB | InnoDB
            | MERGE | MRG_MyISAM}] |
  [AUTO_INCREMENT = целое_число] |
```

```
  [AVG_ROW_LENGTH = целое_число] |
  [ [DEFAULT] CHARACTER SET кодовая_страница ] |
  [CHECKSUM = {0 | 1}] |
  [ [DEFAULT] COLLATE схема_упорядочения ] |
  [COMMENT = "строка"] |
  [CONNECTIOIN = 'строка_подключения'] |
  [DATA DIRECTORY = "путь_к_каталогу"] |
  [DELAY_KEY_WRITE = {0 | 1}] |
  [INDEX DIRECTORY = " путь_к_каталогу "] |
  [INSERT_METHOD = {NO | FIRST | LAST}] |
  [KEY_BLOCK_SIZE = целое_число] |
  [MAX_ROWS = целое_число] |
  [MIN_ROWS = целое_число] |
  [PACK_KEYS = {0 | 1}] |
  [PASSWORD = "строка"] |
  [ROW_FORMAT= { DEFAULT | DYNAMIC | FIXED
               | COMPRESSED | REDUNDANT | COMPACT }] [...]}
[определение_секций[, ...]]
[ [IGNORE | REPLACE] оператор_select ]
```

Обычно MySQL для изменения таблицы делает временную копию, работает с ней, удаляет исходную таблицу и переименовывает копию. Синтаксис *ALTER TABLE* позволяет модифицировать или переименовывать таблицу:

```
ALTER [IGNORE] TABLE имя_таблицы
{ [ADD [COLUMN] (имя_столбца тип_данных атрибуты)
    [FIRST | AFTER имя_столбца][, ...]]
| [ADD [CONSTRAINT] [UNIQUE | FOREIGN KEY | FULLTEXT |
    PRIMARY KEY | SPATIAL] [INDEX | KEY] [имя_индекса]
    ( имя_столбца_индекса [, ...])]
| [ALTER [COLUMN] имя_столбца
    {SET DEFAULT литерал | DROP DEFAULT}]
| [CHANGE | MODIFY] [COLUMN] старое_имя_столбца
    новое_имя_столбца определение_столбца [FIRST | AFTER
    имя_столбца]
| [DROP [COLUMN | FOREIGN KEY | PRIMARY KEY | INDEX | KEY]
    [имя_объекта]]
| [{ENABLE | DISABLE} KEYS]
| [RENAME [TO] новое_имя_таблицы]
| [ORDER BY имя_столбца [, ...]]
| [CONVERT TO CHARACTER SET кодовая_страница
    [COLLATE схема_упорядочения]]
| [{DISCARD | IMPORT} TABLESPACE]
| [{ADD | DROP | COALESCE целоу_число | ANALYZE | CHECK |
    OPTIMIZE | REBUILD | REPAIR} PARTITION]
| [REORGANIZE PARTITION имя_секции INTO(определение_секции)]
| [REMOVE PARTITIONING]
| [опции_таблицы] }[, ...]
```

Параметры и ключевые слова имеют следующие значения:

TEMPORARY

Создает таблицу, существующую только в течение сессии, в которой создается таблица. После закрытия сессии таблица автоматически удаляется.

IF NOT EXISTS

Предотвращает появление сообщения об ошибке, если таблица уже существует. Указание схемы не требуется.

тип_ограничения

Позволяет создавать стандартные ANSI SQL-ограничения на уровне столбца или таблицы. MySQL полностью поддерживает следующие ограничения: первичный ключ, уникальный ключ и значение по умолчанию (должно быть константой). MySQL поддерживает на уровне синтаксиса проверочные ограничения и внешние ключи, но функционально они работают только на таблицах InnoDB. MySQL также поддерживает 6 специальных ограничений:

FULLTEXT [{INDEX | KEY}]

Создает полнотекстовый индекс для быстрого поиска по большим объемам текстовых данных. Помните, что этот тип индекса поддерживается только для таблиц MyISAM и для столбцов типа *CHAR*, *VARCHAR* и *TEXT*.

SPATIAL [{INDEX | KEY}]

Создает пространственный индекс на столбец. Поддерживается только таблицами MyISAM.

AUTO_INCREMENT

Настраивает столбец таким образом, что его значения автоматически увеличиваются на 1 (начальное значения также 1). MySQL допускает по одному столбцу *AUTO_INCREMENT* на таблицу. При удалении из таблицы всех записей (*DELETE* или *TRUNCATE*) значение автоинкремента может быть сброшено. Опция поддерживается таблицами MyISAM, MEMORY, ARCHIVE и InnoDB.

[UNIQUE] INDEX

При использовании для столбца ключевого слова *INDEX* можно по желанию указать также имя индекса. (В MySQL слово *KEY* используется как синоним *INDEX*.) Если для первичного ключа название не указано, то MySQL автоматически создает имя из названию столбца и числового суффикса (_2, _3,...). Все типы таблиц, кроме ISAM, поддерживают индексы по столбцам со значениями *NULL* и по столбцам типов *BLOB* и *TEXT*. Обратите внимание, что *UNIQUE* без *INDEX* также является допустимым синтаксисом в MySQL.

COLUMN_FORMAT {FIXED | DYNAMIC | DEFAULT}

Определяет формат хранения столбцов в таблицах NDB. *FIXED* устанавливает хранение с фиксированной шириной; *DYNAMIC* – с переменной шириной; *DEFAULT* означает, что формат хранения будет выбран в соответствии с типом данных столбца. Эта фраза недоступна в версиях младше 5.1.19-ndb.

STORAGE {DISK | MEMORY}

Определяет, где хранится столбец таблицы NDB – на диске (*DISK*) или, по умолчанию, в памяти (*MEMORY*). Эта фраза недоступна в версиях младше 5.1.19-ndb.

ENGINE

Определяет механизм (engine), используемый для хранения данных. Вы можете менять способ хранения с помощью оператора *ALTER TABLE*. Только таблицы типа InnoDB и BDB восстанавливаемы при сбоях и поддерживают *COMMIT* и *ROLLBACK*. Таблицы других типов при сбоях подвержены потерям данных, но при этом работают значительно быстрее и занимают меньше места. По умолчанию используется MyISAM. Список возможных типов таблиц следующий:

ARCHIVE

Использует механизм хранения *ARCHIVE*, подходящий для хранения больших объемов данных, без индексов, с минимальным занимаемым дисковым пространством. Для таблицы типа *ARCHIVE* создается файл с названием, совпадающим с названием таблицы, и расширением *.FRM*. Данные и метаданные хранятся в одноименных файлах, но с расширениями *.ARX* и *.ARM* соответственно. Иногда в процессе оптимизации может появляться файл с расширением *.ARN*.

CSV

Хранит данные таблицы в текстовом виде, значения в файле разделяются запятыми (comma-separated values). Для таблицы типа *CSV* создается файл с названием, совпадающим с названием таблицы, и расширением *.FRM*. Данные хранятся в файле с таким же именем, но расширением *.CSV*. Данные хранятся в текстовом виде, поэтому следует обратить внимание на вопрос безопасности данных.

EXAMPLE

Механизм хранения *EXAMPLE* является просто заглушкой, которая ничего не делает. В таблицах *EXAMPLE* данные хранить нельзя.

FEDERATED

Хранит данные таким образом, чтобы они были доступны для удаленного доступа из других баз MySQL без использования механизмов кластеризации и репликации. В локальных таблицах при этом данные не хранятся.

HEAP

Создает таблицу, постоянно хранящуюся в памяти и использующую индекс по хеш-значению. Синоним для *MEMORY*. Так как таблица хранится в памяти, то все ее данные теряются в случае сбоя сервера. Рассматривайте таблицы *HEAP* как альтернативу временным таблицам. При использовании таблиц *HEAP* устанавливайте для них свойство *MAX_ROWS*, чтобы они не использовали всю свободную память. Такие таблицы не поддерживают столбцы типа *BLOB* и *TEXT*, опцию *AUTO_INCREMENT*, фразу *ORDER BY* и переменную длину записи.

InnoDB

Создает транзакционную таблицу с блокировками на уровне строк. Также поддерживает управление независимыми табличными пространствами, контрольные точки, неблокирующее чтение и быстрое чтение из больших файлов данных для повышенного параллелизма и производительности. Требует установки параметр *innodb_data_file_path*. InnoDB поддерживает все ограничения, описанные в ANSI, включая *CHECK* и *FOREIGN KEY*.

Известные сайты разработчиков, такие как Slashdot.org, работают на MySQL с InnoDB из-за его отличной производительности. Таблицы и индексы InnoDB хранятся вместе в отдельных табличных пространствах (в отличие от других форматов хранения, например MyISAM, когда таблицы хранятся в отдельных файлах).

ISAM

Создает таблицу в старом формате ISAM. Этот формат устарел и недоступен в MySQL 5.1. При переходе на новую версию MySQL предварительно сконвертируйте таблицы ISAM в MyISAM.

MEMORY

Создает таблицу, постоянно хранящуюся в памяти и использующую индекс по хеш-значению. Синоним для *HEAP*. Так как таблица хранится в памяти, то все ее данные теряются в случае сбоя сервера. Таблицы *MEMORY* могут иметь до 32 индексов на одну таблицу и до 16 столбцов на один индекс, общая длина ключа индекса не должна превышать 500 байт. Рассматривайте таблицы *MEMORY* как альтернативу временным таблицам; как и временные таблицы, они используются пользователями совместно. При использовании таблиц *MEMORY* устанавливайте для них свойство *MAX_ROWS*, чтобы они не использовали всю свободную память. Также такие таблицы не поддерживают столбцы типа *BLOB* и *TEXT*, опцию *AUTO_INCREMENT*, фразу *ORDER BY* и переменную длину записи. Есть еще много правил, касающихся таблиц *MEMORY*, поэтому перед использованием этого формата хранения обязательно прочитайте документацию.

MERGE

Использует несколько одинаково структурированных MyISAM таблиц как одну таблицу, предоставляя некоторые преимущества секционированных таблиц. Операторы *SELECT*, *UPDATE* и *INSERT* работают с этим набором таблиц как с одной таблицей. Рассматривайте таблицу *MERGE* как коллекцию, а не как таблицу с данными. Удаление таблицы *MERGE* только исключает таблицу из коллекции, сами данные не удаляются. Таблица *MERGE* создается при помощи оператора *UNION=(table1, table2, ...)*. Ключевые слова *MERGE* и *MRG_MyISAM* используются как синонимы.

MyISAM

Хранит данные в файлах *.MYD*, а индексы в файлах *.MYI*. MyISAM основан на коде ISAM с некоторыми расширениями. В MyISAM используется двоичная структура хранения, более переносимая, чем ISAM. MyISAM поддерживает *AUTO_INCREMENT*, сжатие таблиц (при помощи утилиты myisampack) и таблицы большого размера, индексирование столбцов типа *BLOB* и *TEXT*, до 64 индексов на таблицу, до 16 столбцов на индекс, и длину ключа до 1000 байт.

NDBCLUSTER

Создает кластеризованные таблицы, устойчивые к сбоям и хранимые в памяти, называемые NDB. Это специальный формат с максимальной доступностью. За дополнительной информацией обращайтесь к документации.

UNION

Смотрите описание *MERGE*.

TABLESPACE...STORAGE DISK

При использовании кластеризованных NDB таблиц перемещает таблицу в указанное табличное пространство Cluster Disk Data. Табличное пространство должно быть предварительно создано с помощью *CREATE TABLESPACE*.

AUTO_INCREMENT = целое_число

Устанавливает для таблицы автоинкрементное значение (только для MyISAM).

AVG_ROW_LENGTH = целое_число

Устанавливает приблизительный средний размер строки для таблиц с переменной длиной строки. MySQL использует формулу *AVG_ROW_LENGTH * MAX_ROWS* для определения того, насколько большой может стать таблица.

[DEFAULT] CHARACTER SET

Устанавливает кодовую страницу, используемую для данных таблицы или отдельного столбца.

CHECKSUM = {0 | 1}

Если установлено значение 1, то для всех строк таблицы вычисляется контрольная сумма (только в MyISAM). Работа при этом замедляется, но данные становятся менее подверженными повреждению.

[DEFAULT] COLLATE

Устанавливает схему упорядочения для данных таблицы или отдельного столбца.

COMMENT = "строка"

Устанавливает комментарий длиной до 60 символов.

CONNECTION = 'строка_подключения'

Строка подключения требуется для таблиц *FEDERATED*, в прочих случаях она игнорируется. В старых версиях MySQL строка подключения указывалась в опции *COMMENT*.

DATA DIRECTORY = "путь_к_каталогу"

Устанавливает путь к каталогу для хранения таблиц MyISAM, который необходимо использовать вместо каталога по умолчанию.

DELAY_KEY_WRITE = {0 | 1}

Если установлено значение 1, то обновление ключей таблицы откладывается до закрытия таблицы (только MyISAM).

INDEX DIRECTORY = " путь_к_каталогу "

Устанавливает путь к каталогу для хранения индексов MyISAM, который необходимо использовать вместо каталога по умолчанию.

INSERT_METHOD = {NO | FIRST | LAST}

Используется для таблиц MERGE. Если никакое значение не установлено или установлено значение *NO*, то вставки в таблицу запрещены. Если установлено *FIRST*, то вставка производится в первую таблицу коллекции, при *LAST* вставка производится в последнюю таблицу коллекции.

KEY_BLOCK_SIZE = целое_число

Позволяет изменить размер блока индекса. 0 означает использование значения по умолчанию.

MAX_ROWS = целое_число

Устанавливает максимальное количество строк в таблице. По умолчанию используется значение, соответствующее 4 гигабайтам занимаемого места.

MIN_ROWS = целое_число

Устанавливает минимальное число строк в таблице.

PACK_KEYS = {0 | 1}

Если установлено значение 1, то выполняется сжатие индексов. При этом чтения становятся быстрее, а обновления – медленнее (только для ISAM и MyISAM). По умолчанию сжимаются только строковые значения. Если установлено значение 1, то сжимаются и строковые, и числовые значения.

PASSWORD = "строка"

Шифрует файл *.FRM* (но не саму таблицу) с использованием указанного пароля.

ROW_FORMAT = { DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT }

Определяет, как в таблице будут храниться строки. *DEFAULT* имеет разное значение в зависимости от механизма хранения. *DYNAMIC* используется для строк переменного размера (например, использующих *VARCHAR*), а *FIXED* – для строк фиксированной длины (*CHAR*, *INT* и т. д.). *REDUNDANT* используется только с таблицами InnoDB и улучшает работу индексов за счет избыточного хранении информации. Таблицы *COMPRESSED* используются только для чтения и занимают примерно на 20% меньше места по сравнению с *REDUNDANT*. *COMPRESSED* также допустим только для таблиц InnoDB.

определение_секции

Определяет секции и подсекции таблицы в MySQL. Описание секционирования в MySQL приводится в следующем разделе. Все описываемые опции используются и для подсекций, за исключением *VALUE*.

[IGNORE | REPLACE] оператор_select

Создает таблицу с набором столбцов, определяемым оператором *SELECT*. Если оператор возвращает строки, то они вставляются в создаваемую таблицу.

ALTER [IGNORE]

Изменяемая таблица будет включать все дубликаты, если только не указать ключевое слово *IGNORE*. Если ключевое слово не используется, то выполнение оператора завершится с ошибкой, если в существующем первичном или уникальном ключе будут обнаружены дубликаты.

{ADD | COLUMN} [FIRST | AFTER имя_столбца]

Добавляет или перемещает столбец, индекс или ключ. При добавлении столбца он становится последним по порядку, если только не задана его позиция с помощью ключевого слова *AFTER*.

ALTER COLUMN

Позволяет изменить определение столбца или его значение по умолчанию.

CHANGE

Переименовывает столбец или меняет его тип.

MODIFY

Меняет тип данных столбца или его атрибуты, например *NOT NULL*. Существующие данные автоматически конвертируются в новый тип данных.

DROP

Удаляет столбец, индекс, ключ или табличное пространство. Удаляемый столбец также удаляется из всех индексов, в которых он участвует. Если при удалении первичного ключа оказывается, что первичного ключа в таблице нет, то будет удален первый уникальный ключ.

{ENABLE | DISABLE} KEYS

Одновременно включает или выключает все неуникальные ключи таблицы MyISAM. Это может быть полезно для массовых загрузок, когда вы хотите временно отключить ограничения до окончания загрузки. Также это увеличивает производительность, так как все блоки индекса сбрасываются на диск в конце операции.

RENAME [TO] новое_имя_таблицы

Переименовывает таблицу.

ORDER BY имя_столбца[,...]

Сортирует строки в указанном порядке.

CONVERT TO CHARACTER SET кодовая_страница [COLLATE схема_упорядочения]

Конвертирует данные таблицы в новую кодировку и схему упорядочения.

DISCARD | IMPORT TABLESPACE

Удаляет текущий файл *.IDB* (*DISCARD*) или делает табличное пространство доступным после восстановления из резервной копии (*IMPORT*).

{ADD | DROP | COALESCE целое_число | ANALYZE | CHECK | OPTIMIZE | REBUILD | REPAIR} PARTITION

Создает или удаляет секцию таблицы. Другие опции используются для обслуживания секций и аналогичны соответствующим опциям для таблиц (например, *CHECK TABLE* или *REPAIR TABLE*). Только *COALESCE PARTITION* имеет уникальное поведение и используется для уменьшения числа *KEY* или *HASH* секций до заданного значения.

REORGANIZE PARTITION имя_секции INTO (определение_секции)

Изменяет определение указанной секции.

REMOVE PARTITIONING

Отключает секционирование таблицы без влияния на другие настройки таблицы и на данные.

Например:

```
CREATE TABLE test_example
  (column_a INT NOT NULL AUTO_INCREMENT,
   PRIMARY KEY(column_a),
   INDEX(column_b))
  TYPE=HEAP
IGNORE
SELECT column_b,column_c FROM samples;
```

В MySQL версии 5.1 и старше индексы по столбцам со значениями NULL поддерживаются только таблицами MyISAM, InnoDB и MEMORY. В других случаях индексируемый столбец должен быть объявлен как NOT NULL. Пространственные типы данных поддерживаются только таблицами MyISAM и для индексации также должны иметь атрибут NOT NULL. В MySQL 5.1 в таблице не может быть больше, чем 4096 столбцов, но даже это значение может не достигаться из-за ограничений на занимаемое пространство.

В этом примере мы создали таблицу типа *HEAP* со столбцами **column_a**, **column_b** и **column_c**. Позднее мы можем переконвертировать эту таблицу в формат MyISAM:

```
ALTER TABLE example TYPE=MyISAM;
```

При создании таблицы типа MyISAM в операционной системе создаются три файла: файл определения таблицы с расширением *.FRM*, файл данных с расширением *.MYD* и файл индекса с расширением *.MYI*. Файл *.FRM* используется для всех других типов таблиц.

В следующем примере создается две базовых таблицы типа MyISAM, а затем они объединяются в MERGE-таблицу:

```
CREATE TABLE message1
  (message_id    INT AUTO_INCREMENT PRIMARY KEY,
   message_text  CHAR(20));
CREATE TABLE message2
  (message_id    INT AUTO_INCREMENT PRIMARY KEY,
   message_text  CHAR(20));
CREATE TABLE all_messages
  (message_id    INT AUTO_INCREMENT PRIMARY KEY,
   message_text  CHAR(20))
   TYPE=MERGE UNION=(message1, message2) INSERT_METHOD=LAST;
```

Секционированные таблицы

В MySQL для лучшего контроля над операциями ввода-вывода и дисковым пространством поддерживается секционирование таблиц. Для секционирования используется следующий синтаксис:

```
PARTITION BY функция
[ [SUB]PARTITION имя_секции
  [VALUES {LESS THAN {(выражение) | MAXVALUE} |
    IN (список_значений)}]
  [[STORAGE] ENGINE [=] механизм_хранения]
```

```
[COMMENT [=] 'комментарий']
[DATA DIRECTORY [=] ' путь_к_каталогу ']
[INDEX DIRECTORY [=] 'путь_к_каталогу']
[MAX_ROWS [=] целое_число]
[MIN_ROWS [=]целое_число]
[TABLESPACE [=] (имя_табличного_пространства)]
[NODEGROUP [=] идентификатор]
[(подсекция [,подсекция] ...)][, ...]]
```

где (неописанные параметры имеют то же значение, что и для таблицы в целом, и рассмотрены в предыдущем разделе):

функция

Определяет функцию, используемую для секционирования. Допустимые значения включают: *HASH(выражение)*, секционирование выполняется на основании значений хеш-функции от выражения, построенного с использованием столбцов таблицы и вызовов функций, возвращающих единственное целочисленное значение; *LINEAR HASH(список_столбцов)* вычисляет более равномерно распределенное хеш-значение; *RANGE(выражение)*, секционирование выполняется по диапазону значений выражения, во фразе *VALUES* указывается, какой диапазон значений содержит каждая секция; *LIST(выражение)*, секционирование выполняется по списку значений, во фразе *VALUES* указывается, какой список значений соответствует каждой секции.

[SUB]PARTITION имя_секции

Определяет имя секции или подсекции.

VALUES {LESS THAN {(выражение)|MAXVALUE}|IN (список_значений)}

Определяет, какие значения попадают в секцию.

NODEGROUP [=] идентификатор

Делает секцию или подсекцию частью группы узлов с заданным идентификатором. Применяется только для таблиц NDB.

Имейте в виду, что все секции и подсекции должны использовать один и тот же механизм хранения.

В следующем примере создаются три таблицы с разными способами секционирования:

```
CREATE TABLE employee (emp_id INT, emp_fname VARCHAR(30), emp_lname VARCHAR(50))
  PARTITION BY HASH(emp_id);

CREATE TABLE inventory (prod_id INT, prod_name VARCHAR(30), location_code CHAR(5))
  PARTITION BY KEY(location_code)
  PARTITIONS 4;

CREATE TABLE inventory (prod_id INT, prod_name VARCHAR(30), location_code CHAR(5))
  PARTITION BY LINEAR KEY(location_code)
  PARTITIONS 5;
```

В следующих примерах демонстрируется секционирование по диапазону значений и по списку значений:

```
CREATE TABLE employee (
  emp_id    INT,
```

```
  emp_fname VARCHAR(30),
  emp_lname VARCHAR(50),
  hire_date DATE)
PARTITION BY RANGE(hire_date)
  (PARTITION prtn1 VALUES LESS THAN ('01-JAN-2004'),
   PARTITION prtn2 VALUES LESS THAN ('01-JAN-2006'),
   PARTITION prtn3 VALUES LESS THAN ('01-JAN-2008'),
   PARTITION prtn4 VALUES LESS THAN MAXVALUE);

CREATE TABLE inventory (
  prod_id       INT,
  prod_name     VARCHAR(30),
  location_code CHAR(5))
PARTITION BY LIST(prod_id)
  (PARTITION prtn0 VALUES IN (10, 50, 90, 130, 170, 210),
   PARTITION prtn1 VALUES IN (20, 60, 100, 140, 180, 220),
   PARTITION prtn2 VALUES IN (30, 70, 110, 150, 190, 230),
   PARTITION prtn3 VALUES IN (40, 80, 120, 160, 200, 240));
```

В следующем примере переименовывается таблица и отдельный столбец:

```
ALTER TABLE employee RENAME AS emp;
ALTER TABLE employee CHANGE employee_ssn emp_ssn INTEGER;
```

Так как в MySQL существует возможность построения индекса по части столбца (например, по первым 10 символам строки), вы можете строить короткие индексы по большим столбцам.

MySQL поддерживает изменение типа данных столбца. Но чтобы не потерять данные, новый тип должен быть совместим со старым типом. Например, столбец с датами можно преобразовать в символьный тип данных, но символьный тип нельзя преобразовать в целочисленный. Вот пример изменения типа данных:

```
ALTER TABLE mytable MODIFY mycolumn LONGTEXT
```

MySQL обеспечивает некоторую гибкость оператора *ALTER TABLE*, позволяя указывать через запятую несколько фраз *ADD*, *ALTER*, *DROP* и *CHANGE*. Однако имейте в виду, что *CHANGE имя_столбца* и *DROP INDEX* являются расширениями MySQL и не описаны в SQL2003. MySQL поддерживает фразу *MODIFY*, подражая аналогичной функциональности в Oracle.

Oracle

Оператор *CREATE TABLE* в Oracle создает реляционную таблицу по заданному описанию либо по существующей таблице. После создания таблицы ее можно модифицировать с помощью оператора *ALTER TABLE*. В Oracle также можно создавать таблицы со столбцами, использующими пользовательские типы данных, объектные таблицы, явно создаваемые для хранения объектов определенного пользовательского типа (обычно *VARRAY* или *NESTED TABLE*), и таблицы типа *XMLType*.

В Oracle поддерживается стандартный оператор в стиле ANSI, но также добавлено много сложных расширений. Например, можно в значительной степени контролировать способ хранения и производительность таблицы. В *CREATE TABLE* и *ALTER TABLE* используется много вложенных и повторно используемых

фраз. Для упрощения понимания синтаксиса мы разбили описание синтаксиса оператора *CREATE TABLE* на три раздела: реляционные таблицы, объектные таблицы и XML-таблицы.

Синтаксис оператора *CREATE TABLE* для обычной реляционной таблицы (без объектной и XML функциональности) следующий:

```
CREATE [GLOBAL] [TEMPORARY] TABLE имя_таблицы
[ ( {столбец | виртуальный_столбец | атрибут} [SORT] [DEFAULT выражение]
[ {ограничение_столбца | ссылочное_ограничение_столбца}] |
  {ограничение_таблицы | ссылочное_ограничение_таблицы} |
  {GROUP журнальная_группа (столбец [NO LOG][, ...])
  [ALWAYS] | DATA (ограничения[, ...]) COLUMNS} ) ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[ограничение_таблицы]
{ [физические_атрибуты] [TABLESPACE табличное_пространство]
[атрибуты_хранения] [[NO]LOGGING] |
[CLUSTER (столбец[, ...])] |
{[ORGANIZATION
   {HEAP [физические_атрибуты]
       [TABLESPACE табличное_пространство]
       [атрибуты_хранения]
       [COMPRESS | NOCOMPRESS] [[NO]LOGGING] |
    INDEX [физические_атрибуты]
      [TABLESPACE табличное_пространство][атрибуты_хранения]
      [PCTTHRESHOLD целое_число]
      [COMPRESS [целое_число] | NOCOMPRESS]
      [MAPPING TABLE | NOMAPPING][...] [[NO]LOGGING]
      [[INCLUDING столбец] OVERFLOW
      [физические_атрибуты]
      [TABLESPACE табличное_пространство]
      [атрибуты_хранения] [[NO]LOGGING]}] |
    EXTERNAL ( [TYPE тип_драйвера] )
      DEFAULT DIRECTORY имя_каталога
      [ACCESS PARAMETERS {USING CLOB подзапрос|
        ( непрозрачный_формат )}]
      LOCATION ( [имя_каталога:]'спецификация'[, ...] )
      [REJECT LIMIT { целое_число | UNLIMITED}]} }
[{ENABLE | DISABLE} ROW MOVEMENT]
[[NO]CACHE] [[NO]MONITORING] [[NO]ROWDEPENDENCIES] [[NO]FLASHBACK ARCHIVE]
[PARALLEL целое_число | NOPARALLEL] [NOSORT] [[NO]LOGGING]]
[COMPRESS [целое_число] | NOCOMPRESS]
[{ENABLE | DISABLE} [[NO]VALIDATE]
  {UNIQUE (столбец [, ...]) | PRIMARY KEY | CONSTRAINT
     имя_ограничения} ]
  [USING INDEX {имя_индекса | CREATE_INDEX_оператор}]
  [EXCEPTIONS INTO] [CASCADE] [{KEEP | DROP} INDEX]] |
[секционирование]
[AS подзапрос]
```

Синтаксис создания реляционной таблицы содержит множество необязательных фраз. Но определение таблиц должно как минимум содержать либо имена и спецификации столбцов, либо фразу *AS подзапрос*.

Синтаксис для создания объектной таблицы следующий:

```
CREATE [GLOBAL] [TEMPORARY] TABLE имя_таблицы
OF объектный_тип[[NOT] SUBSTITUTABLE AT ALL LEVELS]
[ ( {столбец | атрибут} [DEFAULT выражение]
[ {ограничение_столбца | ссылочное_ограничение_столбца}] |
  {ограничение_таблицы | ссылочное_ограничение_таблицы} |
  {GROUP журнальная_группа (столбец [NO LOG][, ...])
  [ALWAYS] | DATA (ограничения [, ...]) COLUMNS} ) ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}]
[OIDINDEX [имя_индекса] ([физические_атрибуты]
    [атрибуты_хранения])]
[физические_атрибуты] [TABLESPACE табличное_пространство] [атрибуты_хранения]
```

Oracle позволяет создавать и изменять таблицы типа XMLType. Таблицы XMLType могут иметь обычные или виртуальные столбцы. Синтаксис для создания таблиц XMLType следующий:

```
CREATE [GLOBAL] [TEMPORARY] TABLE имя_таблицы
OF XMLTYPE
[ ( { столбец | атрибут } [DEFAULT выражение] [{ограничение_столбца |
ссылочное_ограничение_столбца}] |
{ ограничение_таблицы | ссылочное_ограничение таблицы} |
{GROUP журнальная_группа (столбец [NO LOG][, ...]) [ALWAYS] | DATA
(ограничения [, ...]) COLUMNS} ) ]
[XMLTYPE {OBJECT RELATIONAL [атрибуты_хранения_xml] |
[ {SECUREFILE | BASICFILE} ]
[ {CLOB | BINARY XML} [имя_lob_сегмента] [параметры_lob]}] [спецификация_xml_схемы]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}]
[OIDINDEX имя_индекса ([физические_атрибуты] [атрибуты_хранения])]
[физические_атрибуты] [TABLESPACE табличное_пространство] [атрибуты_хранения]
```

Оператор *ALTER TABLE* в Oracle позволяет изменять свойства таблицы и столбцов, параметры хранения, свойства *VARRAY* и *LOB*, параметры секционирования, ограничения целостности, связанные с таблицей и столбцами. Оператор позволяет выполнять и другие задачи, такие как перемещение таблицы в другое табличное пространство, освобождение незанятого пространства, уплотнение сегментов таблицы и сбрасывание маркера максимального уровня заполнения.

В ANSI SQL для изменения существующих элементов таблицы используется ключевое слово *ALTER*, а в Oracle для тех же целей используется *MODIFY*. Поэтому если фразы отличаются только этим ключевым словом (например, *ALTER TABLE...ALTER COLUMN* в ANSI и *ALTER TABLE...MODIFY COLUMN* в Oracle), то считайте их идентичными.

Для *ALTER TABLE* используется следующий синтаксис:

```
ALTER TABLE имя_таблицы
-- Изменение характеристик таблицы
   [физические_атрибуты] [атрибуты_хранения]
   [ {READ ONLY | READ WRITE} ]
   [[NO]LOGGING] [[NO]CACHE] [[NO]MONITORING] [[NO]COMPRESS]
   [[NO]FLASHBACK ARCHIVE]
```

```
   [SHRINK SPACE [COMPACT] [CASCADE]]
   [UPGRADE [[NOT] INCLUDING DATA]
      столбец тип_данных атрибуты]
   [[NO]MINIMIZE RECORDS_PER_BLOCK]
   [PARALLEL целое_число | NOPARALLEL]
   [{ENABLE | DISABLE} ROW MOVEMENT]
   [{ADD | DROP} SUPPLEMENTAL LOG
     {GROUP журнальная_группа [(столбец [NO LOG][, ...])
     [ALWAYS]] | DATA ( {ALL | PRIMARY KEY | UNIQUE |
     FOREIGN KEY}[, ...] ) COLUMNS}]
   [ALLOCATE EXTENT
     [( [SIZE целое_число [K | M | G | T]]
     [DATAFILE 'имя_файла'] [INSTANCE целое_число] )]
   [DEALLOCATE UNUSED [KEEP целое_число  [K | M | G | T]]]
   [ORGANIZATION INDEX ...
     [COALESCE] [MAPPING TABLE | NOMAPPING]
     [PCTTHRESHOLD целое_число]
     [COMPRESS целое_число | NOCOMPRESS]
     [ { ADD OVERFLOW [TABLESPACE табличное_пространство]
       [[NO]LOGGING] [физические_атрибуты] } |
       OVERFLOW { [ALLOCATE EXTENT (
       [SIZE целое_число [K | M | G | T]]
       [DATAFILE'имя_файла'] [INSTANCE целое_число ] ) |
       [DEALLOCATE UNUSED
         [KEEP целое_число  [K | M | G | T]]]] } ]]]
   [RENAME TO новое_имя_таблицы]

-- Изменение характеристик столбца
   [ADD (имя_столбца тип_данных атрибуты [, ...])]
   [DROP { {UNUSED COLUMNS | COLUMNS CONTINUE}
     [CHECKPOINT целое_число] | {COLUMN имя_столбца |
     (имя_столбца [, ...])} [CHECKPOINT целое_число]
     [{CASCADE CONSTRAINTS | INVALIDATE}] }]
    [SET UNUSED {COLUMN имя_столбца | (имя_столбца [, ...])}
     [{CASCADE CONSTRAINTS | INVALIDATE}]]
    [MODIFY { (имя_столбца тип_данных атрибуты [, ...]) |
     COLUMN имя_столбца [NOT] SUBSTITUTABLE AT ALL LEVELS
     [FORCE] }]
    [RENAME COLUMN имя_столбца TO новое_имя_столбца]
    [MODIFY {NESTED TABLE | VARRAY} элемент_коллекции
     [RETURN AS {LOCATOR | VALUE}]]

-- Изменение ограничений целостности
   [ADD CONSTRAINT имя_ограничения табличное_ограничение]
   [MODIFY CONSTRAINT имя_ограничения состояние_ограничения]
   [RENAME CONSTRAINT имя_ограничения
      TO новое_имя_ограничения]
   [DROP { { PRIMARY KEY | UNIQUE (столбец [, ...]) }
      [CASCADE] [{KEEP | DROP} INDEX] |
      CONSTRAINT имя_ограничения [CASCADE] } ]
-- Изменение секционирования
   [изменение_секционирования]

-- Изменение характеристик внешней таблицы
   DEFAULT DIRECTORY имя_каталога
```

```
      [ACCESS PARAMETERS
      {USING CLOB подзапрос | (непрозрачный_формат)}]
      LOCATION ( [имя_каталога:]'спецификация'[, ...] )
    [ADD (имя_столбца...)][DROP имя_столбца...]
    [MODIFY (имя_столбца...)]
    [PARALLEL целое_число | NOPARALLEL]
    [REJECT LIMIT { целое_число | UNLIMITED}]
    [PROJECT COLUMN {ALL | REFERENCED}]

  -- Перемещение таблицы
    [MOVE [ONLINE] [физические_атрибуты]
      [TABLESPACE табличное_пространство]
      [[NO]LOGGING] [PCTTHRESHOLD целое_число]
      [COMPRESS целое_число | NOCOMPRESS]
      [MAPPING TABLE | NOMAPPING]
      [[INCLUDING столбец] OVERFLOW
        [физические_атрибуты]
        [TABLESPACE табличное_пространство]
        [[NO]LOGGING]]
      [LOB ...] [VARRAY ...]
      [PARALLEL целое_число | NOPARALLEL]]

  -- Включение/выключение атрибутов и ограничений
    [{ {ENABLE | DISABLE} [[NO]VALIDATE]
      {UNIQUE (столбец [, ...]) | PRIMARY KEY |
      CONSTRAINT имя_ограничения}
      [USING INDEX {имя_индекса | CREATE_INDEX_оператор |
        [TABLESPACE табличное_пространство]
        [физические_атрибуты] [параметры_хранения]
        [NOSORT] [[NO]LOGGING] [ONLINE] [COMPUTE STATISTICS]
        [COMPRESS | NOCOMPRESS] [REVERSE]
        [{LOCAL | GLOBAL} секционирование]
        [EXCEPTIONS INTO имя_таблицы] [CASCADE]
        [{KEEP | DROP} INDEX]]} |
        [{ENABLE | DISABLE}] [{TABLE LOCK | ALL TRIGGERS}] }]
```

Параметры имеют следующие значения:

виртуальный_столбец

Позволяет создавать *виртуальные столбцы* (то есть столбцы, не хранящиеся физически, а вычисляемые динамически). Например, виртуальный столбец income может вычисляться суммированием столбцов salary, bonus и commission.

ограничение_столбца

Создает ограничение на уровне столбца, синтаксис приводится позднее.

***GROUP** журнальная_группа (столбец [NO LOG][, ...]) [ALWAYS] | DATA (ограничения [, ...]) COLUMNS*

Указывает для таблицы журнальную группу вместо одиночного журнального файла.

ON COMMIT {DELETE | PRESERVE} ROWS

Для временной таблицы определяет период хранения записей – в течение всей сессии (*PRESERVE*) или в течение одной транзакции (*DELETE*).

ограничение_таблицы

Создает ограничение на уровне таблицы, синтаксис приводится позднее.

физические_атрибуты

Определяет физические атрибуты таблицы, синтаксис приводится позднее.

TABLESPACE *табличное_пространство*

Указывает табличное пространство, в котором будет храниться создаваемая таблица. Если эта фраза опущена, то используется табличное пространство по умолчанию для схемы, в которой создается таблица. Детали приводятся ниже. Информацию о табличных пространствах и работе с ними вы можете найти в документации в разделе Oracle Concepts.

параметры_хранения

Определяет физические параметры хранения таблицы, синтаксис приводится позднее.

[NO]LOGGING

Определяет, генерируется при создании (а также при дальнейшей работе с данными) таблицы журнальная информация (*LOGGING*) или нет (*NOLOGGING*). *LOGGING* используется по умолчанию. *NOLOGGING* позволяет ускорить процесс создания таблицы. Но в случае сбоя в базе данных операция, выполненная в режиме *NOLOGGING*, не будет восстановлена в процессе применения журнальных файлов и таблицу нужно будет создавать заново. *LOGGING* используется вместо устаревшего ключевого слова *RECOVERABLE*.

CLUSTER(столбец [,...])

Указывает, что таблица является частью кластера. Список столбцов должен соответствовать столбцам в ранее созданном кластерном индексе. Так как в этом случае используются механизмы хранения кластерного индекса, то фраза *CLUSTER* несовместима с *физическими_атрибутами*, *параметрами хранения* и фразой *TABLESPACE*. Таблицы со столбцами типа *LOB* несовместимы с фразой *CLUSTER*.

ORGANIZATION HEAP

Определяет, каким образом данные записываются на диск. *HEAP* используется по умолчанию и не накладывает никаких ограничений на физический порядок записи строк на диск. С *ORGANIZATION HEAP* можно использовать несколько необязательных фраз, описанных в этом разделе, для определения параметров хранения, журналирования и сжатия.

ORGANIZATION INDEX

Определяет, каким образом данные записываются на диск. *INDEX* указывает, что строки таблицы должны записываться на диск в порядке сортировки по первичному ключу таблицы. Такой способ хранения называется *индексно-организованной таблицей*. Первичный ключ в такой таблице обязателен. При создании индексно-организованной таблицы вы можете указывать *физические_атрибуты* и *параметры_хранения*, фразу *TABLESPACE* и ключевое слово *NOLOGGING*. Также в дополнение вы можете использовать следующие подфразы:

PCTTHRESHOLD *целое_число*

Определяет процент пространства блока индекса, оставляемого под хранение данных. Данные, превышающие заданный объем, хранятся в сегменте переполнения.

INCLUDING *столбец*

Определяет точку, в которой запись делится на части, хранящиеся в блоках индекса и в сегменте переполнения. Все столбцы после указанного хранятся в сегменте переполнения. Указанный столбец не может быть частью первичного ключа.

MAPPING TABLE | NOMAPPING

Создает таблицу соответствия локальных и физических идентификаторов строк (ROWID), что необходимо для построения по индекс-таблицам дополнительных индексов на основе битовых карт. При секционировании основной таблицы таблица соответствия секционируется аналогичным образом. *NOMAPPING* указывает, что таблица соответствия идентификаторов строк не нужна.

[INCLUDING *столбец]* *OVERFLOW*

Определяет сегмент, в котором сохраняются строки, не помещающиеся в свободное пространство блока индекса, задаваемое параметром *PCTTHRESHOLD*. Для определяемого сегмента переполнения можно указывать *физические_атрибуты* и *параметры_хранения*, фразы *TABLESPACE* и ключевое слово *NOLOGGING*. С помощью фразы *INCLUDING* *столбец* можно указать точку деления записи на часть, хранящуюся в блоке индекса, и часть, хранящуюся в сегменте переполнения. Столбцы первичного ключа всегда хранятся в индексе. Все столбцы, не входящие в первичный ключ и следующие за указанным столбцом, хранятся в сегменте переполнения.

ORGANIZATION EXTERNAL

EXTERNAL указывает, что данные таблицы хранятся вне базы данных и как правило доступны только для чтения (но метаданные таблицы хранятся внутри базы данных). Существуют некоторые ограничения на внешние таблицы: они не могут быть временными, иметь ограничения целостности, для них может указываться только столбец, тип столбца и параметры столбца, типы данных *LOB* и *LONG* запрещены. Также в дополнение вы можете использовать следующие подфразы:

TYPE *тип_драйвера*

Определяет драйвер доступа к внешней таблице. По умолчанию используется *ORACLE_LOADER*.

DEFAULT DIRECTORY *имя_каталога*

Указывает каталог файловой системы, в котором хранится внешняя таблица.

ACCESS PARAMETERS {USING CLOB *подзапрос* *| (* *непрозрачный формат* *)}*

Устанавливает и передает драйверу доступа дополнительные параметры. Oracle не интерпретирует эту информацию. *USING CLOB* указывает подзапрос для генерации параметров и их значений, подзапрос должен возвращать одну строку и один столбец типа *CLOB*. Подзапрос не должен со-

держать *ORDER BY*, *UNION*, *INTERSECT*, *MINUS* и *EXCEPT*. Параметр *непрозрачный_формат* позволяет явно перечислить все параметры и их значения, как это описано в разделе для *ORACLE LOADER* в разделе документации Oracle Database Utilities.

LOCATION (имя_каталога:'спецификация'[, ...])

Определяет один или несколько внешних источников данных, обычно файлов. Oracle не интерпретирует эту информацию.

REJECT LIMIT {целое_число | UNLIMITED}

Определяет число ошибок конвертации, при достижении которого запрос к внешней таблице завершается с ошибкой. По умолчанию используется 0. *UNLIMITED* указывает, что запрос будет выполняться при любом количестве ошибок.

{ENABLE | DISABLE} ROW MOVEMENT

Указывает, разрешено (*ENABLE*) или запрещено (*DISABLE*) физическое перемещение (а следовательно, изменение ROWID) строк из одной секции в другую в случае изменения значения ключа секционирования. Если перемещение строк запрещено, то Oracle завершает с ошибкой оператор *UPDATE*, для выполнения которого требуется перемещение строк.

[NO]CACHE

Кэширует таблицу (*CACHE*) для возможности быстрого чтения либо явно отключает эту возможность (*NOCACHE*). Для индексно-организованных таблиц всегда используется *CACHE*.

[NO]MONITORING

Включает или выключает режим сбора статистики обновлений таблицы. По умолчанию используется *NOMONITORING*.

[NO]ROWDEPENDENCIES

Включает для таблицы режим *отслеживания зависимостей на уровне строк*. Для каждой строки сохраняется системный номер последнего изменения этой строки. Для хранения системного номера изменения используется дополнительно 6 байт. Отслеживание зависимостей на уровне строк наиболее полезно при репликации с параллельным распространением данных. По умолчанию используется *NOROWDEPENDENCIES*.

[NO]FLASHBACK ARCHIVE

Включает или отключает режим отслеживания истории изменений таблицы. По умолчанию используется *NO FLASHBACK ARCHIVE*.

PARALLEL [целое_число] | NOPARALLEL

Фраза *PARALLEL* позволяет распараллеливать создание таблицы с использованием нескольких процессоров для ускорения операции. Также с помощью этой фразы включается параллелизм для запросов и других манипуляций с таблицей после ее создания. По желанию можно явно указать число параллельных процессов, используемых для создания таблицы и для дальнейшей работы с ней. (Oracle может автоматически вычислять наилучшую степень параллелизма для каждой операции, поэтому указание точного числа во фразе *PARALLEL* необязательно.) *NOPARALLEL* создает таблицу последовательно и запрещает в будущем параллельные запросы и манипуляции с данными.

COMPRESS [целое_число] | NOCOMPRESS

Включает или отключает сжатие таблицы. В индекс-таблицах сжимается только первичный ключ, обычные таблицы сжимаются полностью. Сжатие может значительно уменьшить занимаемое таблицей место. По умолчанию используется *NOCOMPRESS*. Для индекс-таблиц можно указать точное число столбцов для сжатия. По умолчанию сжимаются все столбцы первичного ключа, кроме последнего.

{ENABLE | DISABLE} [[NO]VALIDATE] {UNIQUE (столбец[, ...]) | PRIMARY KEY | CONSTRAINT имя_ограничения}

Определяет, относится ограничение или ключ ко всем данным новой таблицы или нет. *ENABLE* означает, что ограничение накладывается на все новые данные таблицы, а *DISABLE* указывает, что ограничение отключено. Можно использовать следующие дополнительные опции:

[NO]VALIDATE

VALIDATE проверяет, что все существующие данные таблицы удовлетворяют ограничению. Если *NOVALIDATE* используется совместно с *ENABLE*, то Oracle не проверяет существующие данные на соответствие ограничению, но проверяет все новые данные.

UNIQUE (столбец [, ...]) | PRIMARY KEY | CONSTRAINT имя_ограничения

Указывает уникальный ключ, первичный ключ или другое ограничение, которое включается или выключается.

USING INDEX имя_индекса | оператор_CREATE_INDEX

Указывает имя существующего индекса, используемого для поддержки ограничения, или создает новый индекс с заданными параметрами. Если не указано ни то ни другое, то Oracle самостоятельно создает новый индекс.

EXCEPTIONS INTO имя_таблицы

Указывает таблицу, в которую Oracle помещает информацию о строках, нарушающих ограничения. Предварительно выполните скрипт *utlexpt1.sql* для создания такой таблицы.

CASCADE

Используется для каскадного отключения всех зависимых ограничений целостности. Используется только с фразой *DISABLE*.

{KEEP | DROP} INDEX

Позволяет оставить (*KEEP*) или удалить (*DROP*) индекс, используемый для поддержки первичного или уникального ключа. Ключ можно удалить только при его отключении.

секционирование

Определяет секционирование и подсекционирование таблицы. Синтаксис секционирования достаточно сложен и рассматривается с примерами в одном из следующих разделов.

AS подзапрос

Определяет подзапрос, используемый для наполнения таблицы данными. Имена и типы данных столбцов подзапроса могут использоваться вместо имен и типов данных столбцов в определении таблицы.

OF объектный_тип

Указывает, что таблица создается на базе существующего объектного типа.

[NOT] SUBSTITUTABLE AT ALL LEVELS

Указывает допустимость вставки в таблицу строк, соответствующих подтипам основного типа таблицы. По умолчанию используется *SUBSTITUTABLE AT ALL LEVELS*.

ссылочное_ограничение_столбца и *ссылочное_ограничение_таблицы*

Объявляет ссылочное ограничение целостности для объектной таблицы или таблицы типа XMLType. Эти фразы детально рассматриваются позднее в этом разделе.

OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}

Указывает, что идентификатор объекта (*OID*) в объектной таблице создается системой (*SYSTEM GENERATED*) или базируется на первичном ключе (*PRIMARY KEY*). По умолчанию используется *SYSTEM GENERATED*.

OIDINDEX [имя_индекса]

Объявляет индекс и, возможно, имя индекса для генерируемого системой идентификатора объекта. При необходимости можно также указать *физические_атрибуты* и *параметры_хранения*. Если *OID* базируется на первичном ключе, то эта фраза необязательна.

OF XMLTYPE

Указывает, что таблица создается на базе типа данных *XMLTYPE*.

XMLTYPE {OBJECT RELATIONAL [атрибуты_хранения_xml] | [{SECUREFILE | BASICFILE}] [{CLOB | BINARY XML} [имя_lob_сегмента] [параметры_lob]]

Определяет способ хранения *XML* данных: в *LOB*-объекте, объектно-реляционным способом или в двоичном формате *XML*. *OBJECT RELATIONAL* хранит данные в объектно-реляционных столбцах и позволяет индексировать столбцы для лучшей производительности. При этом необходимо указывать *спецификацию_xml_схемы* и предварительно регистрировать схему с помощью пакета *DBMS_XMLSCHEMA*. *CLOB* указывает, что данные *XMLTYPE* хранятся в *LOB*-столбце для быстрого извлечения. Вы дополнительно можете указать имя сегмента и параметры хранения *LOB*, но вы не можете указать детали LOB и спецификацию схемы *XML* в том же операторе. *BINARY XML* хранит данные в компактном двоичном формате, и для соответствующего столбца *BLOB* указываются любые дополнительные параметры.

спецификация_xml_схемы

Позволяет указать URL одной или нескольких зарегистрированных XML-схем, а также имя XML-элемента. Имя элемента обязательно, а URL опционален. Несколько схем можно указывать только при хранении данных в формате *BINARY XML*. Также вы можете указать *ALLOW ANYCHEMA* для хранения в столбце *XMLTYPE* любого документа, базирующегося на схеме, *ALLOW NONSCHEMA* для хранения документов, не базирующихся на схеме, и *DISALLOW NONSCHEMA* для запрещения хранения документов, не базирующихся на схеме.

READ ONLY | READ WRITE

READ ONLY переводит таблицу в режим «только чтение», при котором запрещены все операции DML, включая *SELECT ... FOR UPDATE*, а разрешены только обычные операторы *SELECT*. *READ WRITE* переводит таблицу в нормальный режим работы.

ADD...

Добавляет в таблицу новый столбец, виртуальный столбец, ограничение, сегмент переполнения или дополнительную журнальную группу (supplemental log group). Также можно изменять таблицу типа *XMLTYPE* путем добавления (или удаления) одной или нескольких XML-схем.

MODIFY...

Изменяет существующий столбец, ограничение или дополнительную журнальную группу.

DROP...

Удаляет из таблицы столбец, ограничение или дополнительную журнальную группу. Вы можете явно удалить неиспользуемые столбцы при помощи фразы *DROP UNUSED COLUMNS*. При этом неиспользуемые столбцы удаляются всегда при *любом* удалении столбца. При использовании ключевого слова *INVALIDATE* все объекты, зависимые от удаляемого объекта (такие как представления и хранимые процедуры), становятся невалидными до их явной рекомпиляции. Фраза *COLUMNS CONTINUE* используется в тех случаях, когда оператор *DROP COLUMN* завершился с ошибкой, и вы хотите продолжить операцию с места остановки.

RENAME...

Переименовывает столбец или ограничение.

SET UNUSED...

Помечает столбец как неиспользуемый. Неиспользуемые столбцы недоступны для запросов, но при этом учитываются при подсчете количества столбцов таблицы. *SET UNUSED* является самым быстрым способом сделать столбец недоступным, но не самым лучшим. Используйте *SET UNUSED* только как временную меру до того момента, когда вы сможете выполнить *ALTER TABLE...DROP*.

COALESCE

Соединяет содержимое блоков индексно-организованной таблицы с целью освобождения части блоков для повторного использования. Параметр *COALESCE* похож на *SHRINK*, но упаковывает сегменты менее плотно и не освобождает неиспользованное пространство.

ALLOCATE EXTENT

Явно выделяет новый экстент для таблицы. Для экстента можно в любой комбинации указывать параметры *SIZE*, *DATAFILE* и *INSTANCE*. Размер экстента можно указывать в байтах (без суффикса), килобайтах (*K*), мегабайтах (*M*), гигабайтах (*G*) или терабайтах (*T*).

DEALLOCATE UNUSED [KEEP целое_число [K|M|G|T]]

Освобождает неиспользуемое пространство в конце таблицы, *LOB* сегмента, секции или подсекции. Освобожденное пространство может быть использовано другими объектами базы данных. Слово *KEEP* используется для указания количества свободного места, которое необходимо оставить.

SHRINK SPACE [COMPACT] [CASCADE]

Уменьшает таблицу, индекс-таблицу, индекс, секцию, подсекцию, материализованное представление или журнал материализованного представления. Сжимать можно только сегменты, находящиеся в табличных пространствах с автоматическим управлением сегментами. *SHRINK* перемещает строки, поэтому возможность перемещения строк должна быть включена командой *ENABLE ROW MOVEMENT*. Oracle сжимает сегменты, освобождает неиспользуемое пространство и сбрасывает маркер максимального уровня заполнения, если только дополнительно не указаны опции *COMPACT* и/или *CASCADE*. *COMPACT* только дефрагментирует сегмент и уплотняет строки таблицы для последующего освобождения, но не меняет маркер максимального уровня заполнения и не освобождает пространство. *CASCADE* выполняет ту же операцию (с некоторыми ограничениями) для всех зависимых объектов, включая дополнительные индексы индексно-организованных таблиц. Используется только с *ALTER TABLE*.

UPGRADE [NOT] INCLUDING DATA

Конвертирует метаданные объектных таблиц и таблиц с объектными столбцами к последней версии каждого используемого объектного типа. При использовании *INCLUDING DATA* конвертируются также данные таблицы.

MOVE ...

Перемещает таблицу, индексно-организованную таблицу, секцию или подсекцию в другое табличное пространство.

[NO]MINIMIZE RECORDS_PER_BLOCK

Ограничивает или не ограничивает допустимое количество записей в блоке. При использовании *MINMIZE* вычисляется максимальное число записей в блоке, и лимит числа записей устанавливается в это значение. (Лучше всего это делать не репрезентативном наборе данных.) Эта фраза несовместима с вложенными таблицами и индекс-таблицами. По умолчанию используется *NOMINIMIZE*.

PROJECT COLUMN {REFERENCE|ALL}

Определяет способ валидации строк внешней таблицы. *REFERENCES* обрабатывает только столбцы, используемые в запросе. *ALL* обрабатывает значения всех столбцов. При использовании *ALL* некорректные строки отбрасываются, даже если ошибка найдена в столбцах, не используемых в запросе. *ALL* возвращает всегда консистентные результаты, в то время как *REFERENCES* возвращает переменное число строк в зависимости от используемых в запросе столбцов.

{ENABLE|DISABLE} {TABLE LOCK|ALL TRIGGERS}

Включает или выключает, соответственно, блокировки уровня таблицы и все триггеры. *ENABLE TABLE LOCK* используется при изменении структуры

существующей таблицы, но не требуется при чтении или изменении данных таблицы.

Глобальные временные таблицы доступны всем пользовательским сессиям, но отдельные строки видны только тем сессиям, которые вставили эти строки. Опция *ON COMMIT*, используемая только при создании временных таблиц, определяет момент очистки временной таблицы – после каждой фиксации транзакции (*DELETE ROWS*) или в конце сессии (*PRESERVE ROWS*). Например:

```
CREATE GLOBAL TEMPORARY TABLE shipping_schedule
  (ship_date DATE,
   receipt_date DATE,
   received_by VARCHAR2(30),
   amt NUMBER)
ON COMMIT PRESERVE ROWS;
```

Этот оператор создает временную таблицу **shipping_schedule**, строки которой сохраняются на протяжении нескольких транзакций.

Фраза «физические_атрибуты» в Oracle

Физические_атрибуты определяют характеристики хранения таблицы или отдельной секции таблицы. Для определения атрибутов новой таблицы или для изменения атрибутов существующей таблицы укажите следующие значения:

```
[{PCTFREE  целое_число |
  PCTUSED  целое_число |
  INITRANS целое_число |
  атрибуты_хранения}]
```

где:

PCTFREE *целое_число*

Определяет процент места, оставляемого свободным в каждом блоке данных. Например, значение 10 резервирует 10% места в блоке для вставки новых записей.

PCTUSED *целое_число*

Определяет процент занятого места в блоке, при котором блок будет использован для вставки новых записей. Например, значение 90 означает, что новые строки будут вставляться в блок, как только в нем окажется занято менее 90% места. Сумма *PCTUSED* и *PCTFREE* не может превышать 100.

INITRANS *целое_число*

Устанавливает для блока данных начальное значение допустимых параллельных транзакций. Значение должно быть в пределах от 1 до 255.

В версиях Oracle до 11g использовался параметр *MAXTRANS* для ограничения максимального числа транзакций для блока данных, но сейчас этот параметр устарел. Oracle 11g автоматически устанавливает для *MAXTRANS* значение 255, вне зависимости от того, какое значение вы указали явно (хотя существующие объекты сохраняют установленное значение *MAXTRANS*).

Фраза «атрибуты_хранения» и объекты LOB в Oracle

Атрибуты хранения определяют некоторые физические параметры хранения данных:

```
STORAGE
( [ {INITIAL целое_число [K | M | G | T]
   | NEXT целое_число [K | M]
   | MINEXTENTS целое_число
   | MAXEXTENTS { целое_число | UNLIMITED}
   | PCTINCREASE целое_число
   | FREELISTS целое_число
   | FREELIST GROUPS целое_число
   | BUFFER_POOL {KEEP | RECYCLE | DEFAULT}} ] [...] )
```

При указании атрибутов хранения возьмите их в скобки и разделите пробелами, например *(INITIAL 32M NEXT 8M)*. Атрибуты имеют следующие значения:

INITIAL *целое_число* *[K|M|G|T]*

Устанавливает размер начального экстента таблицы в байтах, килобайтах (*K*), мегабайтах (*M*), гигабайтах (*G*) или терабайтах (*T*).

NEXT *целое_число* *[K|M]*

Устанавливает размер выделяемого пространства после того, как заполнен начальный экстент.

MINEXTENTS *целое_число*

Указывает минимальное число создаваемых экстентов. По умолчанию создается только один экстент, но можно создать и большее число.

MAXEXTENTS *целое_число* *|UNLIMITED*

Устанавливает максимально допустимое число экстентов. Если установлено *UNLIMITED*, то число экстентов не ограничивается. (Используйте *UNLIMITED* осторожно, так как при этом таблица может использовать все свободное пространство на диске.)

PCTINCREASE *целое_число*

Контролирует рост размера объекта после первого увеличения. Размер начального экстента определяется явно параметром *INITIAL*, размер второго экстента определяется параметром *NEXT*, размер третьего экстента равен *NEXT* + *NEXT*PCTINCREASE* и т. д. Если *PCTINCREASE* равен 0, то для каждого следующего экстента используется размер *NEXT*. В противном случае размер каждого последующего экстента больше предыдущего.

FREELISTS *целое_число*

Устанавливает количество списков свободных блоков для каждой группы. По умолчанию используется значение 1.

FREELIST GROUPS *целое_число*

Устанавливает количество групп списков свободных блоков. По умолчанию используется значение 1.

BUFFER_POOL {KEEP | RECYCLE | DEFAULT}

Определяет буферный кэш, используемый для хранения закэшированных блоков некластерной таблицы. Индексно-организованные таблицы могут использовать разные буферные кэши для сегмента, в котором хранится индекс, и для сегмента переполнения. Секции таблицы наследуют буферный кэш из определения таблицы, если только для секции явно не указан отдельный буферный кэш.

KEEP

Помещает блоки объекта в буферный кэш *KEEP*. Это улучшает производительность из-за уменьшения количества операций ввода-вывода. *KEEP* имеет приоритет над параметром *NOCACHE*.

RECYCLE

Помещает блоки объекта в буферный кэш *RECYCLE*.

DEFAULT

Помещает блоки в буферный кэш *DEFAULT*. *DEFAULT* используется по умолчанию, если не выбрана ни одна опция.

Например, мы можем создать таблицу **books_sales** в табличном пространстве **sales**, с начальным размером 8 Мбайт и последующим увеличением на 8 Мбайт. Таблица будет иметь от 1 до 8 экстентов, таким образом ее общий размер будет не больше 64 Мбайт:

```
CREATE TABLE book_sales
  (qty NUMBER,
   period_end_date DATE,
   period_nbr NUMBER)
TABLESPACE sales
STORAGE (INITIAL 8M NEXT 8M MINEXTENTS 1 MAXEXTENTS 8);
```

Рассмотрим пример создания таблицы **large_objects**, содержащей *LOB*-объекты для хранения текста и изображений:

```
CREATE TABLE large_objects
  (pretty_picture   BLOB,
   interesting_text CLOB)
STORAGE (INITIAL 256M NEXT 256M)
LOB (pretty_picture, interesting_text)
  STORE AS (TABLESPACE large_object_segment
    STORAGE (INITIAL 512M NEXT 512M)
    NOCACHE LOGGING);
```

Для описания столбцов типов *LOB*, *BLOB* и *CLOB* используется фраза *параметры_lob*. Объекты *LOB* могут появляться в разных частях объявления таблицы. Например, отдельные описания объектов *LOB* могут быть на уровне отдельной секции, на уровне подсекции и на уровне самой таблицы. Синтаксис фразы *параметры_lob* следующий:

```
{TABLESPACE имя_табличного_пространства]
  [{SECUREFILE | BASICFILE}]
  [{ENABLE | DISABLE} STORAGE IN ROW]
  [атрибуты_хранения] [CHUNK целое_число]
  [PCTVERSION целое_число]
```

```
    [RETENTION [{MAX | MIN целое_число | AUTO | NONE}]]
    [{DEDUPLICATE | KEEP_DUPLICATES}]
    [{NOCOMPRESS | COMPRESS [{HIGH | MEDIUM}]}]
    [FREEPOOLS целое_число]
  [{CACHE | {NOCACHE | CACHE READS} [{LOGGING | NOLOGGING}]}]
```

В этой фразе все параметры аналогичны параметрам, определяемым для LOB на уровне таблицы. Следующие параметры уникальны и используются только для объектов *LOB*:

SECUREFILE | BASICFILE

Определяет формат хранения объектов *LOB*: либо в высокоэффективном формате (*SECUREFILE*), либо в традиционном формате (*BASICFILE*). При хранении в формате *SECUREFILE* вы получаете доступ к новым возможностям, таким как сжатие, шифрование, удаление дубликатов.

{ENABLE | DISABLE} STORAGE IN ROW

Указывает, хранится ли объект *LOB* в одной строке с остальными столбцами таблицы (*ENABLE*) или вне строки (*DISABLE*). Хранение в строке допустимо, если объекты относительно небольшие – 4000 байт или менее. Этот параметр нельзя поменять после установки.

CHUNK целое_число

Выделяет указанное число байт для манипуляции с объектами *LOB*. Если указанное число байт не кратно размеру блока, то Oracle округлит его в большую сторону. Указанное число также должно быть меньше или равно значению параметра *NEXT* из фразы *атрибуты_хранения*, в противном случае возникнет ошибка. По умолчанию используется размер одного блока данных. Этот параметр нельзя поменять после установки.

PCTVERSION целое_число

Определяет максимальный размер в процентах от всего места хранения *LOB*, используемый для хранения старых версий объектов. По умолчанию используется значение 10%.

RETENTION [{MAX | MIN целое_число | AUTO | NONE}]

Используется вместо *PCTVERSION* в базах данных с автоматическими сегментами отката. *RETENTION* указывает, что необходимо сохранять старые версии *LOB*-объектов. При хранении в формате *SECUREFILE* можно указывать дополнительные опции. *MAX* разрешает файлу отката расти до достижения сегментом *LOB* максимального размера, который определен параметром *MAXSIZE* во фразе *атрибуты_хранения*. *MIN* ограничивает данные отката заданным числом секунд, если база данных работает в ретроспективном режиме (flashback mode). *AUTO* (используется по умолчанию) поддерживает данные отката в количестве, достаточном для согласованного чтения. *NONE* указывает, что данные отката не требуются.

DEDUPLICATE | KEEP_DUPLICATES

Указывает, требуется ли хранить дубликаты значений в *LOB*-сегменте (*KEEP_DUPLICATES*) или нужно удалять дубликаты (*DEDUPLICATE*, значение по умолчанию). Может использоваться только при хранении в формате *SECUREFILE*.

NOCOMPRESS | COMPRESS [{HIGH | MEDIUM}]

NOCOMPRESS (значение по умолчанию) отключает сжатие *LOB*-объектов на стороне сервера. Вы можете включить среднюю (*MEDIUM*) или высокую (*HIGH*) степень сжатия. Высокая степень сжатия требует дополнительных накладных расходов. Сжатие может использоваться только при хранении в формате *SECUREFILE*.

Следующий пример создает таблицу **large_objects** с добавленными параметрами хранения и сохранением старых значений:

```
CREATE TABLE large_objects
  (pretty_picture   BLOB,
   interesting_text CLOB)
STORAGE (INITIAL 256M NEXT 256M)
  LOB (pretty_picture, interesting_text)
  STORE AS (TABLESPACE large_object_segment
    STORAGE (INITIAL 512M NEXT 512M)
    NOCACHE LOGGING
    ENABLE STORAGE IN ROW
    RETENTION);
```

По сравнению с прошлым примером мы добавили параметры *STORAGE IN ROW* и *RETENTION*. Так как мы не установили значение для параметра *CHUNK*, то будет использовано значение по умолчанию.

Вложенные таблицы в Oracle

В Oracle можно создавать вложенные таблицы (*NESTED TABLE*), которые виртуально хранятся в столбце другой таблицы. Фраза *STORE AS* позволяет создавать прокси-имя для вложенной таблицы, но для самой вложенной таблицы должен быть предварительно создан пользовательский тип данных. Эта возможность полезна для разреженных наборов значений, но мы не рекомендуем использовать ее в повседневных задачах. В этом примере создается таблица **proposal_types** с вложенной таблицей **props_nt**, являющейся типом **props_nt_table**:

```
CREATE TYPE prop_nested_tbl AS TABLE OF props_nt;
CREATE TABLE proposal_types
   (proposal_category VARCHAR2(50),
   proposals          PROPS_NT)
NESTED TABLE props_nt STORE AS props_nt_table;
```

Сжатие таблиц в Oracle

Начиная с версии 9i Release 2 в Oracle поддерживается сжатие как ключей индексов, так и целиком таблиц (Oracle 9i Release1 поддерживал только сжатие ключей индексов). Хотя при сжатии появляются накладные расходы, оно существенно уменьшает занимаемое таблицей пространство. Сжатие особенно актуально для баз данных, не укладывающихся в требования по объему. Сжатие ключей индекса указывается во фразе *COMPRESS при создании индекса или изменении индекса*, сжатие таблицы указывается в *ORGANIZE HEAP*, а сжатие индексно-организованных таблиц – во фразе *ORGANIZE INDEX*.

Секционирование таблиц в Oracle

Oracle позволяет создавать в таблицах секции и подсекции. Также можно разбить на отдельные секции и *LOB*-объекты. Таблицу можно разбить на заданное число секций и расположить их в разных частях дисковой системы для ускорения ввода-вывода (используя секционирование по диапазону значений, по списку значений, по хеш-значениям или составное секционирование) либо можно использовать системное секционирование. Для секционирования используется следующий синтаксис:

```
{ PARTITION BY RANGE (столбец [, ...])
     [INTERVAL (выражение)
       [STORE IN (табличное_пространство[, ...])]]
     (PARTITION [имя_секции]
        VALUES LESS THAN ({MAXVALUE | значение}[, ...])
     [описание_секции]) |
  PARTITION BY HASH (столбец [, ...])
     {(PARTITION [имя_секции]
     [атрибуты_хранения_секции][, ...]) |
     PARTITIONS число_хеш_секций
         [STORE IN (табличное_пространство [, ...])]
     [OVERFLOW STORE IN (табличное_пространство [, ...])]} |
  PARTITION BY LIST (столбец[, ...])
     (PARTITION [имя_секции]
        VALUES ({MAXVALUE | значение }[, ...])
     [описание_секции]) |
  PARTITION BY RANGE (столбец column[, ...])
     {подсекционирование_по_списку_значений |
      подсекционирование_по_хеш_значениям}
     (PARTITION [имя_секции] VALUES LESS THAN
         ({MAXVALUE | значение }[, ...])
     [описание_секции]) |
  PARTITION BY SYSTEM [целое_число] |
  PARTITION BY REFERENCE (ограничение)
     [(PARTITION [имя_секции]
     [описание_секции][, ...]) ]
}
```

В следующем примере создается таблица orders, секционированная по диапазону значений:

```
CREATE TABLE orders
  (order_number NUMBER,
   order_date   DATE,
   cust_nbr     NUMBER,
   price        NUMBER,
   qty          NUMBER,
   cust_shp_id  NUMBER)
PARTITION BY RANGE(order_date)
  (PARTITION pre_yr_2000 VALUES LESS THAN
     TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
   PARTITION pre_yr_2004 VALUES LESS THAN
     TO_DATE('01-JAN-2004', 'DD-MON-YYYY'
   PARTITION post_yr_2004 VALUES LESS THAN
     MAXVALUE));
```

В этом примере таблица **orders** разбивается на три секции: одна для заказов, полученных до 2000 года (pre**_yr_2000**), одна для заказов до 2004 года (**pre_yr_2004**) и одна для заказов после 2004 года (**post_yr_2004**). Секции основываются на диапазонах значений в поле **order_date**.

Фраза *INTERVAL* расширяет функциональность секционирования по диапазонам значений путем автоматического создания новых секций при появлении данных, не попадающих в существующие секции. Вместо явного создания секций вы лишь указываете интервал значений для каждой следующей секции. Используйте *STORE IN* для указания табличного пространства, в котором будут храниться интервальные секции. Интервальное секционирование нельзя применять к индексно-организованным таблицам, таблицам с предметными индексами, а также нельзя создавать интервальные подсекции.

В следующем примере создается таблица **orders**, секционированная по хеш-значениям столбца **cust_shp_id**:

```
CREATE TABLE orders
  (order_number NUMBER,
   order_date   DATE,
   cust_nbr     NUMBER,
   price        NUMBER,
   qty          NUMBER,
   cust_shp_id  NUMBER)
PARTITION BY HASH (cust_shp_id)
  (PARTITION shp_id1 TABLESPACE tblspc01,
   PARTITION shp_id2 TABLESPACE tblspc02,
   PARTITION shp_id3 TABLESPACE tblspc03)
ENABLE ROW MOVEMENT;
```

Основным отличием секционирования по диапазону значений и по хеш-значениям является то, что в первом случае явно определяется, по какому критерию для каждой строки выбирается секция, а во втором случае Oracle применяет к значениям полей строки хеш-функцию и по результату хеш-функции определяется секция, в которую попадает строка. (Обратите внимание, что для таблицы мы также включили возможность перемещения строк.)

В дополнение к разбиению таблицы на секции (для упрощения резервного копирования, восстановления и повышения производительности) вы можете разбить секции на подсекции. Синтаксис фразы *подсекционирование_по_списку_значений* выглядит следующим образом:

```
SUBPARTITION BY LIST (столбец)
[SUBPARTITION TEMPLATE
  { (SUBPARTITION имя_подсекции
      [VALUES {DEFAULT | {значение | NULL}[, ...]}]
      [атрибуты_хранения]) |
     число_хехешш_подсекций } ]
```

В качестве примера мы создадим таблицу **order** еще раз, но применим составное секционирование: секции создадим по диапазонам значений, а подсекции создадим по спискам значений. Так как при подсекционировании нам потребуется перечислять все значения, попадающие в каждую подсекцию, то лучше исполь-

зовать столбцы с небольшими списками значений. В этом примере мы используем столбец **shp_region_id**:

```
CREATE TABLE orders
   (order_number NUMBER,
    order_date   DATE,
    cust_nbr     NUMBER,
    price        NUMBER,
    qty          NUMBER,
    cust_shp_id  NUMBER,
    shp_region   VARCHAR2(20))
PARTITION BY RANGE(order_date)
SUBPARTITION BY LIST(shp_region)
  SUBPARTITION TEMPLATE(
    (SUBPARTITION shp_region_north
       VALUES ('north','northeast','northwest'),
     SUBPARTITION shp_region_south
       VALUES ('south','southeast','southwest'),
     SUBPARTITION shp_region_central
       VALUES ('midwest'),
     SUBPARTITION shp_region_other
       VALUES ('alaska','hawaii','canada')
 (PARTITION pre_yr_2000 VALUES LESS THAN
     TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
  PARTITION pre_yr_2004 VALUES LESS THAN
     TO_DATE('01-JAN-2004', 'DD-MON-YYYY'
  PARTITION post_yr_2004 VALUES LESS THAN
     MAXVALUE) )
ENABLE ROW MOVEMENT;
```

В этом примере строки таблицы делятся на три секции по значениям столбца **order_date**, а затем внутри каждой секции строки распределяются на четыре подсекции по значениям столбца **shp_region**. Используя *SUBPARTITION TEMPLATE*, вы создаете одинаковый набор подсекций для каждой секции. Если для каждой секции требуется свой набор подсекций, то вы можете указать подсекции каждой секции явно, не используя *SUBPARTITION TEMPLATE*.

Вы можете создать подсекции на основании хеш-значений. Фраза *подсекционирование_по_хеш_значениям* выглядит следующим образом:

```
SUBPARTITION BY HASH (столбец [, ...])
  {SUBPARTITIONS количество [STORE IN
     (табличное_пространство[, ...])] |
   SUBPARTITION TEMPLATE
   { (SUBPARTITION имя_подсекции
     [VALUES {DEFAULT | {значение | NULL}
     [, ...]}] [атрибуты_хранения]) |
     число_хеш_подсекций }}
```

Фраза *описание_секции*, используемая в синтаксисе секционирования, является очень гибкой и поддерживает работу с данными *LOB* и *VARRAY*:

```
[атрибуты_сегмента] [[NO] COMPRESS [целое_число]]
   [OVERFLOW атрибуты_сегмента]
[подсекции_на_уровне_секций]
```

```
[{ LOB { (lob_элемент[, ...]) STORE AS параметры_lob |
     (lob_элемент) STORE AS {lob_сегмент (параметры_lob) |
      lob_сегмент | (параметры_lob)} } |
   VARRAY varray_элемент
     [{[ELEMENT] IS OF [TYPE] (ONLY имя_типа) |
     [NOT] SUBSTITUTABLE AT ALL LEVELS}]
        STORE AS LOB { lob_сегмент |
        [lob_сегмент] (параметры_lob) } |
     [{[ELEMENT] IS OF [TYPE] (ONLY имя_типа) |
         [NOT] SUBSTITUTABLE AT ALL LEVELS}] }]
```

Фраза *подсекции_на_уровне_секций* имеет следующий вид:

```
{SUBPARTITIONS число_хеш_подсекций
   [STORE IN (имя_табличного_пространства[, ...])] |
   SUBPARTITION имя_подсекции
      [VALUES {DEFAULT | {значение | NULL}[, ...] ]
[атрибуты_хранения] }
```

Фраза *атрибуты_хранения* используется для описания хранения элементов секции или подсекции. Синтаксис фразы следующий:

```
[[TABLESPACE имя_табличного_пространства] |
[OVERFLOW TABLESPACE имя_табличного_пространства] |
  VARRAY varray_элемент STORE AS LOB lob_сегмент |
  LOB (lob_элемент) STORE AS {
   (TABLESPACE имя_табличного_пространства) |
   lob_сегмент [(TABLESPACE имя_табличного_пространства)] }]
```

Системное секционирование (*SYSTEM*) очень просто использовать, потому что для него не требуется указывать ключ секционирования или диапазоны и списки значений. Наоборот, секции *SYSTEM* являются эквисекционированными подчиненными таблицами, как вложенные таблицы, чья родительская таблица секционирована. Если вы не укажете число секций, то Oracle создаст одну секцию с именем, начинающимся с **SYS_P**. Либо вы можете указать требуемое число секций, но не более 1024 K – 1. Системно секционированные таблицы эквивалентны другим секционированным и подсекционированным таблицам, только не поддерживают параметр *OVERFLOW* во фразе *описание_секции*.[1]

Ссылочное секционирование (*REFERENCE*) объявляется только при создании таблицы. Оно выполняется аналогично секционированию другой таблицы, на которую создаваемая таблица ссылается посредством внешнего ключа. Все операции по секционированию основной таблицы автоматически применяются и к подчиненной таблице.

---

[1] Вероятно, авторы имели в виду ссылочное секционирование (reference partitioning). При системном или «бесключевом» секционировании секционированием данных управляет приложение, а не сама СУБД. Отсутствие ключа секционирования не позволяет автоматически производить отсекание ненужных секций (pruning). Однако при этом можно воспользоваться другими преимуществами секционирования, такими как управляемость, повышенная доступность данных, локальное индексирование и индивидуальные атрибуты секций. – *Прим. науч. ред.*

В последнем примере, посвященном секционированию, мы опять создадим таблицу **orders** с указанием составного секционирования и атрибутов хранения секций и *LOB* столбца:

```
CREATE TABLE orders
   (order_number NUMBER,
    order_date   DATE,
    cust_nbr     NUMBER,
    price        NUMBER,
    qty          NUMBER,
    cust_shp_id  NUMBER,
    shp_region   VARCHAR2(20),
    order_desc   NCLOB)
PARTITION BY RANGE(order_date)
SUBPARTITION BY HASH(cust_shp_id)
  (PARTITION pre_yr_2000 VALUES LESS THAN
     TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
       TABLESPACE tblspc01
     LOB (order_desc) STORE AS (TABLESPACE tblspc_a01
       STORAGE (INITIAL 10M NEXT 20M) )
     SUBPARTITIONS subpartn_a,
   PARTITION pre_yr_2004 VALUES LESS THAN
     TO_DATE('01-JAN-2004', 'DD-MON-YYYY')
       TABLESPACE tblspc02
     LOB (order_desc) STORE AS (TABLESPACE tblspc_a02
       STORAGE (INITIAL 25M NEXT 50M) )
     SUBPARTITIONS subpartn_b TABLESPACE tblspc_x07,
   PARTITION post_yr_2004 VALUES LESS THAN
     MAXVALUE (SUBPARTITION subpartn_1,
       SUBPARTITION subpartn_2,
       SUBPARTITION subpartn_3
       SUBPARTITION subpartn_4) )
ENABLE ROW MOVEMENT;
```

В этом, чуть более сложном примере мы вновь создаем таблицу **orders**, добавив в нее столбец **orded_desc** тип *NCLOB*. Все данные (кроме столбца типа *NCLOB*) секций **pre_yr_2000** и **pre_yr_2004** мы помещаем в табличные пространства **tblspc01** и t**blspc02**, соответственно. А значения столбца **order_desc** мы будем хранить в табличных пространствах **tblspc_a01** и **tblspc_a02**, с другими параметрами хранения. Обратите внимание, что секция **subpartn_b** секции **pre_yr_2004** также хранится в отдельном табличном пространстве **tblspc_x07**. Наконец, последняя секция **(post_yr_2004)** и ее подсекции хранятся в табличном пространстве, используемом для таблицы **orders** по умолчанию, так как ни для секции, ни для подсекций не используется фраза *TABLESPACE*.

Изменение секционированных таблиц

Все, что касается секционирования и подсекционирования, явно выполняемого в операторе *CREATE TABLE*, может быть изменено уже после создания таблицы. Большинство фраз в этом случае (например, *SUBPARTITION TEMPLATE* или *MAPPING TABLE*) являются повторением этих же фраз, используемых при создании таблицы; описания мы повторять не будем. Изменение секций и подсекций таблицы в Oracle выполняется в соответствии со следующим синтаксисом:

```
ALTER TABLE имя_таблицы
  [MODIFY DEFAULT ATTRIBUTES [FOR PARTITION имя_секции]
    [физические_атрибуты] [атрибуты_хранения]
    [PCTTHRESHOLD целое_число]
    [{ADD OVERFLOW ... | OVERFLOW ...}] [[NO]COMPRESS]
    [{LOB (имя_lob) | VARRAY имя_varray} [(параметры_lob)]]
    [COMPRESS целое_число | NOCOMPRESS]]
  [SET SUBPARTITION TEMPLATE {число_хеш_подсекций |
    (SUBPARTITION имя_подсекции [список_значений]
      [атрибуты_хранения])}]
  [ { SET INTERVAL (выражение) |
    SET SET STORE IN (табличное_пространство[, ...]) } ]
  [MODIFY PARTITION имя_секции
    { [описание_секции] |
    [[REBUILD] UNUSABLE LOCAL INDEXES] |
    [ADD [спецификация_подсекции]] |
    [COALESCE SUBPARTITION [[NO]PARALLEL]
      [обновление_индексов]] |
    [{ADD | DROP} VALUES (значения_секции[, ...])] |
    [MAPPING TABLE {ALLOCATE EXTENT ... |
      DEALLOCATE UNUSED ...}] }
  [MODIFY SUBPARTITION имя_подсекции
    {параметры_хеш_подсекции |
    параметры_подсекции_по_списку_значений}]
  [MOVE {PARTITION | SUBPARTITION} имя_секции
    [MAPPING TABLE] [описание_секции] [[NO]PARALLEL]
    [обновление_индексов]]
  [ADD PARTITION [имя_секции] [описание_секции]
    [[NO]PARALLEL] [обновление_индексов]]
  [COALESCE PARTITION [[NO]PARALLEL] [обновление_индексов]]
  [DROP {PARTITION | SUBPARTITION} имя_секции [[NO]PARALLEL]
    [обновление_индексов]]
  [RENAME {PARTITION | SUBPARTITION} имя_секции TO
    новое_имя_секции]
  [TRUNCATE {PARTITION | SUBPARTITION} имя_секции
    [{DROP | REUSE} STORAGE] [[NO]PARALLEL]
    [обновление_индексов]]
  [SPLIT {PARTITION | SUBPARTITION} имя_секции
    {AT | VALUES} (значение[, ...])
    [INTO (PARTITION [имя_секции1] [описание_секции],
      PARTITION [имя_секции2] [описание_секции])]
    [[NO]PARALLEL] [обновление_индексов]]
  [MERGE {PARTITION | SUBPARTITION} имя_секции1, имя_секции2
    [INTO PARTITION [имя_секции] [параметры_секции]]
    [[NO]PARALLEL] [обновление_индексов]]
  [EXCHANGE {PARTITION | SUBPARTITION} имя_секции
    WITH TABLE имя_таблицы
    [{INCLUDING | EXCLUDING} INDEXES]
    [{WITH | WITHOUT} VALIDATION]
    [[NO]PARALLEL] [обновление_индексов]
    [EXCEPTIONS INTO имя_таблицы]]
```

где:

MODIFY DEFAULT ATTRIBUTES [FOR PARTITION имя_секции]

Изменяет параметры текущей или указанной секции. Описание параметров секций приводится в предыдущем разделе.

SET SUBPARTITION TEMPLATE {число_хеш_подсекций | (SUBPARTITION имя_подсекции [список_значений] [атрибуты_хранения])}

Устанавливает для таблицы новый шаблон подсекционирования.

SET INTERVAL (выражение)|SET STORE IN (табличное_пространство[, ...])

Преобразует таблицу, секционированную по диапазону значений, в таблицу, секционированную по интервалам. Либо с помощью фразы *SET STORE IN* меняет табличное пространство для интервально-секционированной таблицы. Вы можете преобразовать интервально-секционированную таблицу обратно в таблицу, секционированную по диапазонам с помощью синтаксиса *SET INTERVAL( )*.

MODIFY PARTITION имя_секции

Меняет для указанной секции набор физических атрибутов и атрибутов хранения, включая атрибуты хранения столбцов *LOB* и *ARRAY*. Для *MODIFY PARTITION* используется следующий дополнительный синтаксис:

```
{ [описание_секции] | [[REBUILD] UNUSABLE LOCAL INDEXES]
  | [ADD [спецификация_подсекции]] |
  [COALESCE SUBPARTITION [[NO]PARALLEL]
    [обновление_индексов]
  { [{UPDATE | INVALIDATE} GLOBAL INDEXES] |
    UPDATE INDEXES [ (имя_индекса (
   {секция_индекса | подсекция_индекса} ))[, ...] ] } ] |
  [{ADD | DROP} VALUES (значение_секции[, ...])] |
  [MAPPING TABLE
    {ALLOCATE EXTENT ... | DEALLOCATE UNUSED ...}] }
```

где:

описание_секции

Приводится в предыдущем разделе «Секционирование таблиц в Oracle». Эта фраза может использоваться с любыми секционированными таблицами.

[REBUILD] UNUSABLE LOCAL INDEXES

Помечает локальную секцию индекса как неиспользуемую (*UNUSABLE*). При добавлении слова *REBUILD* выполняется перестроение неиспользуемых секций локальных индексов. Эту фразу нельзя использовать одновременно с любой другой подфразой оператора *MODIFY PARTITION*, равно как и нельзя использовать ее для таблиц с подсекциями, но можно использовать для любых секционированных таблиц.

ADD [спецификация_подсекции]

Добавляет к таблице, секционированной по диапазонам значений, подсекцию на основе хеш-значений или списков значений. Синтаксис описан в предыдущем разделе. Эта фраза используется только для составного секционирования. Oracle заполняет новую подсекцию строками из других

подсекций, используя либо хеш-значения, либо явно заданный список значений. Для оптимального распределения нагрузки мы рекомендуем, чтобы число подсекций было степенью двойки. Вы можете добавить подсекцию со списком значений, только если в таблице нет подсекции по умолчанию (*DEFAULT*). При этом указание списка значений является обязательным, и эти значения не должны встречаться в спецификации любой другой подсекции. При добавлении подсекции вы не можете указать какие-либо атрибуты, кроме фразы *TABLESPACE*. Добавляя фразу *DEPENDENT TABLES (имя_таблицы (спецификация_секции[, ...])[, ...]) [{UPDATE | INVALIDATE} [GLOBAL] INDEXES (имя_индекса (секция_индекса)[, ...]))]*, вы указываете, что операцию изменения секционирования нужно каскадно распространить на все ссылочно-секционированные таблицы (и их индексы), ссылающиеся на текущую таблицу.

COALESCE SUBPARTITION [[NO]PARALLEL] [обновление_индексов]

Объединяет хеш-подсекции таблицы с составным секционированием. При использовании этой фразы Oracle распределяет строки последней хеш-подсекции по остальным подсекциям, а затем эта подсекция удаляется и удаляются соответствующие ей локальные индексы. Фраза *обновление_индексов* описывается позже в этом списке. Глобальные индексы можно обновлять или объявлять недействительными, используя синтаксис *{UPDATE | INVALIDATE} GLOBAL INDEXES*. Также локальные индексы, секции и подсекции индексов можно обновлять, используя синтаксис *UPDATE INDEXES (имя_индекса ({секция_индекса | подсекция_индекса}))*.

{ADD | DROP} VALUES (значение [, ...])

Добавляет одно или несколько значений либо удаляет существующие значения из списка, по которому формируется секция таблицы. Локальные и глобальные индексы этой операцией не затрагиваются. Значения нельзя добавлять или удалять из *DEFAULT*-секции.

MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSED ...}

Создает таблицу соответствия идентификаторов строк для секционированной индексно-организованной таблицы. Фразы *ALLOCATE EXTENT* и *DEALLOCATE UNUSED* уже были рассмотрены в описании синтаксиса оператора *CREATE TABLE*. Эту фразу можно использовать для индексно-организованных таблиц, секционированных любым образом.

MODIFY SUBPARTITION имя_подсекции {параметры_хеш_подсекции | параметры_подсекции_по_списку_значений }

Изменяет указанную подсекцию, устанавливая параметры подсекции, рассмотренные в предыдущем разделе.

MOVE {PARTITION | SUBPARTITION} имя_секции [MAPPING TABLE] [описание_секции] [[NO]PARALLEL] [обновление_индексов]

Преобразует указанную секцию (или подсекцию) в новую секцию (или подсекцию) с указанным описанием. Преобразование секций требует интенсивных операций ввода-вывода, поэтому можно использовать фразу *PARALLEL* для распараллеливания выполнения преобразования. По умолчанию используется *NOPARALLEL*. Также вы можете обновить или объявить недействи-

тельными локальные и глобальные индексы при помощи описанной далее фразы *обновление_индексов*.

ADD PARTITION [имя_секции] [описание_секции] [[NO]PARALLEL] [обновление_индексов]

Добавляет в таблицу новую секцию или подсекцию. *ADD PARTITION* позволяет контролировать все нюансы создания секции или подсекции во фразе *описание_секции*. Создание секции может потребовать интенсивного ввода-вывода, поэтому для распараллеливания выполнения операции можно использовать опцию *PARALLEL*. По умолчанию используется *NOPARALLEL*. Также вы можете обновить или объявить недействительными локальные и глобальные индексы при помощи фразы *обновление_индексов*.

обновление_индексов

Контролирует состояние индекса при изменении секций и подсекций таблицы. По умолчанию Oracle объявляет недействительными индекс(ы) таблицы целиком, а не только те части, которые затрагиваются операцией изменения секционирования. Вы можете обновить или объявить недействительными глобальные индексы таблицы или обновить один или несколько индексов таблицы, используя следующий синтаксис:

```
[{UPDATE | INVALIDATE} GLOBAL INDEXES] |
UPDATE INDEXES [ ( имя_индекса (
    {секция_индекса|подсекция_индекса} ))[, ...] ]
```

COALESCE PARTITION [[NO]PARALLEL] [обновление_индексов]

Распределяет содержимое последней хеш-секции по всем остальным секциям, а затем удаляет ее. Очевидно, что эта фраза используется только для таблиц, секционированных по хеш-значениям. Фраза *обновление_индексов* может быть использована, чтобы обновить или объявить недействительными локальные и/или глобальные индексы для изменяемой таблицы.

DROP {PARTITION | SUBPARTITION} имя_секции [[NO]PARALLEL] [обновление_индексов]

Удаляет существующую секцию или подсекцию таблицы. Данные внутри секции также удаляются. Если вам требуется сохранить данные, то используйте фразу *MERGE PARTITION*. Если вы хотите избавиться от хеш-секции или подсекции, то используйте фразу *COLASCE PARTITION*. Фраза не оказывает действия на таблицы, содержащие только одну секцию; вместо этого используйте оператор *DROP TABLE*.

RENAME {PARTITION | SUBPARTITION} имя_секции TO новое_имя_секции

Переименовывает существующую секцию или подсекцию.

TRUNCATE {PARTITION | SUBPARTITION} имя_секции [{DROP | REUSE} STORAGE] [[NO]PARALLEL] [обновление_индексов]

Удаляет все строки указанной секции или подсекции. При очистке составной секции строки из всех подсекции также удаляются. В индексно-организованных таблицах одновременно очищаются секции таблицы соответствия идентификаторов строк и секции сегмента переполнения. Если в таблице есть столбцы типа *LOB*, то *LOB*-сегменты тоже очищаются. Перед очисткой требу-

ется либо отключить ограничения ссылочной целостности, либо предварительно удалить строки, а затем только удалять секции. Необязательные фразы *DROP* и *REUSE STORAGE* указывают, нужно ли после очистки освободить дисковое пространство и сделать его доступным для других объектов (*DROP*) или нужно оставить его для очищенной секции.

SPLIT {PARTITION | SUBPARTITION} имя_секции {AT | VALUES} (значение[, ...]) [INTO (PARTITION [имя_секции1] [описание_секции]), (PARTITION [имя_секции2] [описание_секции])] [[NO]PARALLEL] [обновление_индексов]

Создает из одной существующей секции (или подсекции) две новых секции (или подсекции). Новые секции могут иметь новые описания либо наследовать параметры исходной секции. При расщеплении секции *DEFAULT* все указанные значения попадают в первую секцию, а все значения по умолчанию попадают во вторую секцию. При расщеплении индексно-организованной таблицы аналогичным образом автоматически расщепляются таблицы соответствия идентификаторов строк. Oracle также расщепляет *LOB*-сегменты и сегменты переполнения, при этом вы можете указать для создаваемых сегментов новые атрибуты хранения.

{AT | VALUES} (значение[, ...])

Расщепляет диапазон значений секции (*AT*) или список значений секции (*VALUES*) в соответствии с указанным значением (или списком значений). *AT* определяет новую невключаемую верхнюю границу первой секции. Новое значение должно быть меньше, чем верхняя граница исходной секции, но больше верхней границы предыдущей секции (если такая существует). *VALUES* определяет список значений, которые попадают в первую из двух секций, а во вторую секцию попадают все оставшиеся значения исходной секции. Список значений *VALUES* должен включать значения из списка значений исходной секции, но при этом не может включать все значения.

[INTO (PARTITION [имя_секции1] [описание_секции]), (PARTITION [имя_секции2] [описание_секции])]

Определяет две новых секции, получающихся в результате расщепления. Как минимум, в скобках должны быть указаны два ключевых слова *PARTITION*. Любые параметры, явно не переопределенные для новых секций, наследуются из исходной секции, включая подсекционирование. Заметьте, что имеется несколько ограничений. Если используется составное секционирование по диапазонам значений и по хеш-значениям (range-hash), то для подсекций можно указать только табличное пространство. Указание подсекционирования вообще не допускается при расщеплении таблиц, секционированных составным образом по диапазонам и по спискам значений (range-list). Любые индексы при расщеплении таблицы по умолчанию объявляются недействительными, для обновления их состояния нужно использовать фразу *обновление_индексов*.

MERGE {PARTITION | SUBPARTITION} имя_секции1, имя_секции2 [INTO PARTITION [имя_секции] [атрибуты_секции]] [[NO]PARALLEL] [обновление_индексов]

Сливает содержимое двух секций (или подсекций) в одну новую секцию (подсекцию). Исходные секции после этого удаляются. Если сливаются секции

таблицы, секционированной по диапазону, то секции должны быть смежными и верхняя граница новой секции будет совпадать с наибольшей из границ исходных секций. При слиянии секций, построенных по спискам значений, список значений новой секции будет включать все значения исходных секций. Если одна из секций является секцией по умолчанию (*DEFAULT*), то созданная секция также будет секцией по умолчанию. Слияние составных секций по диапазону и по списку значений (range-list) допустимо, но при этом нельзя указывать новый шаблон подсекционирования. Oracle создает шаблон подсекционирования автоматически из существующих либо создает одну подсекцию по умолчанию. Не переопределенные явно физические атрибуты наследуются из определения таблицы. По умолчанию глобальные индексы и секции локальных индексов помечаются как неиспользуемые, если только явно не определено другое поведение при помощи фразы *обновление_индексов*. (Исключение составляют индексно-организованные таблицы, которые, являясь индексами сами по себе, остаются в состоянии *USABLE* после слияния секций.) Слияние нельзя использовать для таблиц, секционированных по хеш-значениям; вместо этого используйте фразу *COLESCE PARTITION*.

EXCHANGE {PARTITION | SUBPARTITION} имя_секции WITH TABLE имя_таблицы [{INCLUDING|EXCLUDING} INDEXES] [{WITH|WITHOUT} VALIDATION] [[NO]PARALLEL] [обновление_индексов] [EXCEPTIONS INTO имя_таблицы]

Меняет местами сегменты данных и индексов секции с сегментами несекционированной таблицы либо меняет местами сегменты секций двух таблиц, секционированных различным образом. Структура таблиц при этой операции должна быть идентичной, включая одинаковый первичный ключ. Все атрибуты сегментов (табличное пространство, журналирование, статистика и т. д.) также меняются местами. Если в таблицах есть столбцы типа *LOB*, то меняются местами *LOB*-сегменты. Синтаксические элементы этой команды имеют следующие значения:

WITH TABLE имя_таблицы

Указывает таблицу, которая меняется сегментами с секцией или подсекцией.

{INCLUDING|EXCLUDING} INDEXES

INCLUDING INDEXES обменивает секции или подсекции локальных индексов с индексами таблицы. Если используется *EXCLUDING INDEXES*, то переводит секции и подсекции локальных индексов и индексы обычной таблицы в статус *UNUSABLE*.

{WITH|WITHOUT} VALIDATION

WITH VALIDATION проверяет, что все строки исходной таблицы попадают в секцию, с которой производится обмен, в противном случае возникает ошибка. Если используется *WITHOUT VALIDATION*, то проверка соответствия значений строк условию секционирования не выполняется.

EXCEPTIONS INTO имя_таблицы

Помещает *ROWID* строк, нарушающих ограничение уникальности (в состоянии *DISABLE VALIDATE*) секционированной таблицы, в указанную таблицу. По умолчанию Oracle будет записывать *ROWID* ошибочных строк в таблицу *EXCEPTIONS* в текущей схеме. Таблицу *EXCEPTIONS*

можно создать с помощью скриптов *utlexcpt.sql* и *utlexcp1.sql*, поставляемых с Oracle.

Нужно помнить о нескольких моментах, касающихся изменения секционирования таблиц. Во-первых, если изменяемая таблица является источником для материализованных представлений, то эти материализованные представления требуется обновлять. Во-вторых, все битовые индексы соединения (bitmap join indexes) секционированной таблицы переводятся в статус *UNUSABLE*. В-третьих, есть некоторые ограничения, касающиеся ситуаций, когда секция или подсекция распределена по нескольким табличным пространствам, использующим разный размер блока данных. При выполнении такого рода изменений секционированных таблиц обращайтесь к документации Oracle.

Для последующих примеров создадим таблицу **orders**, секционированную следующим образом:

```
CREATE TABLE orders
  (order_number NUMBER,
  order_date    DATE,
  cust_nbr      NUMBER,
  price         NUMBER,
  qty           NUMBER,
  cust_shp_id   NUMBER)
PARTITION BY RANGE(order_date)
 (PARTITION pre_yr_2000 VALUES LESS THAN
    TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
  PARTITION pre_yr_2004 VALUES LESS THAN
    TO_DATE('01-JAN-2004', 'DD-MON-YYYY'
  PARTITION post_yr_2004 VALUES LESS THAN
    MAXVALUE) ) ;
```

Следующий оператор сделает все локальные индексы таблицы **orders** в секции **post_yr_2004** неиспользуемыми:

```
ALTER TABLE orders MODIFY PARTITION post_yr_2004
  UNUSABLE LOCAL INDEXES;
```

Предположим, мы решили расщепить последнюю секцию **post_yr_2004** на две секции – **pre_yr_2008** и **post_yr_2008**. Все значения меньше 1 января 2008 года будем хранить в секции **pre_yr_2008**, а все остальные значения, вплоть до максимального, будем хранить в секции **post_yr_2008**:

```
ALTER TABLE orders SPLIT PARITION post_yr_2004
  AT (TO_DATE('01-JAN-2008','DD-MON-YYYY'))
  INTO (PARTITION pre_yr_2008, PARTITION post_yr_2008);
```

Если предположить, что в таблице **orders** есть столбцы типов *LOB* или *VARRAY*, то мы могли бы указать дополнительные параметры для хранения этих столбцов, и, например, в конце обновить глобальные индексы:

```
ALTER TABLE orders SPLIT PARITION post_yr_2004
    AT (TO_DATE('01-JAN-2008','DD-MON-YYYY'))
INTO
 (PARTITION pre_yr_2008
    LOB (order_desc) STORE AS (TABLESPACE order_tblspc_a1),
  PARTITION post_yr_2008)
```

```
    LOB (order_desc) STORE AS (TABLESPACE order_tblspc_a1) )
  UPDATE GLOBAL INDEXES;
```

Теперь мы можем объединить секции с противоположного конца таблицы:

```
ALTER TABLE orders
MERGE PARTITIONS pre_yr_2000, pre_yr_2004
INTO PARTITION yrs_2004_and_earlier;
```

Прошло несколько лет, мы хотим избавиться от старых секций или хотя бы переименовать их:

```
ALTER TABLE orders DROP PARTITION yrs_2004_and_earlier;
ALTER TABLE orders RENAME PARTITION yrs_2004_and_earlier
  TO pre_yr_2004;
```

Наконец, давайте очистим все данные секции и освободим используемое ею пространство:

```
ALTER TABLE orders
  TRUNCATE PARTITION pre_yr_2004
  DROP STORAGE;
```

Как показывают эти примеры, все, что касается секционирования и создания подсекций, может быть сделано в операторе *CREATE TABLE* и затем изменено при помощи оператора *ALTER TABLE*.

Организация таблиц: традиционные таблицы, индексно-организованные таблицы и внешние таблицы

В Oracle 11*g* существуют мощные средства управления физическим хранением таблиц.

Наиболее важным аспектом фразы *ORGANIZATION HEAP* является то, что она позволяет сжимать всю таблицу. Это особенно актуально для снижения стоимости хранения баз данных с многотерабайтными таблицами. В следующем примере создается таблица **orders**, для которой используется сжатие, изменения журналируются, создается первичный ключ и указываются параметры хранения:

```
CREATE TABLE orders
 (order_number NUMBER,
  order_date   DATE,
  cust_nbr     NUMBER,
  price        NUMBER,
  qty          NUMBER,
  cust_shp_id  NUMBER,
  shp_region   VARCHAR2(20),
  order_desc   NCLOB,
CONSTRAINT ord_nbr_pk PRIMARY KEY (order_number) )
ORGANIZATION HEAP
COMPRESS
LOGGING
PCTTHRESHOLD 2
STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0
   MINEXTENTS 1 MAXEXTENTS 1)
OVERFLOW STORAGE
```

```
  (INITIAL 4M NEXT 2M PCTINCREASE 0
   MINEXTENTS 1 MAXEXTENTS 1)
ENABLE ROW MOVEMENT;
```

Для создания той же самой таблицы, но с организацией по индексу на базе столбца **order_date** мы можем использовать следующий синтаксис:

```
CREATE TABLE orders
 (order_number NUMBER,
  order_date   DATE,
  cust_nbr     NUMBER,
  price        NUMBER,
  qty          NUMBER,
  cust_shp_id  NUMBER,
  shp_region   VARCHAR2(20),
  order_desc   NCLOB,
CONSTRAINT ord_nbr_pk PRIMARY KEY (order_number) )
ORGANIZATION INDEX
  INCLUDING order_date
PCTTHRESHOLD 2
STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0
   MINEXTENTS 1 MAXEXTENTS 1)
OVERFLOW STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0
   MINEXTENTS 1 MAXEXTENTS 1)
ENABLE ROW MOVEMENT;
```

Наконец, создадим внешнюю таблицу **cust_shipping_external**, содержащую информацию по поставкам:

```
CREATE TABLE cust_shipping_external
 (external_cust_nbr NUMBER(6),
  cust_shp_id       NUMBER,
  shipping_company  VARCHAR2(25) )
ORGANIZATION EXTERNAL
 (TYPE oracle_loader
  DEFAULT DIRECTORY dataloader
  ACCESS PARAMETERS
    (RECORDS DELIMITED BY newline
     BADFILE 'upload_shipping.bad'
     DISCARDFILE 'upload_shipping.dis'
     LOGFILE 'upload_shipping.log'
     SKIP 20
     FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"'
       (client_id INTEGER EXTERNAL(6),
        shp_id CHAR(20),
        shipper CHAR(25) ) )
  LOCATION ('upload_shipping.ctl') )
REJECT LIMIT UNLIMITED;
```

В этом примере внешняя таблица хранится в формате *ORACLE_LOADER* и по умолчанию используется каталог *DATALOADER*. Этот пример иллюстрирует, что сначала внутри Oracle описываются метаданные таблицы, а затем указывается, как эти метаданные соотносятся с данными, хранящимися вне Oracle.

Тип XMLType и объектные таблицы в Oracle

При создании XMLType-таблицы Oracle автоматически сохраняет данные в столбце типа *CLOB*, если вы только явно не создадите таблицу для хранения XML по определенной схеме. В следующем примере сначала создается таблица **distributors** с хранением данных в столбце *CLOB* (*формат хранения не указан, поэтому используется CLOB*), а затем создается таблица **suppliers** с более сложным определением, включающим XML-схему:

```
CREATE TABLE distributors OF XMLTYPE;
CREATE TABLE suppliers OF XMLTYPE
XMLSCHEMA "http://www.lookatallthisstuff.com/suppliers.xsd"
ELEMENT "vendors";
```

Ключевым преимуществом хранения данных XML в таблицах, основанных на XML-схемах, является то, что по таким таблицам можно создавать B-tree индексы. В следующем примере мы создадим индекс по атрибуту **supplier.city**:

```
CREATE INDEX suppliercity_index
ON suppliers
(S."XMLDATA"."ADDRESS"."CITY");
```

Похожим образом можно создавать таблицы, в которых используются как столбцы типа *XMLTYPE*, так и обычные столбцы. В этом случае *XMLTYPE* можно также хранить и в *CLOB*, и в объектно-реляционной таблице, созданной по XML-схеме. Для примера мы еще раз создадим таблицы **distributors** (добавив некоторые характеристики хранения) и **suppliers** с обычным столбцом и столбцом *XMLTYPE*:

```
CREATE TABLE distributors
 (distributor_id   NUMBER,
  distributor_spec XMLTYPE)
XMLTYPE distributor_spec
STORE AS CLOB
  (TABLESPACE tblspc_dist
   STORAGE (INITIAL 10M NEXT 5M)
   CHUNK 4000
   NOCACHE
   LOGGING);

CREATE TABLE suppliers
 (supplier_id   NUMBER,
  supplier_spec XMLTYPE)
XMLTYPE supplier_spec STORE AS OBJECT RELATIONAL
 XMLSCHEMA "http://www.lookatallthisstuff.com/suppliers.xsd"
 ELEMENT "vendors"
OBJECT IDENTIFIER IS SYSTEM GENERATED
OIDINDEX vendor_ndx TABLESPACE tblspc_xml_vendors;
```

При создании объектных таблиц и таблиц с *XMLType* вы можете использовать фразы *ссылочное_ограничение_столбца* и *ссылочное_ограничение_таблицы*. Отличие между ними состоит только в том, что в первом случае создается ограничение на уровне столбца, а во втором случае создается ограничение на уровне таблицы. (Эта разница в поведении и использовании аналогична той, что есть и для обычных

реляционных ограничений, например *PRIMARY KEY* или *FOREIGN KEY*.) Синтаксис фразы *ссылочное_ограничение_столбца* следующий:

```
{SCOPE IS таблица_видимости |
  WITH ROWID |
    [CONSTRAINT имя_ограничения]
    REFERENCES объект [    (имя_столбца) ]
    [ON DELETE {CASCADE | SET NULL}]
    [состояние_ограничения]}
```

Синтаксис фразы *ссылочное_ограничение_таблицы* следующий:

```
{SCOPE FOR (ссылающийся_столбец | ссылающийся_атрибут)
  IS таблица_видимости |
  REF (ссылающийся_столбец|ссылающийся_атрибут) WITH ROWID |
  [CONSTRAINT имя_ограничения]
  FOREIGN KEY   (ссылающийся_столбец | ссылающийся_атрибут)
  REFERENCES объект [ (имя_столбца) ]
  [ON DELETE {CASCADE | SET NULL}]
  [состояние_ограничения]}
```

Фраза *состояние_ограничения* содержит набор параметров, уже описанных ранее при рассмотрении синтаксиса оператора *CREATE TABLE*. К ссылочным ограничениям относятся следующие опции:

```
[NOT] DEFERRABLE
INITIALLY {IMMEDIATE | DEFERRED}
{ENABLE | DISABLE}
{VALIDATE | NOVALIDATE}
{RELY | NORELY}
EXCEPTIONS INTO имя_таблицы
USING INDEX {имя_индекса | ( оператор_create_index ) |
  параметры_индекса}
```

Объектные таблицы – это таблицы, созданные на основе пользовательских типов данных. Например, следующий фрагмент кода создает тип *building_type*:

```
CREATE TYPE OR REPLACE building_type AS OBJECT
 (building_name    VARCHAR2(100),
  building_address VARCHAR2(200));
```

Теперь мы создадим таблицу **offices_object_table**, содержащую объектный тип и описывающую некоторые дополнительные параметры, например способ генерации OID. В дополнение мы создадим еще две таблицы, использующие *building_type*, с ограничениями на уровне столбца и на уровне таблицы:

```
CREATE TABLE offices_object_table
  OF building_type (building_name PRIMARY KEY)
OBJECT IDENTIFIER IS PRIMARY KEY;
CREATE TABLE leased_offices
 (office_nbr NUMBER,
  rent       DEC(9,3),
  office_ref REF building_type
    SCOPE IS offices_object_table);
CREATE TABLE owned_offices
 (office_nbr NUMBER,
```

```
  payment    DEC(9,3),
  office_ref REF building_type
  CONSTRAINT offc_in_bld REFERENCES offices_object_table);
```

В этом примере *SCOPE IS* создает ограничение на уровне столбца, а *CONSTRAINT* – на уровне таблицы.

Оператор ALTER TABLE в Oracle

При использовании оператора *ALTER TABLE* вы можете добавить (*ADD*), удалить (*DROP*) или модифицировать (*MODIFY*) любой элемент таблицы. Например, на синтаксической диаграмме показано, что при добавлении или изменении столбца указываются его атрибуты, причем это все возможные атрибуты, включая нестандартные расширения Oracle. Таким образом, ANSI позволяет модифицировать только такие атрибуты, как *DEFAULT* и *NOT NULL* (и ограничения на уровне столбца), а в Oracle можно изменять любые существующие параметры таблицы, включая настройки *LOB*, *VARRAY*, *NESTED TABLE*, *CLUSTER* и *PARTITION*, *а также настройки* индексно-организованных таблиц.

Например, следующий код добавляет в таблицу новый столбец и уникальное ограничение целостности:

```
ALTER TABLE titles
ADD subtitle VARCHAR2(32) NULL
CONSTRAINT unq_subtitle UNIQUE;
```

При добавлении ограничения СУБД проверяет все строки на соответствие задаваемым правилам; если обнаруживаются строки, не удовлетворяющие добавляемому ограничению, то оператор *ALTER TABLE* завершается с ошибкой.

Все запросы, использующие *SELECT* *, будут возвращать новые столбцы, даже если это и не планировалось. Скомпилированные объекты, такие как хранимые процедуры, будут возвращать новые столбцы, если они используют атрибут %*ROWTYPE*. В противном случае скомпилированные объекты могут не возвращать новые столбцы.

Oracle также позволяет выполнить несколько различных действий, таких как *ADD* и *MODIFY*, если они перечислены в скобках через запятую. В следующем примере одним оператором в таблице добавляется несколько столбцов:

```
ALTER TABLE titles
ADD (subtitles VARCHAR2(32) NULL,
  year_of_copyright INT,
  date_of_origin DATE);
```

PostgreSQL

PostgreSQL поддерживает стандарт ANSI для операторов *CREATE* и *ALTER TABLE*, а также некоторые расширения, позволяющие быстро создавать новую таблицу из существующей. Синтаксис оператора *CREATE TABLE* следующий:

```
CREATE [LOCAL] [[TEMP]ORARY] TABLE имя_таблицы
  (имя_столбца тип_данных атрибуты[, ...]) |
CONSTRAINT имя_ограничения [{NULL | NOT NULL}]
```

```
  {[UNIQUE] | [PRIMARY KEY (имя_столбца[, ...])] |
  [CHECK (выражение)] |
  REFERENCES таблица (столбец[, ...])
    [MATCH {FULL | PARTIAL | default}]
    [ON {UPDATE | DELETE}
    {CASCADE | NO ACTION | RESTRICT | SET NULL |
     SET DEFAULT значение}]
  [[NOT] DEFERRABLE] [INITIALLY {DEFERRED | IMMEDIATE}]}
     [, ...] |
[ограничение_на_уровне_таблицы][, ...]
[WITH[OUT] OIDE]
[INHERITS (наследуемая_таблица[, ...])]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[AS оператор_select]
```

Для оператора *ALTER TABLE* в PostgreSQL используется следующий синтаксис:

```
ALTER TABLE [ONLY] имя_таблицы [*]
[ADD [COLUMN] столбец тип_данных атрибуты [...]]
| [ALTER [COLUMN] имя_столбца
{SET DEFAULT значение | DROP DEFAULT | SET STATISTICS целое_число}]
| [RENAME [COLUMN] имя_столбца TO новое_имя_столбца]
| [RENAME TO новое_имя_таблицы]
| [ADD ограничение_на_уровне_таблицы]
| [DROP CONSTRAINT имя_ограничения RESTRICT]
| [OWNER TO новый_владелец]
```

Параметры имеют следующие значения:

REFERENCES...MATCH...ON {UPDATE | DELETE}...

Проверяет значения столбцов вставляемой строки на соответствие значениям в другой таблице. Эта фраза может использоваться также как часть определения внешнего ключа. Опции *MATCH* включают значения *FULL*, *PARTIAL* и *default*. *FULL* проверяет, что либо все столбцы внешнего ключа должны содержать корректные значения, либо все столбцы должны содержать *NULL*. *default* (значение по умолчанию) допускает смешанные корректные значения и *NULL*. *PARTIAL* поддерживается только на уровне синтаксиса. Во фразе *REFERENCES* также можно указать дополнительные параметры поведения при удалении (*ON DELETE*) и обновлении (*ON UPDATE*):

NO ACTION

При нарушении внешнего ключа возникает ошибка (значение по умолчанию).

RESTRICT

Синоним для *NO ACTION*

CASCADE

Проставляет в столбцах ссылающейся записи новые значения из обновленной родительской записи.

SET NULL

Проставляет NULL в столбцах ссылающейся записи.

SET DEFAULT значение

Проставляет в столбцах ссылающейся записи значение по умолчанию либо NULL, если значение по умолчанию не указано.

[NOT] DEFERRABLE [INITIALLY {DEFERRED|IMMEDIATE}]

Опция *DEFERRABLE* указывает, что проверка ограничения может быть отложена до конца транзакции. По умолчанию используется *NOT DEFERRABLE*. *INITIALLY DEFERRED* создает ограничение с изначально отложенной проверкой, *INITIALLY IMMEDIATE (поведение по умолчанию)* создает ограничение, которое проверяется после каждого оператора.

FOREIGN KEY

Внешний ключ, создаваемый не на уровне столбца, а на уровне таблицы. Полностью поддерживает описанную выше фразу *REFERENCES*. Синтаксис следующий:

```
[FOREIGN KEY (имя_столбца[, ...]) REFERENCES...]
```

WITH[OUT] OIDS

При использовании параметра *WITH OIDS* каждой строке новой таблицы будет присваиваться автоматически генерируемый объектный идентификатор. При использовании *WITHOUT OIDS* идентификаторы не генерируются. Для идентификаторов используется 32-битный счетчик. При достижении максимального значения счетчик обнуляется и дальнейшая уникальность не гарантируется. Мы рекомендуем строить уникальный индекс по столбцу *OID* для защиты от дубликатов при очень больших таблицах. Таблицы с наследуемой структурой используют то же значение параметра *OIDS*, что и родительская таблица.

INHERITS наследуемая_таблица

Определяет одну или несколько таблиц, из которых создаваемая таблица наследует все столбцы. Создаваемая таблица также наследует все функции, связанные с любой таблицей, находящейся выше по иерархии. Если какой-либо наследуемый столбец появляется больше одного раза, то оператор завершается с ошибкой.

ON COMMIT {DELETE|PRESERVE} ROWS

Используется только для временных таблиц. С помощью этого параметра определяется поведение временных таблиц при фиксации транзакции. *ON COMMIT DELETE ROWS* очищает временную таблицу при каждом *COMMIT*. (Это значение по умолчанию.) *ON COMMIT PRESERVE ROWS* сохраняет строки во временной таблице после завершения транзакции.

AS оператор_select

Позволяет создать таблицу и наполнить данными из оператора *SELECT*. Имена и типы столбцов при этом описывать не нужно, так как они наследуются из запроса. Оператор *CREATE TABLE...AS SELECT* по функциональности похож на *SELECT...INTO*, но легче читаем.

ONLY

Указывает, что оператор *ALTER* должен модифицировать только указанную таблицу и не должен затрагивать какие-либо родительские таблицы или таблицы-потомки.

OWNER TO новый_владелец

Устанавливает для таблицы нового владельца.

В PostgreSQL таблица может иметь до 1600 столбцов. Но на практике число столбцов должно быть сильно меньше 1600 по причинам производительности. Вот пример создания таблицы:

```
CREATE TABLE distributors
  (name VARCHAR(40) DEFAULT 'Thomas Nash Distributors',
   dist_id INTEGER DEFAULT NEXTVAL('dist_serial'),
   modtime TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

Уникальной особенностью PostgreSQL является возможность создавать на уровне столбца ограничения, использующие несколько столбцов. Но так как PostgreSQL поддерживает создание ограничений на уровне таблицы по стандарту ANSI, то этим подходом и рекомендуется пользоваться.

Реализация оператора *ALTER TABLE* в PostgreSQL позволяет добавлять в таблицу столбцы с помощью ключевого слова *ADD*. Для существующих столбцов можно устанавливать или удалять значение по умолчанию, используя, соответственно, *ALTER COLUMN...SET DEFAULT* и *ALTER COLUMN...DROP DEFAULT*. При изменении значения по умолчанию для столбца новое значение будет использовано только для новых записей, вставляемых в таблицу. *RENAME* позволяет переименовывать таблицы и отдельные столбцы.

SQL Server

SQL Server поддерживает широкий набор опций при объявлении и изменении таблицы, ее столбцов и ограничений. Синтаксис оператора *CREATE TABLE* следующий:

```
CREATE TABLE имя_таблицы
(имя_столбца тип_данных { [DEFAULT значение]
  | [IDENTITY [(начало,шаг) [NOT FOR REPLICATION]]]
  [ROWGUIDCOL] [NULL | NOT NULL]
     | [{PRIMARY KEY | UNIQUE}
     [CLUSTERED | NONCLUSTERED]
     [WITH FILLFACTOR = целое_число]
     [ON {файловая_граппа | DEFAULT}]]
  | [[FOREIGN KEY]
     REFERENCES таблица [(столбец[, ...])]
     [ON {DELETE | UPDATE} {CASCADE | NO ACTION}]
     [NOT FOR REPLICATION]]
  | [CHECK [NOT FOR REPLICATION] (выражение)]
| [COLLATE схема_упорядочения]
| имя_столбца AS вычисляемое_выражение }
[, ...]
```

```
| [ограничение_на_уровне_таблицы][, ...])
[ON {файловая_группа | DEFAULT}]
[TEXTIMAGE_ON { файловая_группа | DEFAULT}]
```

Синтаксис оператора *ALTER TABLE* следующий:

```
ALTER TABLE имя_таблицы
  [ALTER COLUMN имя_столбца тип_данных атрибуты
      {ADD | DROP} ROWGUIDCOL]
  | [ADD [COLUMN] имя_столбца тип_данных атрибуты][, ...]]
  | [WITH CHECK | WITH NOCHECK] ADD
      ограничение_на_уровне_таблицы][, ...]
  | [DROP { [CONSTRAINT] имя_ограничения
      | COLUMN имя_столбца }][, ...]
  | [{CHECK | NOCHECK} CONSTRAINT { ALL | имя_ограничения[, ...] }]
  | [{ENABLE | DISABLE} TRIGGER { ALL | имя_триггера[, ...] }]
```

Параметры имеют следующие значения:

DEFAULT значение

Устанавливает значение по умолчанию для столбцов любого типа, кроме *TIMESTAMP* и *IDENTITY*. Значение по умолчанию должно быть либо константой, либо системной функцией, например *GETDATE( )*, либо NULL.

IDENTITY [(начало, шаг)]

Создает и заполняет целочисленный столбец монотонно возрастающими значениями. По желанию можно указать начальное значение и шаг увеличения. По умолчанию оба значения равны 1.

NOT FOR REPLICATION

Указывает, что значения столбцов *IDENTITY* и *FOREIGN KEY* не реплицируются на подписанные серверы. Эта возможность используется в ситуациях, когда несколько серверов используют одну и ту же структуру таблиц, но не в точности одинаковые данные.

ROWGUIDCOL

Идентифицирует столбец, содержащий глобальные уникальные идентификаторы (GUID), никогда не повторяющиеся на других серверах. В таблице может быть не более одного такого столбца. Идентификаторы не создаются автоматически, а должны генерироваться вызовами функции *NEWID*.

{PRIMARY KEY|UNIQUE}

Создает в таблице первичный или уникальный ключ. Создание первичного ключа отличается от стандарта ANSI тем, что можно указывать, является индекс первичного ключа кластерным или некластерным, а также можно устанавливать начальный процент заполнения страниц индексов. (Информация о первичных ключах приводится в главе 2.) Атрибуты уникальных и первичных ключей следующие:

CLUSTERED|NONCLUSTERED

CLUSTERED указывает, что порядок сортировки первичного ключа определяет физический порядок хранения записей. *NONCLUSTERED* указывает, что индекс не является кластерным и содержит только указатели на строки таблицы. По умолчанию используется *CLUSTERED*.

WITH FILLFACTOR = целое_число

Определяет процент начального заполнения пространства в каждой странице индекса, оставляемого при создании таблицы. SQL Server не поддерживает постоянное значение уровня заполнения страниц, поэтому необходимо регулярно перестраивать индекс.

ON {файловая_группа | DEFAULT}

Создает индекс либо в указанной файловой группе, либо в файловой группе, используемой по умолчанию.

FOREIGN KEY

Проверяет значения столбцов вставляемых строк на соответствие значениям в другой таблице. Внешние ключи детально рассмотрены в главе 2. Внешний ключ может ссылаться только на столбцы, объявленные в другой таблице как *PRIMARY KEY* или *UNIQUE*. Дополнительно можно указать действие, выполняемое при изменении или удалении записи в таблице, на которую ссылается внешний ключ:

ON {DELETE | UPDATE}

Указывает, что действие в локальной таблице нужно выполнять при удалении или обновлении (или при обоих действиях) записи в таблице, на которую ссылается внешний ключ.

CASCADE

Указывает, что обновление или удаление записи в родительской таблице имеет аналогичный эффект на все записи, ссылающиеся на изменяемую строку.

NO ACTION

Указывает, что при обновлении или удалении записи в родительской таблице никакие действия с записями, ссылающимися на нее, не производятся.

NOT FOR REPLICATION

Указывает, что свойство *IDENTITY* не должно применяться для данных, реплицируемых из другого сервера. Это гарантирует, что данным из опубликованного сервера не присваиваются новые идентификаторы.

CHECK

Проверяет, что значения, вставляемые в определенный столбец, удовлетворяют ограничению, заданному выражением. В следующем примере создается таблица с двумя проверочными ограничениями на уровне столбцов:

```
CREATE TABLE people
 (people_id    CHAR(4)
    CONSTRAINT pk_dist_id PRIMARY KEY CLUSTERED
    CONSTRAINT ck_dist_id CHECK (dist_id LIKE '
      [A-Z][A-Z][A-Z][A-Z]'),
  people_name  VARCHAR(40) NULL,
  people_addr1 VARCHAR(40) NULL,
  people_addr2 VARCHAR(40) NULL,
  city         VARCHAR(20) NULL,
  state        CHAR(2) NULL
    CONSTRAINT def_st DEFAULT ("CA")
```

```
    CONSTRAINT chk_st REFERENCES states(state_ID),
  zip          CHAR(5) NULL
    CONSTRAINT ck_dist_zip
    CHECK(zip LIKE '[0-9][0-9][0-9][0-9][0-9]'),
  phone        CHAR(12) NULL,
  sales_rep    empid NOT NULL DEFAULT USER)
GO
```

Ограничение столбца **people_id** гарантирует идентификатор только из букв, а ограничение столбца **zip** проверяет, что значение состоит только из цифр. Внешний ключ на столбце **state** ссылается на таблицу **states**. Внешний ключ в некоторым смысле похож на проверочное ограничение, которое берет список допустимых значений не из выражения, а из другой таблицы. В этом примере также иллюстрируется, как ограничениям присваиваются имена.

COLLATE

Позволяет поменять кодировку и порядок сортировки, используемые в столбце.

TEXTIMAGE_ON {файловая_группа|DEFAULT}

Позволяет разместить данные столбцов типа *text*, *ntext* и *image* в указанной файловой группе либо хранить их в файловой группе, используемой по умолчанию для всех объектов базы данных.

WITH [NO]CHECK

Указывает, нужно ли проверять данные таблицы на соответствие новым ограничениям целостности. Если ограничение добавляется с опцией *NOCHECK*, то оно игнорируется оптимизатором запросов, пока не будет явно включено командой *ALTER TABLE...CHECK CONSTRAINT ALL*. Если ограничение добавляется с опцией *WITH CHECK*, то все строки немедленно проверяются на соответствие ограничению.

CHECK | NOCHECK CONSTRAINT

Включает или отключает существующее ограничение. Если указан параметр *NOCHECK*, то ограничение отключается и будущие вставки или обновления столбца не проверяются относительно условий ограничения.

{ENABLE | DISABLE} TRIGGER {ALL | имя_триггера[, ...] }

Включает (*ENABLE*) или выключает (*DISABLE*) указанные триггеры. Все триггеры таблицы одновременно можно включить или выключить, используя ключевое слово *ALL*, например: *ALTER TABLE employee DISABLE TRIGGER ALL*. Вы можете включить или отключить как один указанный триггер, так и несколько, перечислив их через запятую.

SQL Server позволяет задавать имена для ограничений на уровне столбцов, используя фразу *CONSTRAINT имя_ограничения ...*, и затем текст ограничения. Для одного столбца можно создать несколько ограничений, если только они не являются взаимоисключающими, как например *PRIMARY KEY* и *NULL*.

В SQL Server можно создавать временные таблицы, но при этом используется синтаксис, отличный от ANSI. При создании локальной временной таблицы, хранящейся в базе **tempdb**, требуется начинать имя таблицы с одиночного символа #. Локальная временная таблица используется человеком или процессом,

создавшим его, и удаляется, после того как пользователь отключается или завершается процесс. Имя глобальной временной таблицы должно начинаться с символов ##. Глобальная временная таблица может использоваться любым пользователем, подключенным к базе данных на момент существования этой таблицы. Удаляется глобальная временная таблица также в момент отключения пользователя или процесса, создавшего ее.

SQL Server позволяет создавать таблицы с вычисляемыми столбцами. Такие столбцы фактически не хранятся, а всего лишь содержат выражение, вычисляемое на базе других столбцов таблицы. Например, вычисляемый столбец может содержать выражение *order_cost AS (price*qty)*. Вычисляемые столбцы могут содержать константы, функции, переменные, обычные столбцы, а также любые комбинации этих элементов с использованием операторов.

Любое из показанных ранее ограничений на уровне столбца может быть создано и на уровне таблицы. То есть ограничения *PRIMARY KEY*, *FOREIGN KEY*, *CHECK* могут быть указаны в конце оператора *CREATE TABLE* после списка столбцов. Это используется для ограничений, использующих несколько столбцов. Например, ограничение *UNIQUE* на уровне столбца применяется только к этому столбцу, а ограничение *UNIQUE* на уровне таблицы может охватывать несколько столбцов. Вот пример обоих типов ограничений:

```
-- Создание ограничения на уровне столбца
CREATE TABLE favorite_books
 (isbn          CHAR(100) PRIMARY KEY NONCLUSTERED,
  book_name     VARCHAR(40) UNIQUE,
  category      VARCHAR(40) NULL,
  subcategory   VARCHAR(40) NULL,
  pub_date      DATETIME NOT NULL,
  purchase_date DATETIME NOT NULL)
GO
-- Создание ограничения на уровне таблицы
CREATE TABLE favorite_books
 (isbn          CHAR(100) NOT NULL,
  book_name     VARCHAR(40) NOT NULL,
  category      VARCHAR(40) NULL,
  subcategory   VARCHAR(40) NULL,
  pub_date      DATETIME NOT NULL,
  purchase_date DATETIME NOT NULL,
    CONSTRAINT pk_book_id PRIMARY KEY NONCLUSTERED (isbn)
      WITH FILLFACTOR=70,
    CONSTRAINT unq_book UNIQUE CLUSTERED
      (book_name,pub_date))
GO
```

Две эти команды дают похожий результат, только ограничение *UNIQUE* на уровне таблицы содержит два столбца, а на уровне столбца – один.

Следующий оператор добавляет в таблицу ограничение *CHECK*, но не проверяет существующие строки таблицы на соответствие этому ограничению:

```
ALTER TABLE favorite_book WITH NOCHECK
ADD CONSTRAINT extra_check CHECK (ISBN > 1)
GO
```

Теперь мы добавляем в таблицу столбец со значением по умолчанию, которое помещается в каждую существующую строку таблицы:

```
ALTER TABLE favorite_book ADD reprint_nbr INT NULL
CONSTRAINT add_reprint_nbr DEFAULT 1 WITH VALUES
GO
-- теперь отключим ограничение
ALTER TABLE favorite_book NOCHECK CONSTRAINT add_reprint_nbr
GO
```

См. также

CREATE SCHEMA

DROP

CREATE/ALTER TRIGGER

Триггер – это хранимая процедура специального типа, которая выполняется автоматически при выполнении определенных манипуляций с таблицей. Триггер явно связан с таблицей и является зависимым объектом. Например, вы могли бы захотеть, чтобы столбец **part_numbers** таблицы **sales** обновлялся автоматически при изменении столбца **part_number** таблицы **products**. Это можно сделать с помощью триггера.

Оператор *ALTER TRIGGER* не является частью стандарта ANSI.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
CREATE TRIGGER имя_триггера
{BEFORE | AFTER} {DELETE | INSERT | UPDATE [OF столбец[, ...]]}
ON имя_таблицы
[REFERENCING {OLD {[ROW] | TABLE} [AS] старое_имя | NEW
{ROW | TABLE} [AS] новое_имя}] [FOR EACH { ROW | STATEMENT }]
[WHEN (условия)]
[BEGIN ATOMIC]блок_кода
[END]
```

Ключевые слова

CREATE TRIGGER имя_триггера

Создает триггер с заданным именем и связывает его с определенной таблицей.

BEFORE | AFTER

Указывает, что действия триггера выполняются либо до (*BEFORE*), либо после (*AFTER*) DML-операции, вызывающей выполнение триггера. Триггер типа *BEFORE* выполняется до операторов *INSERT*, *UPDATE* и *DELETE* и позволяет выполнить любые действия, вплоть до полной подмены DML-операции. Триггер типа *AFTER* выполняется после завершения вызывающей его операции и позволяет постфактум выполнить полезные действия, например пересчет нарастающего итога.

DELETE | INSERT | UPDATE [OF столбец[, ...]]

Определяет тип операции, вызывающей выполнение триггера: операторы *DELETE*, операторы *INSERT* или операторы *UPDATE*. Также вы можете указать список столбцов для оператора *UPDATE*, обновление значений которых вызывает триггер. Если обновляется столбец не из указанного списка, то триггер не выполняется.

ON имя_таблицы

Указывает таблицу, для которой создается триггер.

REFERENCING {OLD{[ROW] | TABLE} [AS] старое_имя | NEW {ROW | TABLE} [AS] новое_имя}

Используется для создания псевдонимов для старых (*OLD*) и новых (*NEW*) версий строки (*ROW*) или таблицы (*TABLE*). Хотя в синтаксисе триггера эти опции показаны как взаимоисключающие, на самом деле вы можете указать до четырех псевдонимов: для старой строки, новой строки, старой таблицы и новой таблицы. Псевдонимы *OLD* относятся к данным, которые содержались в строке или таблице до DML-операции, вызвавшей триггер, а *NEW* относится к данным, которые будут в строке или таблице после выполнения DML-операции. Обратите внимание, что слово *ROW* опционально, а *TABLE* – обязательно. (То есть *OLD ROW AS* – это тоже самое, что *OLD AS*, а для *TABLE* единственным допустимым вариантом является *OLD TABLE AS*.) Триггеры *INSERT* не имеют контекста *OLD*, а триггеры *DELETE* не имеют контекста *NEW*. Ключевое слово *AS* необязательно. Если во фразе *REFERENCING* используется *OLD ROW* или *NEW ROW*, то обязательно должна использоваться фраза *FOR EACH ROW*.

FOR EACH {ROW | STATEMENT}

Определяет способ вызова триггера: либо один раз для каждой измененной записи (*ROW*), либо один раз для каждого оператора. Представьте оператор *UPDATE*, который изменяет зарплаты 100 сотрудников. Триггер *FOR EACH ROW* в таком случае будет выполнен 100 раз, а триггер *FOR EACH STATEMENT* будет выполнен один раз.

WHEN (условия)

Позволяет указать дополнительное условие на запуск триггера. Предположим, у вас есть триггер *DELETE employee*, который вызывается для каждого удаления из таблицы сотрудников. Если при вызове триггера дополнительное условие при вычислении возвращает *TRUE*, то выполнение триггера продолжается. В противном случае триггер не запускается.

BEGIN ATOMIC | *блок_кода* | *END*

Стандарт ANSI требует, чтобы тело триггера состояло либо из одного SQL-оператора, либо было заключено в *BEGIN*...*END*.

Общие правила

Триггеры по умолчанию запускаются один раз для одного оператора. То есть оператор *INSERT* может вставить 500 записей, но *INSERT*-триггер этой таблицы выполнится один раз. Однако в некоторых платформах поддерживаются триггеры, запускающиеся один раз для каждой строки, затрагиваемой DML-операцией. Оператор, вставляющий 500 строк, вызовет такой триггер 500 раз, по разу для каждой строки.

Кроме типа оператора, вызывающего срабатывание триггера, нужно также указывать момент срабатывания. Триггер может срабатывать до (*BEFORE*) выполнения оператора, после (*AFTER*) выполнения оператора, а иногда и вместо (*INSTEAD OF*), если это поддерживается платформой. Триггеры, выполняющиеся до или вместо DML-операции, не видят изменений, выполняемых оператором, а триггеры *AFTER* работают уже с измененными данными.

Триггеры могут использовать две псевдотаблицы. (Псевдотаблицами они называются, так как не создаются явно с помощью оператора *CREATE TABLE*, но тем не менее, логически существуют в базе данных.) Эти псевдотаблицы могут называть в разных платформах по-разному, здесь мы их будем именовать **Before** и **After**. Они имеют в точности такую же структуру, что и таблица, с которой связан триггер, но содержат моментальные копии содержимого таблицы: **Before** содержит копию всех строк таблицы, какими они были до выполнения оператора, вызвавшего триггер, **After** содержит копию всех строк таблицы после выполнения оператора. Вы можете использовать данные этих псевдотаблиц в триггере для определения того, что вы хотите сделать.

Вот пример *BEFORE*-триггера для Oracle, использующего таблицы *OLD* и *NEW*. (В SQL Server таким же образом используются таблицы *INSERTED* и *DELETED*.) Триггер создает запись в таблице аудита перед изменением зарплаты сотрудника:

```
CREATE TRIGGER if_emp_changes
BEFORE DELETE OR UPDATE ON employee
FOR EACH ROW
WHEN (new.emp_salary <> old.emp_salary)
BEGIN
  INSERT INTO employee_audit
  VALUES ('old',
          :old.emp_id,
          :old.emp_salary,
          :old.emp_ssn);
END;
```

Для того чтобы писать более эффективные и понятные триггеры, вы можете использовать дополнительные возможности, предоставляемые некоторыми платформами. Например, в Oracle есть специальная фраза *IF*...*THEN*, используемая только в триггерах. Фраза имеет вид *IF {DELETING* | *INSERTING* | *UPDATING}*

THEN. В следующем примере создается триггер на *DELETE* и *UPDATE*, и в триггере используется фраза *IF DELETING THEN*:

```
CREATE TRIGGER if_emp_changes
BEFORE DELETE OR UPDATE ON employee
FOR EACH ROW
BEGIN
  IF DELETING THEN
    INSERT INTO employee_audit
    VALUES ('DELETED', :old.emp_id,
            :old.emp_salary, :old.emp_ssn);
  ELSE
    INSERT INTO employee_audit
    VALUES ('UPDATED', :old.emp_id,
            :new.emp_salary, :old.emp_ssn);
  END IF;
END;
```

В следующем примере для SQL Server мы создаем триггер *INSTEAD OF INSERT* для подмены операции вставки в таблицу **employee**. В триггере в таблицу **employee** вставляются только сотрудники, находящиеся в штате, а сотрудники, работающие по контракту, вставляются в отдельную таблицу **contractors**:

```
CREATE TRIGGER if_emp_is_contractor
INSTEAD OF INSERT ON employee
BEGIN
  INSERT INTO contractor
  SELECT * FROM inserted WHERE status = 'CON'
  INSERT INTO employee
  SELECT * FROM inserted WHERE status = 'FTE'
END
GO
```

Создание триггера для таблицы, в которой уже есть данные, не вызывает срабатывания триггера. Триггер будет срабатывать только для операторов, выполненных после создания триггера.

Изменение структуры таблицы (например, добавление столбца или изменение типа данных существующего столбца) в целом не влияет на триггеры, связанные с этой таблицей, если только изменение не делает триггер неработоспособным. Например, если вы добавляете столбец в таблицу, то он будет просто игнорироваться триггером (за исключением триггеров *UPDATE*). Однако если вы удалите столбец, используемый триггером, то каждое выполнение триггера будет завершаться ошибкой.

Советы и хитрости

Одной из основных проблем при использовании триггеров является неправильное и неконтролируемое применение вложенных и рекурсивных триггеров. *Вложенный триггер* – это триггер, выполняющий DML-операцию, вызывающую другой триггер. Предположим, например, что у нас есть таблицы **T1**, **T2** и **T3**. Для таблицы **T1** создан триггер *BEFORE INSERT*, вставляющий строки в таблицу **T2**. Для таблицы **T2** также создан триггер *BEFORE INSERT*, вставляющий строки в таблицу **T3**. Само по себе это не всегда плохо, если логика операций хо-

рошо продумана, но порождает две проблемы. Во-первых, вставка записи в **T1** потребует больше транзакций и операций ввода-вывода, чем обычный оператор *INSERT* для таблицы **T1**. Во-вторых, будут большие неприятности, если триггер таблицы **T3** вставляет данные в **T1**. В такой ситуации процесс вызова триггеров может зациклиться, захватить все свободные ресурсы и даже привести к остановке сервера.

Рекурсивный триггер – это триггер, который может запустить срабатывание самого себя (например, триггер *INSERT*, вставляющий записи в свою базовую таблицу). Если триггер запрограммирован неверно, то это также может привести к бесконечному циклу вызовов триггера. В некоторых платформах, в подтверждение опасности такой ситуации, для использования рекурсивных триггеров необходимо устанавливать специальные параметры.

MySQL

Реализация оператора *CREATE TRIGGER* в MySQL выглядит следующим образом:

```
CREATE [DEFINER = {имя_пользователя | CURRENT_USER}]
  TRIGGER имя_триггера {BEFORE | AFTER}
     {INSERT | UPDATE | DELETE}
  ON имя_таблицы FOR EACH ROW блок_кода
```

где:

DEFINER = {имя_пользователя | CURRENT_USER}

Указывает учетную запись, которая используется при проверке привилегий. Вы можете либо указать какого-либо существующего пользователя, либо указать текущего пользователя (*CURRENT_USER*). *CURRENT_USER* используется по умолчанию.

В MySQL нельзя создавать триггеры для временных таблиц. Триггеры на вставку будут срабатывать при вставке строк в таблицу любым способом, а не только оператором *INSERT*. То есть триггеры на вставку будут также срабатывать при выполнении операторов *LOAD DATA* и *REPLACE*. Аналогично, триггер на удаление также будет срабатывать при операторе *REPLACE*. Однако триггеры не запускаются при каскадных изменениях по внешним ключам.

MySQL допускает не более одного триггера каждого типа на таблицу. То есть таблица может иметь триггер *BEFORE INSERT* и триггер *AFTER INSERT*, но не два триггера *AFTER INSERT* одновременно. Для выполнения в триггере нескольких операторов заключите их в *BEGIN...END*.

MySQL не поддерживает оператор *ALTER TRIGGER*.

Oracle

Oracle поддерживает определенный стандартом ANSI вариант оператора *CREATE TRIGGER* с небольшими дополнениями и изменениями:

```
CREATE [OR REPLACE] TRIGGER имя_триггера
{BEFORE | AFTER | INSTEAD OF}
{ {[обьектное_событие] [событие_бд] [...]
ON {DATABASE |схема.SCHEMA}} |
```

```
    {[DELETE] [OR] [INSERT] [OR] [UPDATE [OF столбец[, ...]]]
       [...]} ON {имя_таблицы | [NESTED TABLE имя_столбца OF]
       имя_представления}
     [REFERENCING {[OLD [AS] старое_имя] [NEW [AS] новое_имя]
       [PARENT [AS] имя_родителя]}]
    [FOR EACH ROW] }
[FOLLOWS имя_триггера]
[{ENABLE | DISABLE}]
[WHEN (условия)] блок_кода
```

ALTER TRIGGER позволяет переименовывать, включать и выключать триггер без его удаления и пересоздания. Оператор *ALTER TRIGGER* выглядит следующим образом:

```
ALTER TRIGGER имя_триггера
{ {ENABLE | DISABLE} | RENAME TO новое_имя_триггера |
  COMPILE [директивы_компилятора] [DEBUG] [REUSE SETTINGS] }
```

Параметры имеют следующие значения:

OR REPLACE

Пересоздает существующий триггер с новым определением.

объектное_событие

Помимо обычных DML-операторов триггеры в Oracle могут вызываться и другими событиями, связанными с различными объектами. Фраза *объектное_событие* также сопровождается ключевыми словами *BEFORE* или *AFTER*. Список объектных событий следующий:

ALTER

Триггер срабатывает при выполнении любого оператора *ALTER*, кроме *ALTER DATABASE*.

ANALYZE

Триггер срабатывает при валидации структуры объекта базы данных либо при сборе или удалении статистики индекса.

ASSOCIATE STATISTICS

Триггер срабатывает при связывании статистики с объектом базы данных.

AUDIT

Триггер срабатывает при каждом включении отслеживания базой данных SQL-оператора или действий с объектом базы данных.

COMMENT

Триггер срабатывает при каждом добавлении комментария к объекту.

DDL

Триггер срабатывает при возникновении *любого* события из этого списка.

DISASSOCIATE STATISTICS

Триггер срабатывает при удалении статистики по объекту.

DROP

Триггер срабатывает каждый раз, когда оператор DROP удаляет информацию об объекте из словаря базы данных.

GRANT

Триггер срабатывает при выдаче привилегий или ролей другим пользователям или ролям.

NOAUDIT

Триггер срабатывает, когда оператором *NOAUDIT* отключается отслеживание SQL-операторов и действий с объектами базы данных.

RENAME

Триггер срабатывает при переименовании объекта базы данных.

REVOKE

Триггер срабатывает, когда пользователь отзывает привилегию или роль у другого пользователя или роли.

TRUNCATE

Триггер срабатывает, когда таблица или кластер очищается с помощью оператора *TRUNCATE*.

событие_бд

Помимо обычных DML-операторов триггеры в Oracle могут вызываться и определенными событиями уровня базы данных. Фраза *событие_бд* также сопровождается ключевыми словами *BEFORE* или *AFTER*. Список событий базы данных следующий:

LOGON

Триггер срабатывает при подключении пользователя к базе данных. Допустимы только триггеры типа *AFTER*.

LOGOFF

Триггер срабатывает при отключении пользователя от базы данных. Допустимы только триггеры типа *BEFORE*.

SERVERERROR

Триггер срабатывает при возникновении серверной ошибки. Допустимы только триггеры типа *AFTER*.

SHUTDOWN

Триггер срабатывает при остановке экземпляра СУБД. Допустимы только триггеры *BEFORE* с фразой *ON DATABASE*.

STARTUP

Триггер срабатывает при открытии экземпляра СУБД. Допустимы только триггеры *AFTER* с фразой *ON DATABASE*.

SUSPEND

Триггер срабатывает, когда из-за ошибки базы данных приостанавливается транзакция. Допустимы только триггеры типа *AFTER*.

ON {DATABASE |схема.SCHEMA}

Если используется *ON DATABASE*, то триггер срабатывает для событий, возникающих с объектами в любых схемах базы данных. А *ON схема.SCHEMA* позволяет указать конкретную схему, события с объектами которой будут вызывать запуск триггера.

ON [NESTED TABLE имя_столбца OF] имя_представления

Указывает, что триггер должен срабатывать только при DML-операциях, затрагивающих столбец (столбцы) указанного представления. Фраза *ON NESTED TABLE* совместима только с триггерами типа *INSTEAD OF*.

REFERENCING PARENT [AS] имя_родителя

Задает псевдоним для текущей строки родительской таблицы (т. е. супертаблицы). В остальном идентично стандарту ANSI.

FOLLOWS имя_триггера

Указывает, что новый триггер должен запускаться только после завершения другого существующего триггера того же типа. Однако рекомендуется вместо набора триггеров одинакового типа, которые должны запускаться в определенном порядке, создавать один триггер, обрабатывающий все ситуации, которые обрабатывались бы несколькими триггерами.

ENABLE

Если используется в операторе *ALTER TRIGGER*, то активирует отключенный ранее триггер. При создании триггера создает его включенным (триггер включен по умолчанию). Включить все триггеры можно оператором *ALTER TABLE ... ENABLE ALL TRIGGERS*.

DISABLE

Если используется в операторе *ALTER TRIGGER*, то отключает активный триггер. При создании триггера создает его отключенным. Отключить все триггеры можно оператором *ALTER TABLE ... DISABLE ALL TRIGGERS*.

RENAME TO новое_имя

Используется в операторе *ALTER TRIGGER* для переименования триггера. Состояние триггера остается без изменений.

COMPILE [DEBUG] [REUSE SETTINGS]

Компилирует триггер и все объекты, от которых он зависит. Если хотя бы один из объектов содержит ошибки, то триггер становится невалидным. Если все объекты и тело триггера не содержат ошибок, то триггер компилируется и становится валидным.

DEBUG

Указывает, что при компиляции нужно создавать и сохранять отладочную информацию.

REUSE SETTINGS

Указывает, что Oracle должен сохранить параметры компилятора, что может значительно экономить время при компиляции.

директивы_компилятора

Устанавливает значения параметров компилятора PL/SQL в формате *директива='значение'*. Используются следующие директивы: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS* и *NLS_LENGTH_SEMANTICS*. Установленные значения действуют только для компилируемого объекта.

При использовании псевдотаблиц *OLD* и *NEW* их нужно использовать с префиксом в виде двоеточия, кроме использования во фразе *WHEN*. В следующем примере в триггере вызывается хранимая процедура и значения :*OLD* и :*NEW* используются в качестве аргументов:

```
CREATE TRIGGER scott.sales_check
BEFORE INSERT OR UPDATE OF ord_id, qty ON scott.sales
  FOR EACH ROW
  WHEN (new.qty > 10)
  CALL check_inventory(:new.ord_id, :new.qty, :old.qty);
```

Несколько триггеров одинакового уровня (оператора или записи) для одной таблицы можно комбинировать в один триггер. При этом тело триггера согласно логике обработки можно разбить на отдельные части, используя фразу *IF INSERTING/UPDATING/DELETING THEN*. Также в этой структуре можно использовать *ELSE*.

Вот пример триггера на событие базы данных:

```
CREATE TRIGGER track_errors
AFTER SERVERERROR ON DATABASE
BEGIN
  IF (IS_SERVERERROR (04030))
    THEN INSERT INTO errors ('Memory error');
  ELSIF (IS_SERVERERROR (01403))
    THEN INSERT INTO errors ('Data not found');
  END IF;
END;
```

В следующем примере создается триггер на события в отдельной схеме:

```
CREATE OR REPLACE TRIGGER create_trigger
AFTER CREATE ON scott.SCHEMA
BEGIN
  RAISE_APPLICATION_ERROR (num => -20000, msg =>
  'Scott created an object');
END;
```

DDL-триггеры

Oracle поддерживает триггеры на DDL-операции, то есть вы, например, можете создать триггер, который будет срабатывать при создании таблицы или удалении представления. DDL-триггеры могут быть созданы либо для всей базы данных либо для отдельной схемы для событий типа *CREATE*, *ALTER* или *DROP*. Например:

```
CREATE TRIGGER audit_object_changes AFTER CREATE ON SCHEMA
code_body;
```

Полный список событий, вызывающих срабатывание DDL-триггера, следующий: *ALTER*, *ANALYZE*, *ASSOCIATE STATISTICS*, *AUDIT*, *COMMENT*, *CREATE*, *DISASSOCIATE STATISTICS*, *DROP*, *GRANT*, *NOAUDIT*, *RENAME*, *REVOKE*, *TRUNCATE* или *DDL* (запускает триггер при выполнении любого оператора из списка). Также вы можете создать триггер, который срабатывает при определенном состоянии базы данных, а не при выполнении оператора. Вы можете использовать следующие состояния: *AFTER STARTUP*, *BEFORE SHUTDOWN*, *AFTER*

DB_ROLE_CHANGE, *AFTER LOGON*, *BEFORE LOGOFF*, *AFTER SERVERERROR* и *AFTER SUSPEND*.

PostgreSQL

Реализация оператора *CREATE TRIGGER* в PostgreSQL поддерживает подмножество возможностей, описанных в стандарте ANSI. В PostgreSQL триггер выполняется либо до выполнения DML-операции и проверки ограничений, либо после выполнения операции и проверки всех ограничений, когда сделанные изменения уже видны триггеру. Также можно создавать *INSTEAD OF* триггеры для подмены операторов вставки, обновления и удаления записей. Синтаксис оператора *CREATE TRIGGER* следующий:

```
CREATE TRIGGER имя_триггера
{ BEFORE | AFTER }
{ {[DELETE] [OR | ,] [INSERT] [OR | ,] [UPDATE]} [OR ...] }
ON имя_таблицы
FOR EACH { ROW | STATEMENT }
EXECUTE PROCEDURE имя_функции (параметры)
```

Оператора *ALTER TRIGGER* в PostgreSQL позволяет лишь переименовать триггер:

```
ALTER TRIGGER имя_триггера ON имя_таблицы
RENAME TO новое_имя_триггера
```

Параметры оператора *CREATE TRIGGER* следующие:

OR

Используется для указания дополнительных действий, вызывающих срабатывание триггера. Вместо *OR* можно использовать запятую.

FOR EACH ROW

Явно указывает, что триггер срабатывает для каждой записи. Это является поведением по умолчанию. Слово *STATEMENT* просто поддерживается на уровне синтаксиса, но триггер при этом не становится триггером уровня оператора.

EXECUTE PROCEDURE имя_функции(параметры)

Вызывает функцию, предварительно созданную оператором *CREATE FUNCTION*. В PostgreSQL нет своего встроенного процедурного языка.

В следующем примере создается *BEFORE*-триггер, срабатывающий для каждой вставленной или обновленной записи таблицы **sales** и проверяющий, что код дистрибьютора присутствует в таблице **distributors**:

```
CREATE TRIGGER if_dist_exists
BEFORE INSERT OR UPDATE ON sales
FOR EACH ROW
EXECUTE PROCEDURE check_primary_key
('dist_id', 'distributors', 'dist_id');
```

Триггеры *BEFORE* и *AFTER* поддерживаются стандартом ANSI. Триггеры *INSTEAD OF* в PostgreSQL позволяют подменить операцию, вызвавшую срабатывание триггера, своей функцией.

SQL Server

Триггеры в SQL Server поддерживают базовую функциональность, описанную в ANSI, а также расширения в виде триггеров *INSTEAD OF* и отслеживания изменений столбцов. Фразы *REFERENCING* и *WHEN* не поддерживаются. Синтаксис следующий:

```
{CREATE | ALTER} TRIGGER имя_триггера ON имя_таблицы
[WITH [ENCRYPTION] [EXEC[UTE] AS {CALLER | SELF |
    'имя_пользователя'}]]
{FOR | AFTER | INSTEAD OF}
{ dml_события | ddl_события }
[WITH APPEND]
[NOT FOR REPLICATION]
AS
  [IF UPDATE(столбец) [{AND | OR} UPDATE(столбец)][...]]
  блок_кода
```

где:

{CREATE|ALTER} TRIGGER *имя_триггера*

Создает новый триггер с указанным именем или изменяет существующий триггер, добавляя или модифицируя параметры и тело триггера. При изменении триггера сохраняются все его разрешения и зависимости.

ON *имя_таблицы*

Указывает имя таблицы или представления, для которого создается триггер. Для представлений можно создавать триггер *INSTEAD OF*, в том случае, если представление обновляемое и не использует фразу *WITH CHECK OPTION*.

WITH ENCRYPTION

Шифрует текст оператора *CREATE TRIGGER*, что отражается в таблице **syscomments**. Шифрование полезно для защиты интеллектуальной собственности. Зашифрованный триггер нельзя использовать при репликации.

EXEC[UTE] AS {CALLER|SELF|OWNER|'имя_пользователя'}

Определяет привилегии, с которыми выполняется триггер. *CALLER* (значение по умолчанию) указывает, что необходимо использовать привилегии пользователя, чьи действия вызвали срабатывание триггера. *SELF* указывает, что триггер выполняется с привилегиями создателя. *OWNER* указывает, что триггер выполняется с привилегиями текущего владельца. И, наконец, можно явно указать имя пользователя, привилегии которого будут использоваться при выполнении триггера.

FOR|AFTER|INSTEAD OF

Определяет момент выполнения триггера. *FOR* и *AFTER* являются синонимами и указывают, что триггер запускается только после успешного завершения DML-оператора, всех каскадных изменений и проверки ограничений. Триггер *INSTEAD OF* похож на триггер *BEFORE*, описанный в ANSI, но при этом может полностью подменить DML-операцию. Триггеры *INSTEAD OF DELETE* не могут быть использованы в ситуациях, когда удаление вызывает каскадные изменения. Только триггерам *INSTEAD OF* доступны столбцы типов *TEXT*, *NTEXT* и *IMAGE*.

dml_события

Указывает стандартные операторы DML, вызывающие срабатывание триггера: *INSERT*, *DELETE* и/или *UPDATE*. В одном операторе можно использовать одно или несколько событий.

ddl_события

Указывает операторы DDL, вызывающие срабатывание триггера: *CREATE*, *ALTER*, *DROP*, *GRANT*, *DENY*, *REVOKE*, *BIND*, *UNBIND*, *RENAME* или *UPDATE STATISTICS*. DDL-триггер может срабатывать до или после события. Также вы можете указать триггер, срабатывающий при подключении пользователя (*FOR* или *AFTER LOGON*). В SQL Server также есть предопределенные наборы групп, в которых собраны DDL-события для объектов одного типа: например, группа *DDL_TABLE_EVENTS* включает события *CREATE*, *ALTER* и *DROP TABLE*. Полный список таких групп приводится в документации по SQL Server.

WITH APPEND

Добавляет еще один триггер уже существующего типа. Это фраза поддерживается для обратной совместимости с более ранними версиями и используется только для триггеров типа *FOR*. Эту фразу нельзя использовать для триггеров *INSTEAD OF* или *AFTER*, а также с триггерами, написанными на CLR.

NOT FOR REPLICATION

Указывает, что триггер не вызывается операциями, выполненными при репликации.

IF UPDATE(столбец) [{AND|OR} UPDATE(столбец)][...]

Позволяет выбрать определенные столбцы, изменение которых запускает триггер. Триггеры по набору столбцов создаются только для операции *INSERT* и *UPDATE*, но не для *DELETE*. Если вставляются или обновляются значения столбца, не входящего в указанный список, то триггер не запускается.

SQL Server позволяет создавать несколько триггеров одного типа для таблицы или представления. Поэтому одна таблица может иметь три триггера *UPDATE* и несколько триггеров *AFTER*. Порядок их выполнения недетерминирован, хотя первый и последний триггеры можно выбрать при помощи системной хранимой процедуры **sp_settriggerorder**. На каждую из операций *INSERT*, *UPDATE* и *DELETE* может быть создано по одному триггеру *INSTEAD OF*.

В SQL Server допустимы любые комбинации событий, вызывающих триггер, достаточно перечислить их через запятую при создании триггера. При этом для каждого оператора будет выполняться одинаковый код.

SQL Server выполняет триггеры в режиме *FOR EACH STATEMENT* – один раз для одного оператора, как это описано в стандарте ANSI.

Внутри триггера в SQL Server доступны псевдотаблицы **deleted** и **inserted**. Они аналогичны описанным ранее в разделе SQL2003 псевдотаблицам **before** и **after**. Эти псевдотаблицы идентичны по структуре основной таблице, но содержат копии данных до выполнения DML-операции **(deleted)** и после выполнения DML-операции **(inserted)**.

Фраза *AS IF UPDATE(столбец)* проверяет влияние операторов *UPDATE* и *INSERT* на определенный столбец, как это описано в стандарте ANSI для фразы *UPDATE(столбец)*. Несколько столбцов можно перечислить, указав дополнительные фразы *UPDATE(столбец)*. После *AS IF UPDATE* следует блок кода, и если он состоит из нескольких операторов, то необходимо использовать *BEGIN*... *END*. Эта фраза функционально эквивалентна конструкции *IF*...*THEN*...*ELSE*.

Помимо перехвата триггером DML-операторов, как это показано в примере на ANSI SQL, SQL Server позволяет выполнять и другие действия. В следующем примере мы запрещаем изменение данных таблицы **sales_archive_2002**, о чем уведомляем пользователей:

```
CREATE TRIGGER archive_trigger
ON sales_archive_2002
FOR INSERT, UPDATE
AS RAISERROR
  (50009, 16, 10, 'No changes allowed to this table')
GO
```

Запрещено использовать в теле триггера следующие операторы: *ALTER*, *CREATE*, *DROP*, *DENY*, *GRANT*, *REVOKE*, *LOAD*, *RESTORE*, *RECONFIGURE*, *TRUNCATE*, любые операторы *DISK* и команду *UPDATE STATISTICS*.

SQL Server допускает рекурсивный вызов триггеров, при условии установки соответствующего параметра системной процедуры **sp_dboption**. Рекурсивные триггеры своими действиями опять вызывают себя же. Например, триггер *FOR INSERT* для таблицы **T1** может вставлять записи в эту же таблицу. Так как рекурсивные триггеры достаточно опасны, то по умолчанию такая возможность отключена.

SQL Server допускает также вложенные триггеры, глубина вложенности может достигать 32. Если один из триггеров откатывает транзакцию, то дальнейшие триггеры не вызываются. Примером вложенного триггера может быть триггер на таблице **T1**, который выполняет модификацию таблицы **T2**, что вызывает выполнение триггера, выполняющего модификацию таблицы **T3** и т. д. Триггеры прерываются, если обнаруживается бесконечный цикл. Возможность использования вложенных триггеров включается системной процедурой **sp_configure**. Если использование вложенных триггеров запрещено, то запрещены и рекурсивные триггеры, вне зависимости от настройки **sp_dboption**.

В следующем примере мы создадим триггер, который отслеживает изменения таблицы **people** в столбцах 2, 3, 4 и записывает эти изменения в таблицу **people_reroute**. Триггер также будет сохранять имя пользователя, выполнившего обновление, и время транзакции:

```
CREATE TABLE people
 (people_id   CHAR(4),
  people_name VARCHAR(40),
  people_addr VARCHAR(40),
  city        VARCHAR(20),
  state       CHAR(2),
  zip         CHAR(5),
  phone       CHAR(12),
  sales_rep   empid NOT NULL)
```

```
GO
CREATE TABLE people_reroute
 (reroute_log_id      UNIQUEIDENTIFIER DEFAULT NEWID( ),
  reroute_log_type    CHAR (3) NOT NULL,
  reroute_people_id   CHAR(4),
  reroute_people_name VARCHAR(40),
  reroute_people_addr VARCHAR(40),
  reroute_city        VARCHAR(20),
  reroute_state       CHAR(2),
  reroute_zip         CHAR(5),
  reroute_phone       CHAR(12),
  reroute_sales_rep   empid NOT NULL,
  reroute_user        sysname DEFAULT SUSER_SNAME( ),
  reroute_changed     datetime DEFAULT GETDATE( ) )
GO
CREATE TRIGGER update_person_data
ON people
FOR update AS
IF (COLUMNS_UPDATED(people_name)
   OR COLUMNS_UPDATEE(people_addr)
   OR COLUMNS_UPDATED(city) )
BEGIN
-- Audit OLD record
  INSERT INTO people_reroute (reroute_log_type,
     reroute_people_id, reroute_people_name,
     reroute_people_addr, reroute_city)
  SELECT 'old', d.people_id, d.people_name,
     d.people_addr, d.city
  FROM deleted AS d
-- Audit NEW record
  INSERT INTO people_reroute (reroute_log_type,
     reroute_people_id, reroute_people_name,
     reroute_people_addr, reroute_city)
  SELECT 'new', n.people_id, n.people_name,
     n.people_addr, n.city
  FROM inserted AS n
END
GO
```

Помните, что в SQL Server используется отложенное связывание имен, то есть операторы CREATE могут ссылаться на объекты, которые еще не созданы.

В SQL Server поддерживаются триггеры, написанные на языках, исполняемых в CLR (Common Language Runtime) Microsoft .NET Framework. Синтаксис объявления таких триггеров похож на синтаксис, используемый для обычных триггеров, но вместо кода используются внешние сборки. Если вы хотите узнать о программировании для SQL Server с использованием CLR, то обратитесь к документации.

См. также

CREATE/ALTER FUNCTION/PROCEDURE

DELETE

DROP

INSERT
UPDATE

CREATE/ALTER TYPE

Оператор *CREATE TYPE* позволяет создавать *пользовательские типы данных* (user-defined types, UDT), что эквивалентно понятию классов из объектно-ориентированного программирования. Пользовательские типы данных расширяют возможности SQL в области объектно-ориентированного программирования, поддерживают наследование типов и другие объектно-ориентированные возможности. Также на базе пользовательских типов данных вы можете создавать так называемые *объектные таблицы с помощью фразы CREATE TABLE*. Понятие объектной таблицы эквивалентно понятию «экземпляров классов» из объектно-ориентированного программирования.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с ограничениями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
CREATE TYPE имя_типа
[UNDER имя_супертипа]
[AS [новое_имя_типа] тип_данных [атрибут][, ...]
  {[REFERENCES ARE [NOT] CHECKED
  [ON DELETE {NO ACTION | CASCADE | RESTRICT
            | SET NULL | SET DEFAULT}]] |
  [DEFAULT значение] |
  [COLLATE схема_упорядочения]}]
  [[NOT] INSTANTIABLE]
  [[NOT] FINAL]
[REF IS SYTEM GENERATED |
  REF USING тип_данных
    [CAST {(SOURCE AS REF) | (REF AS SOURCE)}
WITH идентификатор] |
REF новое_имя_типа[, ...] ]
[CAST {(SOURCE AS DISTINCT) | (DISTINCT AS SOURCE)}
  WITH идентификатор]
[определение_метода [, ...]]
```

Для изменения типа данных используется следующий синтаксис:

```
ALTER TYPE имя_типа {ADD ATTRIBUTE опеределение_типа|
    DROP ATTRIBUTE имя_типа}
```

Ключевые слова

{CREATE|ALTER} TYPE *имя_типа*

Создает новый или меняет существующий тип данных.

UNDER супертип

Указывает существующий пользовательский тип данных, подтипом которого является создаваемый тип. (Пользовательский тип данных может выступать супертипом, если он объявлен с параметром *NOT FINAL*.)

AS [новое_имя_типа] тип_данных [атрибут][, ...]

Описывает атрибуты типа в том же виде, как описываются столбцы таблицы в операторе *CREATE TABLE*, но без ограничений целостности. Атрибут создается либо на основании обычного скалярного типа данных, как *VARCHAR(10)*, либо на основании другого, ранее созданного пользовательского типа данных, либо на основании определенного пользователем домена. Объявление пользовательского типа на базе стандартного скалярного типа данных (например, *CREATE TYPE my_type AS INT*) создает *отдельный* (*distinct*) тип, а объявление пользовательского типа, содержащего набор атрибутов, создает *структурный* (*structured*) тип. Допустимы следующие атрибуты структурного типа:

ON DELETE NO ACTION

Вызывает ошибку при нарушении внешнего ключа (поведение по умолчанию).

ON DELETE RESTRICT

Синоним для *NO ACTION*.

ON DELETE CASCADE

В ссылающемся столбце проставляется значение столбца, на который ссылаются.

ON DELETE SET NULL

Ссылающемуся столбцу присваивается значение *NULL*.

ON DELETE SET DEFAULT значение

Определяет для пользовательского типа значение по умолчанию.

COLLATE схема_упорядочения

Определяет схему упорядочения (то есть порядок сортировки) для пользовательского типа данных. По умолчанию используется схема упорядочения базы данных.

[NOT] INSTANTIABLE

Определяет, можно (*INSTANTIABLE*) или нет (*NOT INSTANTIABLE*) создавать экземпляры данного типа. Для создания объектных таблиц тип должен иметь параметр *INSTANTIABLE*.

[NOT] FINAL

FINAL означает, что тип не может иметь подтипов. *NOT FINAL* означает, что тип может иметь подтипы.

REF

Определяет генерируемую системой или пользователем спецификацию ссылки, т. е. некоего уникального идентификатора, являющегося указателем, на который может ссылаться другой тип. Ссылаясь на существующий тип с использованием его ссылочной спецификации, вы можете наследовать в созда-

ваемом типе все атрибуты существующего типа. Существует три способа генерации значений ссылочного столбца объектной таблицы:

новое_имя_udt

Объявляет, что ссылочная спецификация обеспечивается другим указанным пользовательским типом данных.

IS SYSTEM GENERATED

Указывает, что ссылочная спецификация генерируется автоматически (как при автоматическом инкременте столбца). Этот вариант используется по умолчанию.

USING тип_данных [CAST {(SOURCE AS REF) | (REF AS SOURCE)} WITH идентификатор]

Указывает, что пользователь определяет ссылочную спецификацию. Это делается с использованием существующего типа данных и, при необходимости, с приведением типа данных к новому типу. Вы можете использовать *CAST (SOURCE AS REF) WITH* для преобразования значения стандартного типа в ссылочный тип структурного типа, либо использовать *CAST (REF AS SOURCE) WITH* для преобразования значения структурного типа к *типу_данных*. Фраза *WITH* позволяет объявить дополнительный идентификатор для преобразуемого типа.

CAST {(SOURCE AS DISTINCT) | (DISTINCT AS SOURCE)} WITH идентификатор определения_метода[, ...]

Объявляет один или несколько методов пользовательского типа данных. *Метод* – это специальная пользовательская функция, создаваемая оператором *CREATE METHOD* (смотрите описание *CREATE FUNCTION*). Фраза *определение_метода* не нужна для структурных типов, так как их методы создаются неявно. По умолчанию для методов используются настройки *LANGUAGE SQL*, *PARAMETER STYLE SQL*, *NOT DETERMINISTIC*, *CONTAINS SQL* и *RETURN NULL ON NULL INPUT*.

ADD ATTRIBUTE определение_типа

Добавляет атрибут в существующий тип данных, используется такой же синтаксис, как в описанной выше фразе *AS*. Используется в операторе *ALTER TYPE*.

DROP ATTRIBUTE имя_типа

Удаляет атрибут существующего пользовательского типа данных. Используется в операторе *ALTER TYPE*.

Общие правила

Вы можете применять пользовательские типы данных для улучшенного контроля целостности данных и для упрощения этого контроля. Важной концепцией пользовательских типов данных является механизм создания *подтипов*, то есть типов, созданных на базе других типов. Тип, на базе которого создается другой тип, называется родительским типом или *супертипом*. Подтипы наследуют характеристики своих супертипов.

Представьте, что вы создали пользовательский тип *phone_nbr* для хранения телефонных номеров. На базе этого типа можно создать подтипы *home_phone*,

work_phone, *cell_phone* и т. д. Каждый из подтипов унаследует характеристики родительских типов и при этом будет иметь и свои уникальные характеристики.

В этом примере мы создаем пользовательский тип *money* и несколько подтипов:

```
CREATE TYPE money (amount DECIMAL (10,2) )
  NOT FINAL;
CREATE TYPE dollar UNDER money AS DECIMAL(10,2)
  (conversion_rate DECIMAL(10,2) ) NOT FINAL;
CREATE TYPE euro UNDER money AS DECIMAL(10,2)
  (dollar_conversion_rate DECIMAL(10,2) ) NOT FINAL;
CREATE TYPE pound UNDER euro
  (euro_conversion_rate DECIMAL(10,2) ) FINAL;
```

Советы и хитрости

Пользовательские типы данных редко используются и слабо знакомы большинству разработчиков и администраторов баз данных. Поэтому часто проблемы возникают из-за обычного незнания. Тем не менее, пользовательские типы данных являются хорошим средством для представления повторно используемых сущностей базы данных, таких как адреса (т. е. *улица1*, *улица2*, *город*, *область*, *индекс*).

MySQL

Не поддерживается.

Oracle

В Oracle поддерживаются операторы *CREATE TYPE* и *ALTER TYPE*, но они не соответствуют стандарту ANSI. Вместо одного оператора *CREATE TYPE* используются два оператора: *CREATE TYPE* для спецификации атрибутов типа и *CREATE TYPE BODY* для объявления программного кода типа. Синтаксис *CREATE TYPE* следующий:

```
CREATE [OR REPLACE] TYPE имя_типа
{ [OID 'идентификатор_объекта']
[AUTHID {DEFINER | CURRENT_USER}]
  { {AS | IS} OBJECT | [UNDER имя_супертипа] |
  {OBJECT | TABLE OF тип_данных
  | VARRAY ( лимит ) OF тип_данных} }
    EXTERNAL NAME внешнее_имя_java
    LANGUAGE JAVA USING определение_данных
    { [ (атрибут тип_данных[, ...]) [EXTERNAL NAME 'имя'] |
    [[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]
    [{ {MEMBER | STATIC}
    { функция | процедура } |
    конструктор | сортировка } [...]]
    [фраза_pragma] ] } }
```

После того как тип объявлен, вся его логика инкапсулируется в теле типа. Имя типа, указанное в операторах *CREATE TYPE* и *CREATE TYPE BODY*, должно быть одинаковым. Синтаксис оператора *CREATE TYPE BODY* следующий:

```
CREATE [OR REPLACE] TYPE BODY имя_типа
{AS | IS}
```

```
{{MEMBER | STATIC}
{ функция | процедура | конструктор}}[...]
[сортировка]
```

Оператор *ALTER TYPE* в Oracle позволяет добавить или удалить атрибуты или методы типа:

```
ALTER TYPE имя_типа
{COMPILE [DEBUG] [{SPECIFICATION | BODY}]
[директивы_компилятора] [REUSE SETTINGS] |
REPLACE [AUTHID {DEFINER | CURRENT_USER}] AS OBJECT
(атрибут тип_данных[, ...] [определение_элемента[, ...]]) |
[[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]
{ {ADD | DROP} { {MAP | ORDER} MEMBER FUNCTION имя_функции
(параметр тия_данных[, ...]) } |
{ {MEMBER | STATIC} {функция | процедура} |
конструктор | сортировка [фраза_pragma] } [...] |
{ADD | DROP | MODIFY} ATTRIBUTE (атрибут [тип_данных][, ...]) |
MODIFY {LIMIT целое_число | ELEMENT TYPE тип_данных} }
[ {INVALIDATE |
CASCADE [{ [NOT] INCLUDING TABLE DATA | CONVERT TO SUBSTITUTABLE }]
[FORCE] [EXCEPTIONS INTO имя_таблицы]} ]}
```

Параметры трех этих операторов имеют следующие значения:

OR REPLACE

Пересоздает тип, если тип с таким именем уже существует. Все зависящие от типа объекты помечаются как *DISABLED*.

AUTHID {DEFINER|CURRENT_USER}

Определяет, с какими привилегиями выполняются члены-функции типа и как производится разрешение имен. (Имейте в виду, что подтипы наследуют способ определения привилегий от родительских типов.) Эта фраза используется только с объектными типами, но не с типами *VARRAY* и вложенными таблицами.

DEFINER

Выполняет члены-функции с привилегиями создателя типа. Также объекты, для которых не указана схема, будут искаться из схемы, в которой хранятся функции и процедуры.

CURRENT_USER

Выполняет члены-функции с привилегиями пользователя, использующего тип. Объекты, для которых не указана схема, будут искаться в схеме этого пользователя.

UNDER имя_супертипа

Указывает супертип для создаваемого типа. Супертип должен быть создан с фразой *AS OBJECT*. Подтип наследует все свойства супертипа, хотя некоторые из них нужно переопределить, а некоторые добавить, чтобы новый тип отличался от супертипа.

OID 'идентификатор_объекта'

Объявляет эквивалентный идентичный объект в более чем одной базе данных. Эта фраза используется в основном разработчиками, использующими

технологию Oracle Data Cartridges, и весьма редко применяется разработчиками обычного кода на SQL.

AS OBJECT

Создает корневой объектный тип (наивысший тип в иерархии объектов).

AS TABLE OF тип_данных

Создает именованный тип вложенной таблицы. Нельзя создавать вложенную таблицу типа *NCLOB*, хотя *BLOB* и *CLOB* допустимы. Если в качестве типа данных вложенной таблицы указывается объектный тип, то имена столбцов таблицы должны совпадать с именами атрибутов типа.

AS VARRAY (лимит) OF тип_данных

Создает тип данных, являющийся упорядоченным набором элементов одного и того же типа. Для набора указывается ограничение на длину. *VARRAY* можно создавать по встроенным типам данных, по типу *REF* и по объектным типам. *VARRAY* не может содержать объекты типа *LOB* и *XMLType*. Вместо *VARRAY* можно писать *VARYING ARRAY*.

EXTERNAL NAME внешнее_имя_java LANGUAGE JAVA USING определение_данных

Отображает класс Java с указанным именем на пользовательский тип данных SQL. Все экземпляры класса Java должны быть объектами Java. Данные могут быть объявлены при помощи SQLData, CustomDatum или OraData, как описано в «Руководстве разработчика Oracle9i JDBC Developers Guide». Вы можете отображать различные объекты Java на один и тот же класс, но есть два ограничения. Во-первых, нельзя отображать два и более подтипов одного типа на одинаковый класс. Во-вторых, подтипы должны быть отображены на непосредственный подкласс класса, на который отображен их супертип.

атрибут тип_данных

Определяет атрибуты и типы данных пользовательского типа. Атрибуты не могут иметь тип *ROWID*, *LONG*, *LONG RAW*, *UROWID*. Во вложенных таблицах и *VARRAY* не допускаются атрибуты *AnyType*, *AnyData*, *AnyDataSet*.

EXTERNAL NAME 'имя' [NOT] OVERRIDING

Указывает, что метод перегружает (*OVERRIDING*) или не перегружает (*NOT OVERRIDING*) метод родительского типа. Используется только во фразах *MEMBER*.

MEMBER|STATIC

Определяет способ, которым подпрограммы связываются с типом данных. *MEMBER* имеет неявный первый атрибут *SELF*, как в *object_expression.method()*. *STATIC* не имеет неявных аргументов, как в *type_name.method()*. Для вызова подпрограмм используются следующие варианты:

функция

Определяет подпрограмму, основанную на функции:

```
FUNCTION имя_функции (параметр тип_данных[, ...])
фраза_return | объект_java
```

Эта фраза позволяет создать тело пользовательского типа на базе функции PL/SQL без использования оператора *CREATE TYPE BODY*. Имя функ-

ции не может совпадать с именем какого-либо атрибута, в том числе атрибута супертипа. Фразы *return* и *объект_java* рассматриваются ниже.

процедура

Определяет подпрограмму, основанную на процедуре:

```
PROCEDURE имя_процедуры (параметр тип_данных [, ...])
{AS | IS} LANGUAGE
  {спецификация_вызова_java | спецификация_вызова_C}
```

Эта фраза позволяет создать тело пользовательского типа на базе процедуры PL/SQL без использования оператора *CREATE TYPE BODY*. Имя процедуры не может совпадать с именем какого-либо атрибута, в том числе атрибута супертипа. Фразы *спецификация_вызова_java* и *спецификация_вызова_C* рассматриваются ниже.

конструктор

Объявляет один или несколько конструкторов с использованием следующего синтаксиса:

```
[FINAL] [INSTANTIABLE] CONSTRUCTOR FUNCTION тип_данных
[ ( [SELF IN OUT тип_данных,]
     параметр тип_данных [, ...] ) ]
RETURN SELF AS RESULT
[ {AS | IS} LANGUAGE
  {спецификация_вызова_java |
   спецификация_вызова_C} ]
```

Конструктор – это функция, возвращающая инициализированный экземпляр пользовательского типа данных. Спецификации конструктора всегда объявляются как *FINAL*, *INSTANTIABLE* и *SELF IN OUT*, так что эти ключевые слова необязательны. Подфразы *спецификация_вызова_java* и *спецификация_вызова_C* (рассматриваются позже в этом разделе) можно заменить блоком кода PL/SQL в операторе *CREATE TYPE BODY*.

сортировка

Задает правило сортировки или упорядочения супертипа с использованием следующего синтаксиса:

```
{MAP | ORDER} MEMBER функция
```

MAP использует более эффективные алгоритмы сравнения объектов, поэтому больше подходит при интенсивных сортировках и хеш-соединениях. *MAP* указывает относительную позицию заданного экземпляра при сортировке всех экземпляров пользовательского типа. *ORDER* возвращает явную позицию экземпляра с помощью функции, возвращающей тип *NUMBER*.

фраза_return

Определяет формат возвращаемого значения пользовательского типа данных с использованием следующего синтаксиса:

```
RETURN тип_данных [ {AS | IS} LANGUAGE
  {спецификация_вызова_java | спецификация_вызова_C}
```

обьект_java

Определяет формат возвращаемого значения пользовательского типа Java с использованием следующего синтаксиса:

```
RETURN { тип_данных | SELF AS RESULT} EXTERNAL [VARIABLE] NAME 'имя_java'
```

Если вы используете фразу *EXTERNAL*, то значение публичного метода Java должно быть совместимым со значением, возвращаемым в SQL.

фраза_pragma

Определяет для типа директивы прекомпилятора с использованием следующего синтаксиса:

```
PRAGMA RESTRICT REFERENCES ( {DEFAULT | имя_метода},
{RNDS | WNDS |RNPS | WNPS | TRUST}[, ...] )
```

Эта фраза устарела и ее использования следует избегать. С ее помощью описывается, каким образом пользовательский тип читает и изменяет данные в таблицах и переменных базы данных.

DEFAULT

Устанавливает настройки прекомпилятора, используемые по умолчанию для всех методов пользовательского типа.

имя_метода

Указывает, к какому методу применять директиву

RNDS

Не читает из базы данных – чтения из базы данных запрещены.

WNDS

Не пишет в базу данных – запись в базу данных запрещена.

RNPS

Не читает пакеты – чтение пакетов запрещено.

WNPS

Не пишет в пакеты – запись в пакеты запрещена.

TRUST

Указывает, что корректность используемых директив не проверяется.

спецификация_вызова_java

Указывает реализацию метода на Java в формате *JAVA NAME 'имя_java'*. Это позволяет вам определять тело типа на Java без использования оператора *CREATE TYPE BODY*.

спецификация_вызова_C

Определяет спецификацию вызова языка C с использованием следующего синтаксиса:

```
C [NAME имя] LIBRARY имя_библиотеки [AGENT IN (аргумент)]
[WITH CONTEXT] [PARAMETERS (параметр[, ...])]
```

Эта фраза позволяет вам определять тело типа на C без использования оператора *CREATE TYPE BODY*.

COMPILE

Компилирует спецификацию и тело объектного типа. Это является поведением по умолчанию, если не используется ни фраза *SPECIFICATION*, ни фраза *BODY*.

DEBUG

Генерирует и добавляет к телу типа дополнительный код для PL/SQL отладчика. Не используйте одновременно *DEBUG* и директиву прекомпилятора *PLSQL_DEBUG*.

SPECIFICATION | BODY

Указывает, что необходимо перекомпилировать либо спецификацию (*SPECIFICATION*) объектного типа (созданную оператором *CREATE TYPE*) либо тело типа (*BODY*, создаваемое оператором *CREATE TYPE BODY*).

директивы_компилятора

Задает значения для директив компилятора в формате *директива='значение'*. Директивы компилятора следующие: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS* и *NLS_LENGTH_SEMANTICS*. Для каждой директивы можно указать только одно значение. Значение директивы действует только для компилируемого объекта.

REUSE SETTINGS

Сохраняет исходные значения директив компилятора.

REPLACE AS OBJECT

Добавляет в спецификацию новый подтип. Эта фраза используется только с объектными типами.

[NOT] OVERRIDING

Указывает, что метод переопределяет или не переопределяет (*NOT*) метод супертипа. Эта фраза применяется только с методами и требуется для методов, переопределяющих методы супертипа. *NOT OVERRIDING* является значением по умолчанию.

ADD

Добавляет в пользовательский тип новый метод (подпрограмму, основанную на процедуре или функции) или атрибут.

DROP

Удаляет из пользовательского типа метод (подпрограмму, основанную на процедуре или функции) или атрибут.

MODIFY

Изменяет свойства существующего атрибута пользовательского типа данных.

MODIFY LIMIT целое_число

Увеличивает число элементов в коллекции VARRAY до указанного значения, при условии, что новое значение больше текущего. Не используется для вложенных таблиц.

MODIFY ELEMENT TYPE тип_данных

Увеличивает точность, размер или длину скалярного типа данных вложенной таблицы или *VARRAY*. Это фраза используется с необъектными типами. Если коллекция построена на типе *NUMBER*, то вы можете увеличить точность и масштаб. Если на типе *RAW*, то вы можете увеличить максимальный размер. Если на типе *VARCHAR2* и *NVARCHAR2*, то вы можете увеличить максимальную длину.

INVALIDATE

Без проверок делает невалидными все зависимые объекты.

CASCADE

Каскадно распространяет изменения на все подтипы и таблицы. По умолчанию действие откатывается, если возникают ошибки в зависимых типах и таблицах.

[NOT] INCLUDING TABLE DATA

Преобразует или не преобразует (*NOT*) данные в столбцах пользовательского типа в наиболее актуальную версию типа столбца. Если указано *NOT*, то Oracle только проверяет метаданные, но не проверяет и не обновляет данные в зависимых таблицах.

CONVERT TO SUBSTITUTABLE

Используется при преобразовании типа из *FINAL* в *NOT FINAL*. Измененный тип может использоваться в подстановочных таблицах и столбцах, а также в подтипах, экземплярах зависимых таблиц и столбцов.

FORCE

При использовании *CASCADE* продолжает операцию даже при наличии ошибок в зависимых подтипах и таблицах. Все возникшие ошибки журналируются в предварительно созданную таблицу исключений.

EXCEPTIONS INTO таблица

Записывает сообщения об ошибках в специальную таблицу, предварительно созданную при помощи системного пакета **DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE**.

В качестве примера мы создадим объектный тип с названием *address_type* на базе класса Java:

```
CREATE TYPE address_type AS OBJECT
  EXTERNAL NAME 'scott.address' LANGUAGE JAVA
  USING SQLDATA (
    street1       VARCHAR(30) EXTERNAL NAME 'str1',
    street2       VARCHAR(30) EXTERNAL NAME 'str2',
    city          VARCHAR(30) EXTERNAL NAME 'city',
    state         CHAR(2) EXTERNAL NAME 'st',
    locality_code CHAR(15) EXTERNAL NAME 'lc',
  STATIC FUNCTION square_feet RETURN NUMBER
    EXTERNAL VARIABLE NAME 'square_feet',
  STATIC FUNCTION create_addr (str VARCHAR,
    City VARCHAR, state VARCHAR, zip NUMBER)
    RETURN address_type
```

```
    EXTERNAL NAME 'create (java.lang.String,
      java.lang.String, java.lang.String, int)
      return scott.address',
  MEMBER FUNCTION rtrims RETURN SELF AS RESULT
    EXTERNAL NAME 'rtrim_spaces ( ) return scott.address' )
NOT FINAL;
```

В следующем примере мы создаем пользовательский тип на базе коллекции *VARRAY* из четырех элементов:

```
CREATE TYPE employee_phone_numbers AS VARRAY(4) OF CHAR(14);
```

В следующем примере мы добавляем в тип *address_type* атрибут *phone_array*, являющийся коллекцией *VARRAY*:

```
ALTER TYPE address_type
  ADD ATTRIBUTE (phone phone_varray) CASCADE;
```

В последнем примере мы создает супертип и подтип с названиями *menu_item_type* и *entry_type*:

```
CREATE OR REPLACE TYPE menu_item_type AS OBJECT
(id INTEGER, title VARCHAR2(500),
  NOT INSTANTIABLE
  MEMBER FUNCTION fresh_today
  RETURN BOOLEAN)
NOT INSTANTIABLE
NOT FINAL;
```

В предыдущем примере мы создали спецификацию (но не тело) типа, определяющего объекты меню в кафе. В спецификацию типа включен метод *fresh_today*, возвращающий индикатор того, что меню составлено в тот же день. Фраза *NOT FINAL* в конце спецификации говорит, что тип может выступать в качестве супертипа, поэтому далее мы создадим на его базе подтип *entry_type*:

```
CREATE OR REPLACE TYPE entry_type UNDER menu_item_type
(entre_id INTEGER, desc_of_entre VARCHAR2(500),
  OVERRIDING MEMBER FUNCTIOIN fresh_today
  RETURN BOOLEAN)
NOT FINAL;
```

PostgreSQL

В PostgreSQL поддерживаются оба оператора – *CREATE TYPE* и *ALTER TYPE*. Реализация *ALTER TYPE* не соответствует стандарту:

```
CREATE TYPE имя_типа
( INPUT = функция_ввода,
OUTPUT = функция_вывода
[, INTERNALLENGTH = { целое_число | VARIABLE } ]
[, DEFAULT = значение ]
[, ELEMENT = тип_элемента_массива ]
[, DELIMITER = разделитель ]
[, PASSEDBYVALUE ]
[, ALIGNMENT = {CHAR | INT2 | INT4 | DOUBLE} ]
[, STORAGE = {PLAIN | EXTERNAL | EXTENDED | MAIN} ]
```

```
[, SEND = функция_посылки ]
[, RECEIVE = функция_получения ]
[, ANALYZE = функция_анализа ] )
```

С помощью оператора *ALTER TYPE* можно поменять владельца и схему существующего типа:

```
ALTER TYPE имя_типа [OWNER TO новый_владелец]
  [SET SCHEMA новая_схема]
```

Параметры следующие:

CREATE TYPE имя_типа

Создает новый пользовательский тип с указанным именем. Название не может быть длиннее 30 символов в длину, а также не может начинаться с подчеркивания.

INPUT = функция_ввода

Определяет имя существующей функции, конвертирующей значения аргументов во внутренний формат представления типа.

OUTPUT = функция_вывода

Определяет имя существующей функции, конвертирующей внутренний формат представления типа в формат отображения.

INTERNALLENGTH = { целое_число| VARIABLE }

Определяет длину внутреннего представления типа для типов с фиксированной длиной. Ключевое слово *VARIABLE* (значение по умолчанию) указывает, что тип имеет переменную длину.

DEFAULT = значение

Определяет для типа значение по умолчанию.

ELEMENT = тип_элемента_массива

Указывает, что пользовательский тип данных представляет собой массив из элементов указанного типа. Например, массив целочисленных значений следует объявлять как *ELEMENT=INT4*. Как правило, вы должны использовать для элементов массива значения по умолчанию. Единственный случай, когда вам потребуется переопределить значения по умолчанию, это создание пользовательского типа фиксированной длины, состоящего из массива идентичных элементов, пригодных для индексации.

DELIMITER = разделитель

Определяет символ, являющийся разделителем элементов массива в отображаемом формате. Используется только вместе с фразой *ELEMENT*. По умолчанию используется запятая.

PASSEDBYVALUE

Указывает, что значения типа передаются по значению, а не ссылке. Эта фраза опциональна и не может использоваться для типов, имеющих длину больше, чем у типа *DATUM* (4 байта в большинстве операционных систем, 8 байт в остальных).

ALIGNMENT = {CHAR | INT2 | INT4 | DOUBLE}

Определяет выравнивание при хранении типа. Каждому из используемых типов соответствуют свои границы: *CHAR* – 1 байт, *INT2* – 2 байта, *INT4* – 4 байта, *DOUBLE* – 8 байт.

STORAGE = {PLAIN | EXTERNAL | EXTENDED | MAIN}

Определяет способ хранения пользовательских типов с переменной длиной (для типов с фиксированной длиной используется *PLAIN*).

PLAIN

Если пользовательский тип используется для столбца в таблице, то его данные хранятся вместе с другими данными таблицы в несжатом виде.

EXTERNAL

Хранит данные пользовательского типа вне данных таблицы, в несжатом виде.

EXTENDED

Сжимает данные пользовательского типа и хранит их вместе с данными таблицы, если значение достаточно невелико. При большой длине значений PostgreSQL хранит их вне таблицы.

MAIN

Хранит данные пользовательского типа в сжатом виде внутри таблицы. Имейте в виду, что иногда бывают ситуации, что пользовательский тип нельзя сохранить в таблице ввиду его большого значения. Но при использовании *MAIN* данные пользовательского типа будут храниться в таблице всегда, когда это возможно.

SEND = функция_посылки

Преобразует внутреннее представление типа во внешнее двоичное представление. Обычно программируется на C или другом низкоуровневом языке.

RECEIVE = функция_получения

Преобразует внешнее двоичное представление во внутреннее представление типа. Обычно программируется на C или другом низкоуровневом языке.

ANALYZE = функция_анализа

Собирает специфическую статистику по столбцам, использующим этот тип.

Дополнительная информация по функциям *SEND*, *RECEIVE* и *ANALYZE* доступна в документации по PostgreSQL.

При создании в PostgreSQL нового типа он доступен только в текущей базе данных. Владельцем типа становится пользователь, который его создал. Большинство параметров при создании типа могут идти в любом порядке и являются необязательными (кроме функций ввода и вывода).

Перед созданием типа вам нужно создать как минимум две функции. Вы должны создать функцию ввода (*INPUT*), заполняющую тип внешними значениями, а также функцию вывода (*OUTPUT*), создающую пригодное к использованию

внешнее представление значения типа. Есть еще несколько требований к функциям ввода и вывода:

- Функция ввода должна либо принимать один аргумент тип *OPAQUE*, либо три аргумента типов *OPAQUE*, *OID* и *INT4*. Во втором случае аргумент типа *OPAQUE* является входной строкой С, аргумент типа *OID* определяет тип данных для элементов массива, а аргумент типа *INT4* определяет режим целевого столбца.
- Функция вывода должна принимать на вход либо аргумент типа *OPAQUE*, либо два аргумента типов *OPAQUE* и *OID*. В последнем случае аргумент *OPAQUE* представляет сам тип данных, а аргумент *OID* – тип данных элементов массива, если требуется.

Например, мы можем создать тип *floorpan* и использовать его при объявлении столбцов таблиц *house* и *condo*:

```
CREATE TYPE floorplan
  (INTERNALLENGTH=12, INPUT=squarefoot_calc_proc,
  OUTPUT=out_floorplan_proc);
CREATE TABLE house
  (house_plan_id int4,
  size           floorplan,
  descrip        varchar(30) );
CREATE TABLE condo
  (condo_plan_id INT4,
  size           floorplan,
  descrip        varchar(30)
  location_id    varchar(7) );
```

SQL Server

SQL Server поддерживает оператор *CREATE TYPE*, но не *ALTER TYPE*. Новые типы данных можно создавать также, используя системную хранимую процедуру **sp_addtype**. Синтаксис реализации оператора *CREATE TYPE* следующий:

```
CREATE TYPE имя_типа
{ FROM базовый_тип [ ( точность [, масштаб])] [[NOT] NULL] |
AS TABLE определение_таблицы |
CLR_определение }
```

где:

FROM *базовый_тип*

Указывает тип, на базе которого создается новый тип. Базовым типом может быть любой из следующих типов: *BIGINT*, *BINARY*, *BIT*, *CHAR*, *DATE*, *DATETIME*, *DATETIME2*, *DATETIMEOFFSET*, *DECIMAL*, *FLOAT*, *IMAGE*, *INT*, *MONEY*, *NCHAR*, *NTEXT*, *NUMERIC*, *NVARCHAR*, *REAL*, *SMALLDATETIME*, *SMALLINT*, *SMALLMONEY*, *SQL_VARIANT*, *TEXT*, *TIME*, *TINYINT*, *UNIQUEIDENTIFIER*, *VARBINARY* и *VARCHAR*. Можно указывать точность и масштаб для тех типов, где это применимо.

[NOT] NULL

Указывает, допустимы ли для типа значения *NULL*. По умолчанию пустые значения допустимы.

AS TABLE определение_таблицы

Определяет пользовательский табличный тип с описаниями столбцов, типов данных, ключей, ограничений (*CHECK*, *UNIQUE* и *PRIMARY KEY*) и свойств (таких как *CLUSTERED* и *NONCLUSTERED*), как для обычной таблицы.

SQL Server поддерживает создание типов, написанных на Microsoft .NET Framework CLR. Эти типы создаются и изменяются так же, как и обычные типы SQL, однако телами таких типов являются внешние сборки. Информацию по созданию пользовательских типов на CLR ищите в документации.

Типам, созданным с помощью процедуры **sp_addtype**, доступны роли *PUBLIC*. Права на использование пользовательских типов, созданных оператором *CREATE TYPE*, необходимо выдавать явно.

См. также

CREATE/ALTER FUNCTION/PROCEDURE
DROP

CREATE/ALTER VIEW

С помощью этих операторов создаются и модифицируются представления (иногда называемые также виртуальными таблицами). Представление используется в запросах как обычная таблица, но создается как запрос. Когда представление используется в операторе, то результат выполнения этого запроса становится содержимым представления на время выполнения этого оператора. Практически любой корректный оператор *SELECT* можно использовать для создания представления, хотя в некоторых платформах существуют определенные ограничения.

В некоторых случаях представления могут быть обновляемыми, при этом оператор *UPDATE* для представления транслируется в операторы *UPDATE* для тех таблиц, на базе которых создано представление. Некоторые платформы поддерживают материализованные представления, которые физически являются таблицами, определяемыми с помощью запроса.

Оператор *ALTER VIEW* не поддерживается стандартом ANSI.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
CREATE [RECURSIVE] VIEW имя_представления
{[(столбец[, ...])] |
```

```
[OF имя_udt [UNDER имя_супертипа [REF IS имя_столбца
    {SYSTEM GENERATED | USER GENERATED | DERIVED}]
[имя_столбца WITH OPTIONS SCOPE имя_таблицы]]]}
    AS оператор_select [WITH [CASCADED|LOCAL] CHECK OPTION]
```

Ключевые слова

CREATE VIEW *имя_представления*

Создает новое представление с заданным именем.

RECURSIVE

Указывает, что представление является рекурсивным, то есть извлекает данные из самого себя. В рекурсивном представлении должны содержаться определения столбцов и нельзя использовать фразу *WITH*.

[(столбец [, ...])]

Именует столбцы представления. Количество имен столбцов должно совпадать с количеством столбцов в операторе *SELECT* представления. Если имена не указаны, то они формируются из оператора *SELECT*. Указание имен обязательно, если некоторые столбцы представления вычисляются при помощи выражений, а не напрямую извлекаются из таблицы.

OF *имя_udt* *[UNDER* *имя_супертипа]*

Создает представление на базе пользовательского типа данных. Для каждого атрибута типа создается столбец в представлении. Для создания представления по подтипу используйте фразу *UNDER*.

REF IS *имя_столбца* *{SYSTEM GENERATED | USER GENERATED | DERIVED}*

Определяет столбец представления, являющийся объектным идентификатором.

имя_столбца *WITH OPTIONS SCOPE* *имя_таблицы*

Определяет область действия ссылочного столбца представления. (Так как столбцы представления формируются из атрибутов типа, то список столбцов не указывается. Поэтому для указания параметров столбцов используется фраза *имя_столбца* *WITH OPTIONS*.)

AS *оператор_select*

Определяет оператор *SELECT*, используемый для формирования данных представления.

WITH [CASCADED | LOCAL] CHECK OPTION

Используется только с обновляемыми представлениями. Проверяет, что через представление могут быть вставлены, изменены или удалены только те данные, которые доступны представлению. Например, если представление **employees** показывает зарплаты только сотрудников, работающих за фиксированный оклад, то через это представление нельзя добавить, обновить или удалить записи о сотрудниках с почасовой оплатой труда. Опции *CASCADED* и *LOCAL* используются для вложенных представлений. *CASCADED* выполняет проверку *CHECK OPTION* для текущего представления и для всех представлений, на которых текущее представление построено. *LOCAL* выполняет проверку только для текущего представления.

Общие правила

Представления эффективны настолько, насколько эффективны запросы, используемые в представлениях. Вот почему важно в представлениях использовать корректно написанные, быстрые запросы. Самое простое представление выводит полностью содержимое одной таблицы:

```
CREATE VIEW employees
AS
SELECT *
FROM employee_tbl;
```

После имени представления можно указать список столбцов. Этот список содержит псевдонимы для столбцов результирующего набора оператора *SELECT*. Если вы используете список столбцов, то вы должны указать псевдоним для каждого столбца оператора *SELECT*. Если вы не используете список столбцов, то имена столбцов представления будут такими же, как имена столбцов в операторе *SELECT*. Иногда вы можете видеть в представлениях операторы *SELECT*, использующие фразу *AS*: это позволяет разработчику представления дать столбцу осмысленное имя без указания в представлении полного списка столбцов.

Стандарт ANSI требует либо использовать список столбцов, либо фразу *AS*. Однако некоторые платформы предоставляют большую гибкость, поэтому применяйте *AS* в следующих случаях:

- Когда оператор *SELECT* содержит вычисляемый столбец, например (*salary*1.04*).
- Когда оператор *SELECT* содержит столбцы с указанием полного имени, например **pubs.scott.employee**.
- Когда оператор *SELECT* содержит несколько столбцов с одинаковыми именами (но из разных схем или баз данных).

Например, следующие два оператора дают в итоге идентичные представления:

```
-- Используется список столбцов
CREATE VIEW title_and_authors
  (title, author_order, author, price, avg_monthly_sales,
  publisher)
AS
SELECT t.title, ta.au_ord, a.au_lname, t.price,
  (t.ytd_sales / 12),t.pub_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0;

-- Используется AS для каждого столбца
CREATE VIEW title_and_authors
AS
SELECT t.title AS title, ta.au_ord AS author_order,
  a.au_lname AS author, t.price AS price,
  (t.ytd_sales / 12) AS avg_monthly_sales,
  t.pub_id AS publisher
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
```

```
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0
```

С помощью списка столбцов можно менять имена столбцов запроса. В следующем примере мы меняем **avg_monthly_sales** на **avg_sales**. Обратите внимание, что значения из списка столбцов имеют приоритет над тем, что указано во фразе *AS* (выделено жирным):

```
CREATE VIEW title_and_authors
  (title, author_order, author, price, avg_sales, publisher)
AS
SELECT t.title AS title, ta.au_ord AS author_order,
  a.au_lname AS author, t.price AS price,
  (t.ytd_sales / 12) AS avg_monthly_sales,
  t.pub_id AS publisher
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0;
```

По стандарту ANSI через представление можно обновлять базовые таблицы, если оно удовлетворяет следующим критериям:

- Представление не содержит операторов *UNION*, *EXCEPT* и *INTERSECT*.
- Оператор *SELECT* не содержит фраз *GROUP BY* и *HAVING*.
- Оператор *SELECT* не использует нестандартные псевдостолбцы, такие как *ROWNUM* или *ROWGUIDCOL*.
- Оператор *SELECT* не использует *DISTINCT*.
- Представление не является материализованным.

Следующий пример демонстрирует представление **california_authors**, допускающее обновление записей только авторов из Калифорнии:

```
CREATE VIEW california_authors
AS
SELECT au_lname, au_fname, city, state
FROM authors
WHERE state = 'CA'
WITH LOCAL CHECK OPTION
```

Показанное представление допускает применение операторов *INSERT*, *UPDATE* и *DELETE* только для записей, в которых поле **state** имеет значение ‘CA’. Это поведение достигается за счет использования фразы *WITH CHECK OPTION*.

При использовании обновляемых представлений важно помнить, что в базовых таблицах могут использоваться столбцы с ограничениями *NOT NULL* и в эти столбцы нельзя записывать пустое значение при обновлении или вставке в таблицу через представление. Поэтому нужно или явно вставлять в такие столбцы непустые значения, или полагаться на значения по умолчанию. Также представления не отменяют ограничения базовых таблиц. Все обновляемые или вставляемые в базовые таблицы значения должны удовлетворять ограничениям этих таблиц, реализованным через первичные ключи, уникальные индексы и т. д.

Советы и хитрости

Представления можно создавать на основе других представлений, хотя такой подход не рекомендуется и считается плохой практикой. В зависимости от платформы такие представления могут дольше компилироваться, но производительность таких представлений будет такой же, как и при операциях над базовыми таблицами. В других платформах, где представления создаются динамически при вызовах, запросы к вложенным представлениям могут очень долго возвращать результат, так как каждый уровень вложенности означает полное выполнение отдельного запроса. В самом худшем случае запрос к представлению с третьим, к примеру, уровнем вложенности может потребовать выполнения трех коррелированных подзапросов.

Материализованные представления, хоть и объявляются как обычные представления, после создания занимают место для хранения данных как обычные таблицы. Перед созданием материализованного представления убедитесь, что имеется достаточно свободного пространства.

MySQL

В MySQL поддерживаются операторы *CREATE VIEW* и *ALTER VIEW*. Но MySQL не поддерживает рекурсивных представлений, представлений с пользовательскими типами данных и столбцами *REF*. Синтаксис обоих операторов следующий:

```
{ALTER | CREATE [OR REPLACE]}
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
[DEFINER = {имя_пользователя | CURRENT_USER}]
[SQL SECURITY {DEFINER | INVOKER}]
VIEW имя_представления [(столбец[, ...])]
AS оператор_select
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

где:

ALTER | CREATE [OR REPLACE]

Изменяет существующее или создает новое представление. Если указано *OR REPLACE*, то если уже существует представление с указанным именем, оно заменяется на новое.

ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}

Определяет, каким образом MySQL будет работать с представлением. *MERGE* объединяет запрос представления с запросом, использующим это представление, а затем генерирует оптимальный план выполнения получившегося запроса. *TEMPTABLE* означает, что результат выполнения запроса помещается во временную таблицу, а затем запрос к представлению использует получившуюся временную таблицу. При *UNDEFINED* (значение по умолчанию) MySQL самостоятельно выбирает оптимальный вариант работы с представлением.

DEFINER = {имя_пользователя | CURRENT_USER}

Указывает учетную запись, используемую для проверки привилегий. Вы можете либо явно указывать существующего пользователя, либо указать текущего пользователя (*CURRENT_USER*), выполняющего *CREATE VIEW*. Если вы не укажете ничего, то по умолчанию будет подразумеваться *CURRENT_USER*.

SQL SECURITY {DEFINER|INVOKER}

Определяет контекст безопасности, в котором работает представление: либо это контекст пользователя, создавшего представление (*DEFINER*, значение по умолчанию), либо это контекст пользователя, использующего представление в своем запросе (*INVOKER*).

В MySQL имена таблиц и представлений не могут совпадать, так как находятся в одном пространстве имен.

Oracle

В Oracle поддерживаются расширения стандарта ANSI, позволяющие создавать объектно-ориентированные представления, представления типа XMLType, представления с поддержкой *LOB* и объектных типов:

```
CREATE [OR REPLACE] [[NO] FORCE] VIEW имя_представления
  {[ (столбец[, ...]) [ограничения] ] |
  [OF имя_типа {UNDER родительское_представление |
    WITH OBJECT IDENTIFIER {DEFAULT | (атрибут[, ...])}
    [(ограничения)]}] |
  [OF XMLTYPE [ [XMLSCHEMA url_xml_схема] ELEMENT
    {элемент | url_xml_схема # элемент } ]
    WITH OBJECT IDENTIFIER {DEFAULT | (атрибут[, ...])}] ]}
AS
(оператор_select)
[WITH [READ ONLY | CHECK OPTION
  [CONSTRAINT имя_ограничения]]]
```

Оператор *ALTER VIEW* поддерживает дополнительные возможности, такие как добавление, изменение и удаление ограничений, связанных с представлением. Также с помощью этого оператора можно явно перекомпилировать представление, что позволяет обнаружить ошибки компиляции до использования представления в запросах. Явная перекомпиляция позволяет определить, будет ли изменение базовых таблиц влиять на представление:

```
ALTER VIEW имя_представления
{ADD ограничения |
MODIFY CONSTRAINT ограничения [NO]RELY]] |
DROP {PRIMARY KEY | CONSTRAINT ограничения | UNIQUE (столбец[, ...])}}
COMPILE
```

Параметры имеют следующие значения:

OR REPLACE

Заменяет существующее представление новым.

[NO] FORCE

FORCE позволяет создать представление вне зависимости от того, существуют ли таблицы и функции, используемые в представлении, и достаточно ли привилегий на чтение и запись в эти таблицы. Также *FORCE* позволяет создавать представление при наличии ошибок компиляции. При использовании *NO FORCE* представление создается, только если все используемые в представлении объекты существуют и достаточно привилегий для их использований.

ограничения

Дает возможность создавать ограничения для представлений (в операторе *CREATE VIEW*) или изменять именованные ограничения (в операторе *ALTER VIEW*). Вы можете создавать ограничения как на уровне представления, так и на уровне отдельных столбцов представления. Имейте в виду, что хотя Oracle и позволяет создавать ограничения для представления, выполнение этих ограничений не проверяется. Ограничения представлений создаются в режимах *DISABLE* и *NOVALIDATE*.

OF имя_типа

Позволяет создать представление объектного типа. Столбцы представления однозначно соответствуют атрибутам типа. Имена столбцов для объектных представлений и представлений XMLType указывать не нужно.

UNDER родительское_представление

Указывает, что создаваемое представление является потомком указанного родительского представления. Представление должно находиться в той же схеме, что и родительское, а *имя_типа* (по которому создается представление) должно быть непосредственным потомком типа данных *родительского_представления*. У представления может быть только один потомок.

WITH OBJECT IDENTIFIER {DEFAULT | (атрибут [, ...])}

Определяет корневое объектное представление, а также атрибуты объектного типа, идентифицирующие каждую строку в объектном представлении. Эти атрибуты обычно соответствуют столбцам первичного ключа базовой таблицы и должны уникально идентифицировать каждую строку в представлении. Эта фраза несовместима с вложенными представлениями и с разыменованными и закрепленными (dereferenced or pinned) объектными ссылками. Ключевое слово *DEFAULT* обозначает использование существующего идентификатора базовой объектной таблицы или представления.

OF XMLTYPE [[XMLSCHEMA url_xml_схемы] ELEMENT {элемент | url_xml_схемы # элемент}] WITH OBJECT IDENTIFIER {DEFAULT | (атрибут[, ...])}

Объявляет, что представление возвращает экземпляры типа XMLType. Указание опционального *url_xml_схемы* и названия элемента дополнительно ограничивает возвращаемый XML как элемент в этой XML-схеме. Фраза *WITH OBJECT IDENTIFIER* определяет уникальный идентификатор XMLType-представления, который может состоять из одного или нескольких атрибутов и неагрегатных функций, например *EXTRACTVALUE*.

WITH READ ONLY

Создает представление, используемое только для чтения, но не для модификации данных.

WITH CHECK OPTION [CONSTRAINT имя_ограничения]

Гарантирует, что через представление можно будет вставить или изменить только те данные, которые будут возвращаться оператором *SELECT* представления. При желании вы можете указать имя этого ограничения. Если вы не укажете имя, то Oracle сгенерирует имя **SYS_Cn**, где n – целое число.

ADD ограничение

Добавляет новое ограничение для представления. Ограничения представлений создаются только в режимах *DISABLE* и *NOVALIDATE*.

MODIFY CONSTRAINT ограничение [NO]RELY

Изменяет параметр *RELY* или *NORELY* существующего ограничения представления. (*RELY/NORELY* описывается в разделе, посвященном оператору *CREATE TABLE*.)

DROP {PRIMARY KEY|CONSTRAINT ограничение |UNIQUE (столбец[, ...])}

Удаляет существующее ограничение представления.

COMPILE

Перекомпилирует представление.

Любые указатели (dblinks) на базы данных, используемые в представлении, должны быть предварительно созданы оператором C*REATE DATABASE LINK... CONNECT TO*. Если представление содержит ретроспективный (flashback) запрос, то фраза *AS OF* будет вычисляться при каждом вызове представления, а не при каждой его компиляции.

В этом примере мы создадим представление с ограничением:

```
CREATE VIEW california_authors (last_name, first_name,
  author_ID UNIQUE RELY DISABLE NOVALIDATE,
  CONSTAINT id_pk PRIMARY KEY (au_id)
    RELY DISABLE NOVALIDATE)
AS
SELECT au_lname, au_fname, au_id
FROM authors
WHERE state = 'CA';
```

В следующем примере создается пользовательский тип данных и объектное представление:

```
CREATE TYPE inventory_type AS OBJECT
( title_id NUM(6),
  warehouse wrhs_typ,
  qty NUM(8) );

CREATE VIEW inventories OF inventory_type
WITH OBJECT IDENTIFIER (title_id)
AS
SELECT i.title_id, wrhs_typ(w.wrhs_id, w.wrhs_name,
  w.location_id), i.qty
FROM inventories i
JOIN warehouses w ON i.wrhs_id = w.wrhs_id;
```

Мы могли бы перекомпилировать представление следующим образом:

```
ALTER VIEW inventories COMPILE:
```

Обновляемые представления в Oracle не могут содержать следующие элементы:

- Фразу *DISTINCT*.
- Фразы *UNION*, *INTERSECT*, *MINUS*.

- Соединения, при которых вставка или обновление представления влияют более чем на одну таблицу.[1]
- Агрегатные и аналитические функции.
- Фразу *GROUP BY*, *ORDER BY*, *CONNECT BY* и *START WITH*.
- Подзапросы и выражения с коллекциями в списке *SELECT* (подзапросы допускаются во фразе *WHERE*).
- Обновление псевдостолбцов и выражений.

Есть определенные ограничения на создание материализованных представлений и представлений-потомков:

- В представлениях-потомках должны быть объявлены синонимы для псевдостолбцов *ROWID*, *ROWNUM* и *LEVEL*.
- В представлениях-потомках нельзя использовать псевдостолбцы *CURRVAL* и *NEXTVAL*.
- В представлениях-потомках нельзя использовать фразу *SAMPLE*.
- Все столбцы в *SELECT * FROM* определяются при компиляции представления. Поэтому любые столбцы, добавляемые впоследствии к базовым таблицам, не будут извлекаться представлением до его перекомпиляции.

PostgreSQL

В PostgreSQL поддерживается только минимальный набор функций оператора *CREATE VIEW*:

```
CREATE [OR REPLACE] [TEMP[ORARY]]
  VIEW имя_представления [ (столбец[, ...]) ]
AS оператор_select
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

PostgreSQL не поддерживает оператор *ALTER VIEW*, но вы можете заменить определение старого представления новым определением, используя *CREATE OR REPLACE*. В дополнение PostgreSQL позволяет создавать временные представления, что расширяет стандарт SQL3.

CREATE VIEW в PostgreSQL не поддерживает множество возможностей, поддерживаемых в других платформах, но поддерживает создание представлений на базе таблиц и других объектов классов. Представления всегда создаются только для чтения и не позволяют модифицировать данные в базовых таблицах.

SQL Server

SQL Server предлагает некоторые расширения стандарта ANSI, но не поддерживает объектных представлений и представлений-потомков:

```
CREATE VIEW имя_представления [(столбец[, ...])]
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA}[, ...]]
AS оператор_select
[WITH CHECK OPTION]
```

---

[1] Правильнее было бы сказать, что DML-операция, выполняемая для представления, должна изменять только одну базовую таблицу. – *Прим. науч. ред.*

Оператор *ALTER VIEW* позволяет изменить представление, не затрагивая привилегии и зависимые объекты:

```
ALTER VIEW имя_представления [(столбец[, ...])]
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA}[, ...]]
AS оператор_select
[WITH CHECK OPTION]
```

Параметры имеют следующие значения:

ENCRYPTION

Шифрует текст представления, хранящийся в системной таблице **syscomments**. Эта опция обычно используется разработчиками программного обеспечения, защищающими свой интеллектуальный капитал.

SCHEMABINDING

Привязывает представление к определенной схеме, при этом объекты, используемые в представлении, должны именоваться полностью (то есть указывается как имя владельца, так и имя объекта). Представление и все объекты, используемые в представлении, должны полностью уточняться, как, например, **pubs.scott.employee.**[1] Для удаления или изменения представления (или используемых в нем таблиц), созданного с опцией *SCHEMABINDING*, необходимо предварительно удалить привязку при помощи оператора *ALTER VIEW*.

VIEW_METADATA

Указывает, что для вызовов через программные интерфейсы DBLIB и OLEDB возвращаются метаданные о представлении. Представления с опцией *VIEW_METADATA* могут обновляться триггерами *INSERT* и *INSTEAD OF UPDATE*.

WITH CHECK OPTION

Гарантирует, что через представление можно будет вставить или изменить только те данные, которые будут возвращаться оператором *SELECT* представления.

В операторе *SELECT* представления в SQL Server нельзя:

- Использовать *COMPUTE*, *COMPUTE BY*, *INTO* и *ORDER BY* (*ORDER BY* допустим только при *SELECT TOP*)
- Использовать временные таблицы
- Использовать табличные переменные
- Использовать более 1024 столбцов (считая вместе со столбцами подзапросов)

В следующем примере мы создадим представление с опциями *ENCRYPTION* и *CHECK OPTION*:

```
CREATE VIEW california_authors (last_name, first_name, author_id)
WITH ENCRYPTION
```

[1] Авторы противоречат себе. Имя базы данных указывать не следует. Более того, все фигурирующие в выражении объекты должны относиться к одной базе данных. – *Прим. науч. ред.*

```
AS
  SELECT au_lname, au_fname, au_id
  FROM authors
  WHERE state = 'CA'
WITH CHECK OPTION
GO
```

В представлении можно использовать несколько операторов *SELECT*, если они объединены при помощи *UNION* или *UNION ALL*. Также оператор *SELECT* представления может содержать вызовы функций и подсказки оптимизатору. Представление в SQL Server будет обновляемым, если выполнены все следующие условия:

- Не используются агрегатные функции
- Не используются фразы *TOP*, *GROUP BY*, *DISTINCT* и *UNION*
- Не создаются производные столбцы (смотрите *SUBQUERY*)
- Во фразе *FROM* используется хотя бы одна таблица[1]

В SQL Server можно создавать индексы по представлениям. Если вы создадите уникальный кластерный индекс по представлению, то SQL Server сохранит физическую копию данных представления. Изменения базовых таблиц автоматически вызывают обновление представления. Индексированные представления занимают дополнительное пространство, но дают огромный прирост в производительности. Такие представления должны создаваться с опцией *SCHEMABINDING*.

SQL Server позволяет создавать *локальные* и *распределенные секционированные представления*. Секционированное представление соединяет горизонтально секционированные данные из группы таблиц, находящихся на одном или нескольких серверах, представляя данные так, как будто они содержатся в единой таблице. В локальном секционированном представлении все участвующие в нем таблицы и само представление находятся на одном и том же экземпляре SQL Server. Распределенное секционированное представление может использовать данные, находящиеся на удаленных серверах.[2]

Секционированные представления явно выбирают данные из нескольких различных источников, и каждый источник объединяется с другими при помощи оператора *UNION ALL*. Оператор *SELECT* должен выбирать все столбцы из представлений, и наборы столбцов должны быть идентичными.[3] (Идея заключается в том, что через пользовательское приложение вы логически разделяете данные, а SQL Server затем распределяет их через секционированные представления.) Следующий пример показывает, как создается представление с данными из трех различных источников:

---

[1] Вероятно, авторы имели в виду следующее правило: любые изменения, включая использование инструкций UPDATE, INSERT и DELETE, должны касаться столбцов только из одной базовой таблицы. – *Прим. науч. ред.*

[2] В распределенном секционированном представлении хотя бы одна из участвующих таблиц находится на другом (удаленном) сервере. – *Прим. науч. ред.*

[3] Вероятно, авторы имели в виду, что список столбцов в базовых таблицах должен совпадать, и выбираться должны все столбцы. – *Прим. науч. ред.*

```
CREATE VIEW customers
AS
--Выбираем данные из локальной таблицы на сервере New_York
SELECT *
FROM sales_archive.dbo.customers_A
UNION ALL
SELECT *
FROM houston.sales_archive.dbo.customers_K
UNION ALL
SELECT *
FROM los_angeles.sales_archive.dbo.customers_S
```

Обратите внимание, что каждый удаленный сервер (*New_York*, *houston* и *los_angeles*) должен быть зарегистрирован на всех серверах, использующих распределенное секционированное представление.

Секционированные представления позволяют увеличить производительность системы путем распределения нагрузки на несколько серверов. Однако они сложны в проектировании, создании и использовании. Обязательно прочитайте документацию, чтобы знать все возможные детали, касающиеся секционированных представлений.

При изменении (*ALTER VIEW*) существующего представления SQL Server захватывает и удерживает эксклюзивную блокировку представления до окончания операции. *ALTER VIEW* также удаляет все индексы, связанные с представлением, поэтому их потребуется явно пересоздавать с помощью оператора *CREATE INDEX*.

См. также

CREATE/ALTER TABLE
DROP
SELECT
SUBQUERY

DECLARE CURSOR

Оператор *DECLARE* является одной из четырех (наряду с *FETCH*, *OPEN* и *CLOSE*) команд для работы с курсорами. Курсоры позволяют вместо одновременной обработки множества строк обрабатывать каждую строку отдельно. Команда *DECLARE CURSOR* определяет, какие строки извлекаются из таблицы или представления для обработки.

Другими словами, курсоры особенно важны при работе с реляционными базами данных, так как базы данных работают с множествами строк, а клиентские приложения работают с данными построчно. Курсоры, во-первых, позволяют программистам использовать методологию их любимого языка программирования. Во-вторых, так как механизм работы курсоров в некоторых СУБД кардинально отличается от обычного механизма работы с наборами строк, то курсоры в этих СУБД могут работать существенно медленнее, чем обычные пакетные операции.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Поддерживается с ограничениями
PostgreSQL	Поддерживается с ограничениями
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

```
DECLARE имя_курсора [{SENSITIVE | INSENSITIVE | ASENSITIVE}]
[[NO] SCROLL] CURSOR [{WITH | WITHOUT} HOLD]
  [{WITH | WITHOUT} RETURN]
FOR оператор_select
[FOR {READ ONLY | UPDATE [OF столбец[, ...]]}]
```

Ключевые слова

DECLARE имя_курсора

Определяет имя курсора, уникальное в контексте, в котором объявляется курсор (например, в базе данных или отдельной схеме). Не может быть создано другого курсора с таким же именем, как у уже существующего.

SENSITIVE | INSENSITIVE | ASENSITIVE

Определяет режим взаимодействия курсора с таблицами-источниками и способ формирования результирующего множества, возвращаемого курсором.

SENSITIVE

В этом режиме курсор работает напрямую с данными таблицы, и все изменения таблицы видны через курсор.

INSENSITIVE

В этом режиме для курсора создается временная копия таблицы-источника, и поэтому все изменения таблицы, выполненные другими операторами, невидимы во время работы курсора.

ASENSITIVE

Оставляет выбор режима работы курсора на усмотрение СУБД. Является режимом по умолчанию в SQL2003.

[NO] SCROLL

В режиме *NOSCROLL* требуется обработка каждой строки курсора и переход между строками осуществляется только оператором *FETCH NEXT*. *SCROLL* позволяет обрабатывать строки в произвольном порядке и для перехода между строками использовать любые формы оператора *FETCH*.

{WITH | WITHOUT} HOLD

Курсор, объявленный с параметром *WITH HOLD*, остается открытым при фиксации транзакции. При использовании *WITHOUT HOLD* курсор закрывается при выполнении *COMMIT*. (Курсор *WITH HOLD* закрывается либо при выполнении *ROLLBACK*, либо оператором *CLOSE CURSORE*.)

{WITH | WITHOUT} RETURN

Используется только в хранимых процедурах. *WITH RETURN* позволяет вернуть из хранимой процедуры открытый курсор. *WITHOUT RETURN* означает, что все открытые в хранимой процедуре курсоры автоматически закрываются при выходе из процедуры.

FOR оператор_select

Объявляет оператор *SELECT*, используемый для построения результирующего множества строк. Как и в обычном операторе *SELECT*, строки курсора можно сортировать, используя фразу *ORDER BY*.

FOR {READ ONLY | UPDATE [OF столбец[, ...]]}

Параметр *FOR READ ONLY* означает, что строки курсора не обновляются. Это является поведением по умолчанию для курсоров с опциями *SCROLL* или *INSENSITIVE*, с *ORDER BY* в операторе *SELECT* или курсоров по необновляемым таблицам. Если вы хотите обновлять курсор, то используйте параметр *FOR UPDATE OF столбец1, столбец2[,...]*, где список столбцов можно опустить и тогда обновляемыми будут все столбцы курсора.

Общие правила

Команда *DECLARE CURSOR* дает возможность извлекать из таблицы и обрабатывать строки по одной. Это позволяет реализовать построчную обработку данных вместо традиционной обработки наборов строк.

Схематично работа с курсорами представляет собой следующую последовательность:

1. Курсор объявляется командой *DECLARE*.
2. Курсор открывается командой *OPEN*.
3. Осуществляется работа с курсором с помощью команды *FETCH*.
4. Курсор закрывается командой *CLOSE*.

В команде *DECLARE CURSOR* объявляется оператор *SELECT*. Каждую строку, возвращаемую этим оператором, можно отдельно извлечь и обработать. В следующем примере для Oracle курсор объявляется вместе с некоторыми другими переменными в инициализационном блоке. Затем курсор открывается, с ним выполняются определенные манипуляции, и затем курсор закрывается:

```
DECLARE CURSOR title_price_cursor IS
  SELECT title, price
  FROM titles
  WHERE price IS NOT NULL;
title_price_val title_price_cursor%ROWTYPE;
new_price NUMBER(10,2);
BEGIN
  OPEN title_price_cursor;
  FETCH title_price_cursor INTO title_price_val;
  new_price := "title_price_val.price" * 1.25
  INSERT INTO new_title_price
  VALUES (title_price_val.title, new_price)
  CLOSE title_price_cursor;
END;
```

Так как в этом примере используется PL/SQL, то большая часть кода выходит за рамки рассматриваемого в этой книге. Тем не менее, легко видеть объявление курсора оператором *DECLARE*.[1] Затем в выполняемой части блока PL/SQL курсор инициализируется оператором *OPEN*, затем извлекаются значения оператором *FETCH*, и в конце курсор закрывается оператором *CLOSE*.

Оператор *SELECT* является сердцем вашего курсора, поэтому перед использованием его в *DECLARE CURSOR* следует внимательно его проверить. Оператор *SELECT* может обращаться к таблице или к представлению. Если курсор используется только для чтения, то он также может обращаться к необновляемым представлениям. Также оператор *SELECT* необновляемого курсора может содержать подфразы *ORDER BY*, *GROUP BY* и *HAVING*. Если же курсор объявлен с параметром *FOR UPDATE*, то желательно исключить указанные подфразы из оператора *SELECT*.

Локальные курсоры иногда используются как выходные параметры хранимых процедур. То есть хранимая процедура может объявить и инициализировать курсор, а затем вернуть его вызывающему процессу или процедуре.

В следующем примере для SQL Server объявляется и открывается курсор на базе таблицы **publishers**. Из курсора извлекается первая строка и вставляется в другую таблицу, затем следующая строка, и т. д., пока не будут обработаны все строки. Наконец, курсор закрывается и освобождается (*DEALLOCATE* используется для освобождения ресурсов курсора только в SQL Server):

```
DECLARE @publisher_name VARCHAR(20)
DECLARE pub_cursor CURSOR
FOR SELECT pub_name FROM publishers
  WHERE country <> 'USA'
OPEN pub_cursor
FETCH NEXT FROM pub_cursor INTO @publisher_name
WHILE @@FETCH_STATUS = 0
BEGIN
    INSERT INTO foreign_publishers VALUES(@publisher_name)
END
CLOSE pub_cursor
DEALLOCATE pub_cursor
```

В этом примере показано, как происходит перемещение по строкам курсора. (Этот пример только демонстрирует работу с курсорами, а для выполнения описанной задачи лучше подходит оператор *INSERT...SELECT*.)

Советы и хитрости

В большинстве платформ не поддерживаются динамические выполняемые курсоры. Наоборот, курсоры встраиваются в приложения, хранимые процедуры, пользовательские функции и т. д. Другими словами, различные операторы для создания и использования курсоров используются только в контексте хранимых процедур либо других программных объектов, но не в обычных SQL-скриптах.

---

[1] В PL/SQL ключевое слово DECLARE – это не часть объявления курсора, т. к. является ключевым словом, начинающим секцию объявления переменных, типов данных, курсоров и т. д. – *Прим. науч. ред.*

Для упрощения разработки и миграции не используйте фразу *SCROLL* и параметры *SENSITIVE*, *INSENSITIVE*, *ASENSITIVE*. В большинстве платформ поддерживаются только курсоры, по которым можно перемещаться только вперед, поэтому вы избежите многих проблем, если будете использовать только простейшие формы курсора.

Курсоры в разных СУБД по-разному ведут себя при завершении и в начале транзакций – например, когда транзакции откатываются во время работы курсора. Внимательно ознакомьтесь с особенностями поведения в таких ситуациях на каждой платформе.

В MySQL не поддерживаются серверные курсоры, описанные в ANSI SQL, но аналогичную функциональность можно получить, используя хорошую поддержку разработки расширений на C.

MySQL

MySQL поддерживает подмножество синтаксиса SQL3:

```
DECLARE имя_курсора
FOR оператор_select
```

Курсоры можно использовать только в хранимых процедурах, функциях и триггерах. Курсоры в MySQL всегда необновляемые, непрокручиваемые (перемещаться по курсору можно только на одну строку вперед, и строки нельзя пропускать) и всегда работают в режиме *ASENSITIVE* (СУБД сама выбирает, создавать ли для курсора временную копию данных или работать напрямую с таблицами-источниками). Внутри одного блока нельзя создавать несколько курсоров с одинаковыми именами. Курсоры объявляются после переменных и условий, но перед обработчиками.

Oracle

Oracle имеет весьма интересную реализацию курсоров. В реальности, любой оператор работы с данными (*INSERT*, *UPDATE*, *DELETE* и *SELECT*) неявно открывает курсор. Например, в программе на языке C для построчной обработки не требуется объявлять курсор, так как построчная обработка и так является поведением по умолчанию в Oracle. Поэтому вам потребуется объявлять курсоры с помощью *DECLARE CURSOR* только из PL/SQL блоков, таких как хранимые процедуры, но не из скриптов на чистом SQL.

Так как курсоры в Oracle используются только в хранимых процедурах и пользовательских функциях, то они описываются в документации по PL/SQL, а не по SQL.

В Oracle синтаксис *DECLARE CURSOR*[1] немного модифицирован для поддержки параметризации:

[1] В PL/SQL ключевое слово DECLARE не является частью объявления курсора, т. к. является ключевым словом, начинающим секцию объявления переменных, типов данных, курсоров и т. д. – *Прим. науч. ред.*

```
DECLARE CURSOR имя_курсора [ (параметр тип_данных[, ...]) ]
IS оператор_select
[FOR UPDATE [OF столбец[, ...]]]
```

где:

[(параметр тип_данных[, ...])]IS оператор_select

Определяет имя и тип данных каждого параметра курсора, а также оператор *SELECT*, используемый для построения результирующего множества строк. Служит для тех же целей, что и фраза *FOR оператор_select* из ANSI SQL2003.

FOR UPDATE [OF столбец[, ...]]

Указывает, что курсор (или некоторые его столбцы) является обновляемым.

В Oracle можно объявить курсорные переменные, а затем использовать их во фразе *WHERE* курсорного оператора *SELECT*. Значения этих переменных не задаются в операторе *DECLARE*, они устанавливаются при открытии курсора оператором *OPEN*. Также важно знать, что все системные функции возвращают одинаковые значения для каждой строки курсора.

PostgreSQL

В PostgreSQL не поддерживаются фразы *WITH*, зато есть возможность возвращать результирующее множество строк в двоичном, а не текстовом формате. Хотя компилятор воспринимает многие ключевые слова из синтаксиса оператора по ANSI, в реальности оператор *DECLARE CURSOR* в PostgreSQL более ограничен, чем может показаться на первый взгляд:

```
DECLARE имя_курсора [BINARY] [INSENSITIVE] [[NO] SCROLL] CURSOR [{WITH | WITHOUT} HOLD]
FOR оператор_select
[FOR {READ ONLY | UPDATE [OF столбец[, ...]]}]
```

где:

BINARY

Возвращает данные из курсора в двоичном, а не текстовом формате.

INSENSITIVE

Указывает, что обновления таблиц-источников из других процессов не влияют на данные курсора. Это поведение реализовано в PostgreSQL по умолчанию, поэтому ключевое слово является необязательным.

[NO]SCROLL

SCROLL позволяет за одну операцию *FETCH* извлекать несколько строк, причем можно двигаться в любом направлении курсора. Будьте осторожны с использованием *SCROLL*, так как это может сильно замедлить работу с курсором. *NOSCROLL* позволяет извлекать из курсора только по одной строке и не позволяет перескакивать через строки.

{WITH|WITHOUT} HOLD

WITH HOLD указывает, что курсор можно продолжать использовать после фиксации транзакции, в которой он был создан. *WITHOUT HOLD*, значение по умолчанию, указывает, что курсор нельзя использовать вне транзакции, в которой он был создан.

FOR {READ-ONLY | UPDATE [OF столбец[, ...]]}

Выбирает режим, в котором открывается курсор: только чтение (*READ-ONLY*) или обновление. Однако в PostgreSQL поддерживаются только необновляемые курсоры. *FOR UPDATE* не имеет никакого эффекта, а *FOR UPDATE OF* со списком столбцов генерирует сообщение об ошибке.

PostgreSQL автоматически закрывает курсор при открытии нового курсора с тем же именем. Двоичные курсоры обычно работают быстрее, чем обычные, так как PostgreSQL хранит данные также в двоичном формате. Но так как клиентские приложения все равно должны работать с текстовыми данными, то вам часто будет требоваться дополнительная обработка бинарных курсоров.

Курсоры в PostgreSQL можно использовать только в рамках транзакций. Поэтому курсор нужно оборачивать в блок *BEGIN* и *COMMIT/ROLLBACK*.

PostgreSQL не поддерживает отдельно команду *OPEN*. Курсоры открываются в момент их объявления. Поэтому для объявления и открытия курсора в PostgreSQL можно использовать примерно такой код:

```
DECLARE pub_cursor CURSOR
FOR SELECT pub_name FROM publishers
  WHERE country <> 'USA';
```

SQL Server

SQL Server поддерживает стандарт ANSI, а также некоторые расширения, придающие особую гибкость в возможности навигации по курсору и его обработке:

```
DECLARE имя_курсора CURSOR
  [LOCAL | GLOBAL] [INSENSITIVE | FORWARD_ONLY | SCROLL]
  [STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
  [READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
  [TYPE_WARNING]
FOR оператор_select
[FOR {READ ONLY | UPDATE [OF столбец[, ...] ]} ]
```

где:

LOCAL | GLOBAL

Определяет область действия курсора, ограничивая ее текущим блоком Transact-SQL (*LOCAL*) или всеми блоками текущей сессии. Глобальный курсор может использоваться любой процедурой, функцией или блоком T-SQL, запущенным в текущей сессии. Глобальные курсоры неявно закрываются при завершении сессии, а локальные курсоры требуется закрывать вручную. Ключевые слова *LOCAL* и *GLOBAL* являются необязательными, а значение по умолчанию определяется параметром базы данных `default_to_local`.

INSENSITIVE | FORWARD_ONLY | SCROLL

Определяет способ перемещения курсора по результирующему набору строк:

INSENSITIVE

Результирующий набор создается в базе **TEMPDB**. Изменения в базовых таблицах не отражаются на результате курсора. Эта опция несовместима с такими расширениями, как *LOCAL*, *GLOBAL*, *STATIC*, *KEYSET*, *DYNAMIC*, *FAST_FORWARD* и т. п. Может использоваться только в операто-

ре *DECLARE CURSOR* в стиле стандарта SQL-92, то есть *DECLARE* `курсор` *CURSOR INSENSITIVE FOR* `оператор_select` *FOR UPDATE*.

FORWARD_ONLY

Указывает, что курсор проходит все строки результирующего набора в одном направлении, при этом для перехода можно использовать только *FETCH NEXT*. Предполагается, что курсор работает в режиме *DYNAMIC*, если только явно не указаны опции *STATIC* или *KEYSET*. *FAST_FORWARD* и *FORWARD_ONLY* взаимоисключаемы.

SCROLL

Позволяет использовать все формы оператора *FETCH* (*ABSOLUTE*, *FIRST*, *LAST*, *NEXT*, *PRIOR* и *RELATIVE*). В противном случае допускается только *FETCH NEXT*. Если используются ключевые слова *DYNAMIC*, *STATIC* или *KEYSET*, то режим *SCROLL* подразумевается по умолчанию. *FAST_FORWARD* и *SCROLL* взаимоисключаемы.

STATIC | KEYSET | DYNAMIC | FAST_FORWARD

Определяет способ работы со строками курсора. Эти опции несовместимы с фразами *FOR READ ONLY* и *FOR UPDATE*.

STATIC

Создается временная копия результирующего набора и сохраняется в базе **tempdb**. Модификации исходных таблиц не отражаются на результате курсора. Курсоры *STATIC* не позволяют изменять данные в базовых таблицах.

KEYSET

Создается временная копия результирующего набора, а также набор ключей, который связывает результат курсора с данными исходной таблицы или представления. Это позволяет видеть через курсор изменения в исходных данных. Обновленные или удаленные строки имеют в *@@FETCH_STATUS* значение -2 (если только обновление не сделано с помощью *UPDATE... WHERE CURRENT OF*, в этом случае обновление полностью видно через курсор), а строки, вставленные другими пользователями, не видны совсем.

DYNAMIC

Актуализирует набор строк в курсоре при каждом выполнении *FETCH*, что позволяет видеть через курсор все изменения базовых таблиц, даже выполненные другими пользователями. Так как результирующий набор строк может постоянно изменяться, то курсоры типа *DYNAMIC* не поддерживают *FETCH ABSOLUTE*.

FAST_FORWARD

Создает курсор в режиме *FORWARD_ONLY*, *READ_ONLY* для быстрого однократного чтения набора данных.

READ_ONLY | SCROLL_LOCKS | OPTIMISTIC

Определяет параметры поведения курсора при параллельных позиционных обновлениях. Эти параметры несовместимы с *FOR READ ONLY* и *FOR UPDATE*. Применяются следующие параметры:

READ_ONLY

Запрещает обновление курсора, а также его использование в операторах *UPDATE* и *DELETE*, содержащих *WHERE CURRENT OF*.

SCROLL_LOCKS

Гарантирует успешные позиционные обновления и удаления при помощи блокировок, накладываемых на данные по мере чтения курсора. Блокировки удерживаются до момента закрытия курсора и освобождения ресурсов. *SCROLL_LOCKS* и *FAST_FORWARD* взаимоисключаемы.

OPTIMISTIC

Допускает позиционные обновления и удаления строк, если они не изменялись с момента прочтения их курсором. Для этого SQL Server выполняет проверки времени последнего обновления или контрольных сумм, при этом данные не блокируются. *OPTIMISTIC* и *FAST_FORWARD* взаимоисключаемы.

TYPE_WARNING

Предупреждает пользователя в случае неявного преобразования типа курсора (например, из *SCROLL* в *FORWARD_ONLY*).

FOR {READ ONLY | UPDATE [OF столбец[, ...]]}

FOR READ ONLY является стандартным синтаксисом определения курсоров, используемых только для чтения. Эта фраза несовместима с другими параметрами курсоров, только что обсуждавшимися, и может быть использована только с *INSENSITIVE*, *FORWARD_ONLY* и *SCROLL*. *FOR UPDATE* позволяет обновлять столбцы курсора, используя операторы *UPDATE* и *DELETE* с фразой *WHERE CURRENT OF*. Если *FOR UPDATE* используется без списка столбцов, то можно обновлять любой столбец курсора. В противном случае можно обновлять только перечисленные столбцы.

В SQL Server существует два варианта синтаксиса оператора *DECLARE CURSOR*. Эти варианты не совместимы друг с другом! Базовый вариант синтаксиса совместим с SQL-92 и расширениями Transact-SQL. Нельзя смешивать ключевые слова из этих двух вариантов.

Синтаксис оператора *DECLARE CURSOR*, совместимый с SQL92, выглядит следующим образом:

```
DECLARE имя_курсора [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR оператор_select
[ FOR { READ ONLY | UPDATE [ OF столбец[, ...] ] } ]
```

А синтаксис объявления курсора в Transact-SQL следующий:

```
DECLARE имя_курсора CURSOR
[ LOCAL | GLOBAL ] [ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR оператор_select
[ FOR UPDATE [ OF столбец [, ...] ] ]
```

Синтаксис SQL92 позволяет улучшить переносимость кода. Синтаксис Transact-SQL позволяет создавать курсоры тех же типов, что доступны через программные интерфейсы ODBC, OLEDB и ADO, но они могут использоваться только в хранимых процедурах, пользовательских функциях, триггерах и нерегламентированных запросах.

Если вы объявляете курсор в Transact-SQL, но не определяете параметры поведения при параллельных обновлениях, используя *OPTIMISTIC*, *READ_ONLY* или *SCROLL_LOCKS*, то:

- Если курсор объявлен с параметрами *FAST_FORWARD* или *STATIC*, или у пользователя нет привилегий на обновление базовых таблиц, то курсор будет иметь тип *READ_ONLY*.
- Если курсор объявлен с параметрами *DYNAMIC* или *KEYSET*, то по умолчанию он будет иметь тип *OPTIMISTIC*.

Обратите внимание, что в курсорном операторе *SELECT* можно использовать переменные, но они вычисляются в момент объявления курсора. Поэтому, если в курсоре есть столбец, основанный на системной функции *GETDATE( )*, то каждая строка курсора будет иметь в этом столбце одинаковое значение.

В следующем примере мы используем курсор типа *KEYSET* для замены пробелов на дефисы в столбце **phone** таблицы **authors**:

```
SET NOCOUNT ON
DECLARE author_name_cursor CURSOR LOCAL KEYSET TYPE_WARNING
  FOR SELECT au_fname FROM pubs.dbo.authors
DECLARE @name varchar(40)
OPEN author_name_cursor
FETCH NEXT FROM author_name_cursor INTO @name
WHILE (@@fetch_status <> -1)
BEGIN
-- используем @@fetch_status для обнаружения удаленных
-- и ошибочных строк
  IF (@@fetch_status <> -2)
  BEGIN
    PRINT 'обновляем строку для ' + @name
    UPDATE pubs.dbo.authors
    SET phone = replace(phone, ' ', '-')
    WHERE CURRENT OF author_name_cursor
  END
  FETCH NEXT FROM author_name_cursor INTO @name
END
CLOSE author_name_cursor
DEALLOCATE author_name_cursor
GO
```

См. также

CLOSE CURSOR

FETCH

OPEN

DELETE

Оператор *DELETE* удаляет строки из одной или нескольких таблиц. Операторы *DELETE*, применяемые к таблицам, называются иногда *поисковыми удалениями* (search deletes). Оператор *DELETE* также можно использовать совместно с курсорами, тогда он называется *позиционным удалением*.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

```
DELETE FROM { имя_таблицы | ONLY (имя_таблицы) }
[{ WHERE условие_поиска | WHERE CURRENT OF имя_курсора }]
```

Ключевые слова

FROM {имя_таблицы | ONLY (имя_таблицы) }

Указывает таблицу, из которой удаляются строки. Если явно не указано имя схемы, то подразумевается, что таблица находится в текущей схеме. Можно указать также имя представления. Ключевое слово *FROM* обязательно, если только не используется *DELETE...WHERE CURRENT OF*. Если вы не используете ключевое слово *ONLY*, то имя таблицы не заключается в скобки. *ONLY* используется только для объектных таблиц и представлений и ограничивает каскадное удаление записей в таблицах-потомках целевой таблицы. Если использовать слово *ONLY* для обычных таблиц, то оно будет проигнорировано и не вызовет ошибки.

WHERE условие_поиска

Определяет условие поиска удаляемых записей. В *WHERE* допускаются любые корректные выражения. Обычно условие поиска оценивается для каждой записи таблицы перед началом удаления.

WHERE CURRENT OF имя_курсора

Удаляет текущую запись указанного открытого курсора.

Общие правила

Оператор *DELETE* удаляет строки из таблицы или представления. Освобожденное при удалении строк пространство возвращается в базу данных, но не всегда это происходит сразу.

В простейшем виде оператор *DELETE* не содержит фразы *WHERE* и удаляет все записи из таблицы:

```
DELETE FROM sales;
```

Вы можете использовать фразу *WHERE* для определения тех записей, которые следует удалить. В следующем примере все три оператора *DELETE* являются до-

пустимыми, и во всех используется фраза *FROM*, так как они являются поисковыми удалениями:

```
DELETE FROM sales
WHERE qty IS NULL;
DELETE FROM suppliers
WHERE supplierid = 17
  OR companyname = 'Tokyo Traders';
DELETE FROM distributors
WHERE postalcode IN
  (SELECT territorydescription FROM territories);
```

Помните, что в позиционных удалениях фраза *FROM* не используется.

В некоторых случаях вам может требоваться удалять записи из открытого курсора:

```
DELETE titles WHERE CURRENT OF title_cursor;
```

В этом операторе подразумевается, что вы объявили и открыли курсор с названием **title_cursor**. При выполнении оператора *DELETE* удаляется та запись курсора, на которой он находится в этот момент.

Советы и хитрости

Оператор *DELETE* достаточно редко выполняется без фразы *WHERE*, так как в этом случае удаляются все строки таблицы. Перед удалением выполните оператор *SELECT* с тем же фильтром в *WHERE*, это позволит вам убедиться, что вы удаляете те записи, которые нужно.

Если вам требуется удалить все записи из таблицы, то используйте не описанный в ANSI, но весьма распространенный оператор *TRUNCATE TABLE*. В тех базах данных, в которых он поддерживается, оператор *TRUNCATE TABLE* обычно работает быстрее, чем *DELETE*, так как при его выполнении не журналируется удаление каждой записи. При удалении большого числа записей отказ от журналирования значительно экономит время, но при этом делает невозможным откат удаления. Более того, в некоторых базах данных перед выполнением *TRUNCATE TABLE* требуется удалить все внешние ключи.

MySQL

В MySQL есть несколько расширений стандарта ANSI, но не поддерживается фраза *WHERE CURRENT OF*. Синтаксис оператора следующий:

```
DELETE [LOW_PRIORITY] [QUICK] [имя_таблицы[.*][,...]]
{FROM имя_таблицы [.*][,...]|[USING имя_таблицы[.*][,...]]}
[WHERE условие_поиска]
[ORDER BY фраза]
[LIMIT число_строк]
```

Параметры имеют следующие значения:

LOW PRIORITY

Откладывает удаление до того момента, когда другие клиенты закончат чтение таблицы.

QUICK

Предотвращает слияние листьев индекса в процессе удаления.

DELETE имя_таблицы[,...]

Позволяет за один раз удалить строки из нескольких таблиц. Таблицы, перечисленные перед фразой *FROM*, при наличии списка таблиц после *FROM* являются целевыми таблицами: из всех этих таблиц будут удалены все требуемые записи.

FROM имя_таблицы [.]*

Указывает одну или несколько таблиц, из которых удаляются записи. (.* используется для улучшенной совместимости с MS Access.) Если перед *FROM* указан список таблиц, то таблицы после *FROM* используются в операциях соединения и поиска по справочникам.

USING имя_таблицы [.][,...]*

Заменяет одну или несколько таблиц, указанных перед фразой *FROM*, таблицами после фразы *FROM*.

ORDER BY фраза

Определяет порядок, в котором будут удаляться строки. Имеет смысл только при использовании вместе с фразой *LIMIT*.

LIMIT число_строк

LIMIT позволяет ограничить число строк, после удаления которых управление возвращается пользователю.

MySQL позволяет одной командой удалить данные из нескольких таблиц. В следующем примере два оператора *DELETE* функционально эквивалентны:

```
DELETE orders FROM customers, orders
WHERE customers.customerid = orders.customerid
  AND orders.orderdate BETWEEN '19950101' AND '19951231'
DELETE FROM orders USING customers, orders
WHERE customers.customerid = orders.customerid
  AND orders.orderdate BETWEEN '19950101' AND '19951231'
```

В этих примерах мы удаляем все заказы из таблицы **orders**, сделанные клиентами из таблицы **customers** в течение 1995 года. Обратите внимание, что в мультитабличных удалениях нельзя использовать фразы *LIMIT* и *ORDER BY*.

Используя *LIMIT* и *ORDER BY* вы можете удалять записи упорядоченными наборами:

```
DELETE FROM sales
WHERE customerid = 'TORTU'
ORDER BY customerid
LIMIT 5
```

MySQL в некоторых случаях позволяет ускорить операции удаления. Например, обычно после завершения удаления возвращается число удаленных записей. Но если удаляются все записи таблицы, то MySQL не будет считать удаленные записи и вернет 0, так как это быстрее. В режиме *AUTOCOMMIT* оператор *DELETE* без фразы *WHERE* будет заменен на более быструю операцию *TRUNCATE*.

Помните, что скорость удаления строк из таблицы напрямую зависит от количества индексов, построенных по этой таблице и от размера индексного кэша. Удаление будет выполняться быстрее, если данные удаляются из таблицы с небольшим числом индексов или вообще без индексов, либо если доступен большой индексный кэш.

Oracle

Oracle позволяет удалять строки из таблиц, представлений, материализованных представлений, вложенных подзапросов и секционированных таблиц и представлений:

```
DELETE [FROM]
  {имя_таблицы | ONLY (имя_таблицы)} [псевдоним]
    [{PARTITION (имя_секции) |
    SUBPARTITION (имя_подсекции)}] |
  (подзапрос [WITH {READ ONLY |
    CHECK OPTION [CONSTRAINT имя_ограничения]}]) |
  TABLE (выражение_коллекция) [ (+) ]}
[подсказка]
[WHERE условие_поиска]
[RETURNING выражение[, ...] INTO переменная[, ...]]
[LOG ERRORS [INTO [схема.]имя_таблицы] [(простое_выражение)]
  [REJECT LIMIT {UNLIMITED |целое_число }]]
```

Параметры имеют следующие значения:

имя_таблицы [псевдоним]

Указывает таблицу, материализованное представление, секционированную таблицу или представление, из которого удаляются строки. Вы можете при необходимости указать имя схемы или связь с другой базой данных. По умолчанию Oracle будет использовать таблицу в текущей схеме локального сервера. Также можно указать псевдоним для таблицы. Псевдоним необходим, если целевая таблица использует атрибут или метод объектного типа.

PARTITION имя_секции

Применяет операцию удаления только к указанной секции. Использование *PARTITION* не является обязательным при удалении из секционированной таблицы, но позволяет уменьшить сложность фразы *WHERE*.

SUBPARTITION имя_секции

Применяет операцию удаления только к указанной подсекции вместо всей таблицы.

(подзапрос [WITH {READ ONLY|CHECK OPTION [CONSTRAINT имя_ограничения]}])

Указывает, что строки удаляются из подзапроса, а не из таблицы или представления. Для этой фразы используются следующие параметры:

подзапрос

Определяет *SELECT*-оператор подзапроса. В качестве подзапроса можно использовать любой корректный подзапрос, но без фразы *ORDER BY*.

WITH READ ONLY

Указывает, что подзапрос не может обновляться.

WITH CHECK OPTION [CONSTRAINT имя_ограничения]

Гарантирует, что Oracle не выполнит удаление тех записей, которые не попадают в результирующее множество подзапроса. Фразой [*CONSTRAINT имя_ограничения*] можно ограничить допустимые изменения с помощью указанного ограничения.

TABLE (выражение_коллекция) [(+)]

Позволяет работать с указанной коллекцией как с таблицей, хотя коллекция может задаваться подзапросом, функцией или другим конструктором коллекций. В любом случае, указанное выражение должно быть либо вложенной таблицей, либо иметь тип *VARRAY*.

подсказка

Указывает оптимизатору необходимый порядок выполнения запроса, отличный от того, который мог бы быть выбран по умолчанию (например, можно игнорировать при выполнении определенный индекс). Информацию о подсказках оптимизатору ищите в документации.

RETURNING выражение

Возвращает строки, обработанные командой. (По умолчанию *DELETE* возвращает только число удаленных записей.) Фразу *RETURNING* можно использовать при удалении из таблицы, материализованного представления или представления с одной базовой таблицей. При удалении единичной записи *RETURNING* возвращает значения из удаленной записи (определяемые выражением) в переменные PL/SQL или связанные переменные. При удалении нескольких записей их значения сохраняются в связанных массивах.

INTO переменная

Определяет переменные, в которые сохраняются значения удаленных записей, определенные в *RETURNING*. Для каждого выражения из *RETURNING* в *INTO* должна быть определена соответствующая переменная.

LOG ERRORS [INTO [схема.]имя_таблицы] [(простое_выражение)] [REJECT LIMIT {UNLIMITED|целое_число}]

Сохраняет в журнальной таблице информацию об ошибках DML-операции и значения соответствующих строк. Нарушения ограничений целостности всегда вызывают остановку и откат операции, вне зависимости от того, используется или нет фраза *LOG ERRORS*. При помощи *LOG ERRORS* нельзя журналировать ошибки в столбцах типов *LONG* и *LOB*, а также в столбцах объектных типов, хотя сами таблицы могут содержать такие столбцы. Параметры этой фразы следующие:

INTO [схема.]имя_таблицы

Указывает имя журнальной таблицы. По умолчанию используется значение **ERR$_xxx**, где xxx соответствует первым 25 символам из названия таблицы, из которой удаляются записи.

(простое_выражение)

Указывает выражение, которым помечаются записи в таблице ошибок. Это позволяет различать ошибки, вызванные разными DML-операциями.

REJECT LIMIT {UNLIMITED |целое_число}

Указывает предельное значение числа ошибок, при достижении которого операция DELETE прерывается и транзакция откатывается. По умолчанию используется значение 0. Ключевое слово *UNLIMITED* используется при неограниченном числе ошибок.

При выполнении *DELETE* Oracle освобождает место и возвращает его таблице или индексу, в котором хранились данные.

Для удаления из представления требуется, чтобы представление не содержало операторов работы с множествами, ключевого слова *DISTINCT*, соединений, агрегатных функций, аналитических функций, подзапросов или коллекций в списке *SELECT*, фраз *GROUP BY*, *ORDER BY*, *CONNECT BY* и *START WITH*.

Вот пример удаления записей из таблицы удаленного сервера:

```
DELETE FROM scott.sales@chicago;
```

Следующий оператор удаляет данные из производной таблицы, то есть из коллекции:

```
DELETE TABLE(SELECT contactname FROM customers c
  WHERE c.customerid = 'BOTTM') s
WHERE s.region IS NULL OR s.country = 'MEXICO';
```

Вот пример удаления данных из определенной секции:

```
DELETE FROM sales PARTITION (sales_q3_1997)
WHERE qty > 10000;
```

В последнем примере мы возвращаем удаленные значения в переменные с помощью фразы *RETURNING*:

```
DELETE FROM employee
WHERE job_id = 13
AND hire_date + TO_YMINTERVAL('01-06') =< SYSDATE;
RETURNING job_lvl
INTO :int01;
```

В этом примере удаляются записи из таблицы **employee**, и значения **job_lvl** сохраняются в заранее объявленной переменной **int01**.

PostgreSQL

В PostgreSQL команда *DELETE* используется для удаления строк и любых подклассов из таблицы. В остальном реализация соответствует стандарту ANSI. Синтаксис следующий:

```
DELETE [FROM] [ONLY] [схема.]имя_таблицы
[ USING список_использования ]
[ WHERE условие_поиска | WHERE CURRENT OF имя_курсора ]
[ RETURNING { * | выражение [AS псевдоним][, ...] } ]
```

Для удаления записей только из указанной таблицы используйте фразу *ONLY*. В противном случае PostgreSQL удалит записи также из любых подтаблиц. В PostgreSQL также поддерживаются две другие важные фразы:

USING список_использования

Определяет список табличных выражений, что позволяет использовать столбцы этих таблиц в *WHERE*. Эта функциональность аналогична использованию нескольких таблиц во фразе *FROM* оператора *SELECT*.

RETURNING { | выражение [AS псевдоним][, ...] }*

Определяет выражение, возвращаемое оператором для каждой удаленной строки. Можно использовать либо символ * для возврата всех столбцов, либо указать произвольное выражение, использующее столбцы таблиц из *FROM* и *USING*.

Следующий оператор удаляет все записи из таблицы **titiles**:

```
DELETE titles
```

Для удаления из таблицы **authors** всех строк, начинающихся на 'Mc', можно использовать следующий оператор:

```
DELETE FROM authors
WHERE au_lname LIKE 'Mc%'
```

Для удаления из **titles** всех строк со старыми значениями **title_id** можно выполнить следующее:

```
DELETE titles WHERE title_id >= 40
```

Для удаления записей из **titles** с отсутствующими продажами:

```
DELETE titles WHERE ytd_sales IS NULL
```

Можно удалить строки из одной таблицы на основании результатов запроса к другой таблице (в примере мы удаляем записи из таблицы **titleauthors**, для которых в таблице **titles** есть записи со словом «computers»):

```
DELETE FROM titleauthor
WHERE title_id IN
  (SELECT title_id
  FROM titles
  WHERE title LIKE '%computers%')
DISCONNECT
```

Можно также при удалении вернуть полную информацию обо всех удаленных строках:

```
DELETE FROM titles WHERE ytd_sales IS NULL RETURNING *;
```

SQL Server

В SQL Server можно удалять строки как из таблиц, так и из представлений, построенных по одной таблице. В SQL Server можно использовать вторую фразу *FROM* для соединений. Синтаксис оператора следующий:

```
[WITH табличное_выражение[, ...]]
DELETE [TOP ( число ) [PERCENT]]
[FROM] имя_таблицы [[AS] псевдоним]
```

```
[WITH ( подсказка [...] )]
[OUTPUT выражение INTO {@табличная_переменная | таблица} [ (список_столбцов[, ...]) ]]
[FROM таблица_источник[, ...]]
[ [{INNER | CROSS | [LEFT | RIGHT | FULL] OUTER}]
  JOIN соединяемая_таблицы ON выражение][, ...] ]
[WHERE условие_поиска | WHERE CURRENT OF [GLOBAL]
  имя_курсора]
[OPTION ( подсказка[, ...N] )]
```

Элементы синтаксиса имеют следующие значения:

WITH *табличное_выражение*

Определяет именованный временный набор строк, порождаемый оператором *SELECT*.

DELETE *имя_таблицы*

Указывает таблицу или представление, из которого удаляются строки. Вы можете удалить строки из представления, если оно построено на базе одной таблицы, в нем не используются агрегатные функции и производные столбцы. Если для таблицы или представления вы не указываете имя сервера, базы данных или схемы, то SQL Server будет использовать текущий контекст. Вместо таблицы или представления вы можете использовать функции *OPENDATASOURCE* и *OPENQUERY*, описанные в разделе, посвященном оператору *SELECT*.

TOP (число) [PERCENT]

Определяет либо число строк, которое должен удалить оператор, либо процент (*PERCENT*) от общего числа строк таблицы. Если вместо константного значения используется выражение или переменная, то необходимо заключать выражение в круглые скобки. Если указывается процент строк, то выражение должно иметь тип *FLOAT* и значение в диапазоне от 0 до 100. Если указывается точное число строк, то выражение должно иметь тип *BIGINT*.

WITH (подсказка)

Указывает оптимизатору необходимый порядок выполнения запроса, отличный от того, который мог бы быть выбран по умолчанию (например, можно игнорировать при выполнении определенный индекс). *WITH* позволяет указать одну или несколько подсказок, применимых к целевой таблице. Информацию о подсказках оптимизатору ищите в документации.

OUTPUT выражение INTO {@табличная_переменная | таблица}[(список_столбцов[,...])]

Возвращает значения (задаваемые выражением) удаленных строк в указанную табличную переменную или таблицу (в обычном режиме *DELETE* вернет только число удаленных строк). Если не указан список_столбцов, то столбцы, указанные в выражении *OUTPUT*, должны соответствовать столбцам табличной переменной или таблицы, в которую сохраняются удаленные строки. (Также эта таблица не может иметь триггеры, ограничения *CHECK* и внешние ключи.)

FROM *таблица_источник*

Вторая фраза *FROM* используется для соединений таблицы из первой фразы *FROM* с другими таблицами, что позволяет обходиться без коррелированных подзапросов. В этой фразе *FROM* можно указать несколько таблиц.

[[{INNER | CROSS | [LEFT | RIGHT | FULL] OUTER}] JOIN соединяемая_таблицы ON выражение][, ...]]

Используется для соединений таблиц во второй фразе *FROM*. Вы можете использовать любой тип соединений, поддерживаемый в SQL Server. Соединения объясняются в разделе, посвященном оператору *SELECT*, позднее в этой главе.

GLOBAL имя_курсора

Указывает, что нужно удалить текущую запись открытого глобального курсора. Эта фраза является аналогом определенной стандартом фразы *WHERE CURRENT OF*.

OPTION (подсказка[, ...])

Позволяет изменить некоторые шаги плана выполнения запроса. Так как оптимизатор обычно выбирает наилучший план выполнения запроса, мы не рекомендуем использовать подсказки оптимизатору в запросах.

Существенным расширением в SQL Server стандартного оператора *DELETE* является вторая фраза *FROM*. Она позволяет использовать соединения таблиц для удаления строк из одной таблицы на основании значений связанных строк из других таблиц. Например, вы могли бы использовать достаточно сложный подзапрос для удаления записей о продажах из таблицы **sales** для книг о компьютерах:

```
DELETE FROM sales
WHERE title_id IN
  (SELECT title_id
  FROM titles
  WHERE type = 'computer')
```

В SQL Server ту же операцию можно выполнить намного элегантнее[1], используя вторую фразу *FROM*:

```
DELETE FROM sales
FROM sales AS s
INNER JOIN titles AS t ON s.title_id = t.title_id
  AND type = 'computer'
```

В следующем примере мы удаляем пачками по 2500 строк все строки с датой в поле **order_date** раньше 2003 года:

```
WHILE 1 = 1
BEGIN
  DELETE TOP (2500)
  FROM sales_history WHERE order_date <= '20030101'
  IF @@rowcount < 2500 BREAK
END
```

TOP следует применять вместо использовавшейся ранее команды *SET ROWCOUNT*, так как теперь *TOP* может быть использована большим числом алгоритмов оптимизации запроса. (До версии SQL Server 2005 оператор *SET ROW-*

[1] Элегантность этого решения сомнительна, т. к. работа оператора DELETE с JOIN будет значительно менее эффективна. – *Прим. науч. ред.*

COUNT часто использовался для пакетной обработки большого числа строк небольшими пачками, что позволяло избежать переполнения журнала транзакций и эскалации блокировок строк в табличную блокировку.)

В операторах *SELECT*, *INSERT*, *UPDATE*, *DELETE* и *CREATE VIEW* можно использовать табличные выражения. Эти выражения используются для объявления временных результирующих наборов строк, задаваемых оператором *SELECT*. При объявлении табличных выражений нельзя использовать фразы *COMPUTE*, *COMPUTE BY*, *FOR XML*, *FOR BROWSE*, *INTO*, *OPTION* и *ORDER BY*. В табличном выражении можно использовать несколько операторов *SELECT*, если они объединены при помощи операторов *UNION*, *UNION ALL*, *EXCEPT* или *INTERSECT*. Вот пример простого оператора *DELETE*, использующего табличное выражение:

```
WITH direct_reports (Manager_ID, DirectReports) AS
( SELECT manager_ID, COUNT(*)
  FROM hr.employee AS e
  WHERE manager_id IS NOT NULL
  GROUP BY manager_id )
DELETE FROM direct_reports
WHERE DirectReports <= 1;
```

Фраза *OUTPUT* позволяет увидеть все удаленные строки:

```
DELETE TOP 10 error_log WITH (READPAST)
OUTPUT deleted.*
WHERE error_log_id = '28-OCT-2008';
```

См. также

INSERT
SELECT
TRUNCATE TABLE
UPDATE

DISCONNECT

Оператор *DISCONNECT* используется для разрыва одного или нескольких соединений между текущим SQL-процессом и сервером базы данных.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с ограничениями
PostgreSQL	Не поддерживается
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

```
DISCONNECT {CURRENT | ALL | имя_подключения | DEFAULT}
```

Ключевые слова

CURRENT

Используется для закрытия активного соединения.

ALL

Закрывает все активные соединения текущего пользователя.

Общие правила

DISCONNECT может быть использована для закрытия именованной SQL-сессии (имя_подключения), текущего соединения (*CURRENT*), соединения, используемого по умолчанию (*DEFAULT*), или всех соединений пользователя (*ALL*). Например, для закрытия соединения с названием *new_york* можно выполнить следующую команду:

```
DISCONNECT new_york
```

А для закрытия всех сессий, открытых для текущего пользовательского процесса, используйте:

```
DISCONNECT ALL
```

Советы и хитрости

DISCONNECT не является универсальным оператором, поддерживаемым всеми платформами. Не используйте *DISCONNECT* в кросс-платформенных приложениях. Вместо этого используйте способы закрытия соединений, являющиеся предпочтительными для каждой конкретной платформы.

MySQL

Не поддерживается.

Oracle

В Oracle оператор *DISCONNECT* поддерживается только в SQL*Plus, инструменте выполнения запросов. Используется следующий синтаксис:

```
DISC[ONNECT]
```

Этот оператор закрывает текущую сессию с сервером базы данных, но при этом можно продолжать работу в SQL*Plus. Например, программист может продолжить редактировать буфер, сохранять файлы и т. д., но для выполнения любых команд SQL потребуется установить новое подключение. Для выхода из SQL*Plus используются команды *EXIT* или *QUIT*. Для закрытия текущего подключения к Oracle выполните:

```
DISCONNECT;
```

Аналогичный результат вы можете получить используя оператор *ALTER SYSTEM DISCONNECT SESSION*.

PostgreSQL

Не поддерживается. Вместо этого в каждом программном интерфейсе имеются отдельные команды для закрытия соединений. В Server Programming Interface это *SPI_FINISH*, а в PL/TCL это *PG_DISCONNECT*.

SQL Server

SQL Server поддерживает стандартный оператор *DISCONNECT* только во встраиваемом SQL (Embedded SQL, ESQL), но не в инструменте выполнения запросов SQL Server Management Studio. Для «аккуратного» отключения от SQL Server из ESQL-приложения используйте оператор *DISCONNECT ALL*.

См. также

CONNECT

DROP

Любые объекты базы данных, создаваемые операторами *CREATE*, могут быть удалены соответствующими операторами *DROP*. В некоторых платформах *ROLLBACK* позволяет восстановить удаленный объект. В других платформах оператор *DROP* не подлежит отмене, так что его стоит использовать с осторожностью.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с ограничениями
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

На текущий момент стандарт SQL2003 поддерживает возможность удаления большого числа разных типов объектов, многие из которых не поддерживаются производителями. Синтаксис оператора согласно ANSI следующий:

```
DROP [тип_объекта] имя_объекта {RESTRICT | CASCADE}
```

Ключевые слова

DROP [тип_объекта] имя_объекта

Безвозвратно удаляет объект указанного типа с указанным именем. Если не указана схема, в которой находится объект, то подразумевается текущая схема. ANSI SQL2003 поддерживает длинный список типов объектов, каждый из которых создается соответствующим оператором *CREATE*. Операторы *CREATE*, рассматриваемые в этой книге и имеющие парный оператор *DROP*, используются с фразами:

- *DOMAIN*
- *FUNCTION*
- *METHOD*
- *PROCEDURE*
- *ROLE*
- *SCHEMA*
- *TABLE*

- *TRIGGER*
- *TYPE*
- *VIEW*

RESTRICT | CASCADE

Позволяет предотвратить выполнение оператора *DROP* при наличии зависимых объектов (*RESTRICT*), либо позволяет одновременно удалить как сам объект, так и все зависимые объекты (*CASCADE*). Эта опция недопустима для некоторых типов объектов, например для *DROP TRIGGER*, а в некоторых случаях она обязательна, например для *DROP SCHEMA*. В частности, *DROP SCHEMA RESTRICT* удалит только пустую схему, а если в схеме есть объекты, то оператор будет прерван. А *DROP SCHEMA CASCADE* удалит и схему, и все содержащиеся в ней объекты.

Общие правила

Правила создания и модификации объектов различных типов приводятся в разделах, посвященных соответствующим операторам *CREATE/ALTER*.

Оператор *DROP* удаляет существующий объект. Объект удаляется безвозвратно, и все пользователи, имевшие ранее доступ к этому объекту, мгновенно теряют возможность использовать объект.

Вы можете указывать полное имя объекта вместе со схемой, в которой находится объект, например:

```
DROP TABLE scott.sales_2008 CASCADE;
```

Этот оператор удалит не только таблицу **scott.sales_2008**, но и все представления, триггеры и ограничения, созданные на базе этой таблицы. С другой стороны, можно не указывать имя схемы, если подразумевается текущая схема. Например:

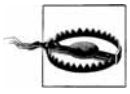
```
DROP TRIGGER before_ins_emp;
DROP ROLE sales_mgr;
```

Хотя это и не оговорено стандартом, в большинстве реализаций оператор *DROP* завершается с ошибкой, если удаляемый объект используется другим пользователем.

Советы и хитрости

Оператор *DROP* отработает корректно, только если выполнен для существующего объекта правильного типа пользователем, имеющим соответствующие привилегии (обычно требуется привилегия *DROP TABLE*, подробности приводятся в описании оператора *GRANT*). Стандарт SQL требует, чтобы объект мог удалить только создатель этого объекта, но в большинстве платформ есть некоторые отступления от этого правила. Например, суперпользователь или администратор базы данных обычно может удалить любой объект сервера баз данных.

В некоторых платформах оператор *DROP* не удалит объекты, имеющие какие-либо расширенные свойства. Например, в SQL Server нельзя удалить реплицируемую таблицу, пока она не будет исключена из репликации.



Имейте в виду, что в большинстве платформ вы не увидите предупреждения или сообщения об ошибке, если оператор *DROP* порождает проблемы с зависимыми объектами. Например, если вы попытаетесь удалить таблицу, используемую в нескольких представлениях и хранимых процедурах, то таблица удалится без проблем, но зато вы получите ошибку при попытке использования зависящих от таблицы представлений или процедур. Для предотвращения таких проблем вы можете либо воспользоваться опцией *RESTRICT*, либо проверить зависимости до выполнения оператора *DROP*.

Вы, возможно, обратили внимание, что стандарт ANSI не поддерживает некоторые операторы *DROP*, например *DROP DATABASE* и *DROP INDEX*, хотя они поддерживаются в каждой платформе, рассматриваемой в этой книге (и почти всеми платформами, представленными на рынке). Точный синтаксис таких команд рассматривается в разделах, посвященных каждой платформе.

MySQL

MySQL поддерживает сравнительно небольшое число типов объектов, соответственно синтаксис оператора *DROP* тоже ограничен:

```
DROP { {DATABASE | SCHEMA} | FUNCTION |
   INDEX [ONLINE | OFFLINE] | PROCEDURE |
   [TEMPORARY] TABLE | TRIGGER | VIEW }
[IF EXISTS] имя_объекта[, ...]
[RESTRICT |CASCADE]
```

Элементы синтаксиса имеют следующие значения:

{DATABASE | SCHEMA} *имя_базы_данных*

Удаляет указанную базу данных, а также все объекты внутри базы данных. В MySQL операторы *DROP SCHEMA* и *DROP DATABASE* являются синонимами. *DROP DATABASE* удаляет все файлы и каталоги, используемые базой данных. MySQL возвращает сообщение о числе удаленных файлов из каталога базы данных. (Удаляются файлы со следующими расширениями: *.BAK*, *.DAT*, *.FRM*, *.HSH*, *.ISD*, *.ISM*, *.MRG*, *.MYD*, *.MYI*, *.DM* и *.FM*) Вы можете удалить одним оператором только одну базу данных. Опции *RESTRICT* и *CASCADE* в операторе *DROP DATABASE* не используются.

FUNCTION *имя_функции*

В версиях MySQL 5.1 и старше используется для удаления пользовательских функций. Вы также можете использовать *IF EXISTS*.

INDEX [ONLINE | OFFLINE] *имя_индекса* *ON* *имя_таблицы*

В версиях MySQL 3.22 и старше используется для удаления индексов таблиц. В версиях младше 3.22 этот оператор ничего не делает, а в 3.22 он фактически заменяется на *ALTER TABLE ... DROP INDEX*. Начиная с версии 5.1.22-ndb-6.2.5 вы можете удалять индексы в оперативном режиме, используя ключевое слово *ONLINE*. При оперативном удалении индекса не создается промежуточная временная копия таблицы. В *DROP INDEX* нельзя использовать опции *RESTRICT*, *CASCADE* и *IF EXISTS*.

PROCEDURE имя_процедуры

В версиях MySQL 5.1 и старше используется для удаления хранимых процедур. Вы также можете использовать *IF EXISTS*.

[TEMPORARY] TABLE имя_таблицы[, ...]

Удаляет одну или несколько таблиц (имена таблиц разделяются запятыми). MySQL удаляет описание таблицы, а также соответствующие файлы с расширениями *.FRM*, *.MYD* и *.MYI*. При выполнении этого оператора фиксируются все активные транзакции. Слово *TEMPORARY* позволяет удалить временную таблицу, при этом не фиксируются транзакции и не проверяются права доступа.

TRIGGER [имя_схемы.]имя_триггера

Используется в MySQL версий 5.0.2 и старше для удаления триггеров. Вы можете использовать *IF EXISTS* для удаления триггера только в том случае, если он реально существует.

VIEW имя_представления

Используется для удаления представления с указанным именем. Вы также можете использовать *IF EXISTS*.

IF EXISTS

Подавляет сообщение об ошибке в случае попытки удаления реально не существующего объекта. Поддерживается начиная с версии MySQL 3.22.

RESTRICT | CASCADE

Слова используются как синтаксические заглушки, не вызывают сообщений об ошибках и не имеют никакого другого эффекта.

MySQL поддерживает только удаление базы данных, таблицы или индекса из таблицы. Хотя в операторе *DROP* можно использовать ключевые слова *RESTRICT* и *CASCADE*, они не имеют реального эффекта. Вы можете использовать *IF EXISTS* для того, чтобы при попытке удаления несуществующего объекта не возникали сообщения об ошибках.

MySQL позволяет удалять и другие типы объектов с использованием аналогичного синтаксиса:

```
DROP { EVENT | FOREIGN KEY | LOGFILE GROUP | PREPARE |
  PRIMARY KEY | SERVER | TABLESPACE | USER }
имя_объекта
```

Эти варианты оператора *DROP* выходят за рамки этой книги. Более подробную информацию о них ищите в документации.

Oracle

В Oracle поддерживается большая часть вариантов оператора *DROP*, описанных в ANSI, а также некоторые другие варианты, соответствующие типам объектов, поддерживаемым только в Oracle. Oracle поддерживает удаление следующих типов объектов из SQL3:

```
DROP { DATABASE | FUNCTION | INDEX | PROCEDURE | ROLE |
  TABLE | TRIGGER | TYPE [BODY] | VIEW }
имя_объекта
```

Операторы *DROP* для разных объектов могут иметь различный вид, поэтому для каждого варианта оператора приводится его полный синтаксис:

DATABASE имя_базы_данных

Удаляет указанную базу данных.

FUNCTION имя_функции

Удаляет функции, не входящие в состав пакетов. (Если вы хотите удалить функцию из пакета, то используйте оператор *CREATE PACKAGE...OR REPLACE* для объявления пересоздания пакета без этой функции.) Любые локальные объекты, использующие удаленную функцию, становятся невалидными, а любые статистические типы, связанные с функцией, отвязываются от нее.

INDEX имя_индекса [FORCE]

Удаляет обычный или предметный индекс. Удаление индекса делает невалидными все объекты (представления, пакеты, функции, хранимые процедуры), зависящие от таблицы, по которой был создан индекс. Также удаление индекса делает невалидными курсоры и планы выполнения запросов, использовавшие индекс, что приводит к полным разборам при следующих выполнениях соответствующих операторов SQL.

Обычные индексы являются вторичными объектами и могут удаляться и пересоздаваться без потери пользовательских данных. Индексно-организованные таблицы комбинируют в себе и данные, и индекс, поэтому их нельзя удалять и пересоздавать таким же образом, как и обычные индексы. Индексно-организованные таблицы удаляются оператором *DROP TABLE*.

При удалении секционированного индекса удаляются все его секции, а при удалении индекса с составным секционированием удаляются все секции и подсекции. При удалении предметного индекса удаляются все связанные с ним статистики, и удаляется связь со всеми статистическими типами. Ключевое слово *FORCE* используется только при удалении предметного индекса. *FORCE* позволяет удалить предметный индекс, находящийся в состоянии *IN PROGRESS*, или для которого процедура *indextype* возвращает ошибку. Например:

```
DROP INDEX ndx_sales_salesperson_quota;
```

PROCEDURE имя_процедуры

Удаляет указанную хранимую процедуру. Любые зависимые объекты становятся невалидными, а попытки их использования завершаются с ошибками, пока вы не пересоздадите процедуру. Если вы пересоздали хранимую процедуру, а затем используете зависимый объект, то зависимый объект перекомпилируется.

ROLE имя_роли

Удаляет роль и отзывает ее у всех пользователей и других ролей. Вновь создаваемые сессии не могут использовать удаленную роль, но те сессии, которые использовали роль в момент ее удаления, не затрагиваются. Вот пример удаления роли:

```
DROP ROLE sales_mgr;
```

TABLE имя_таблицы [CASCADE CONSTRAINTS] [PURGE]

Удаляет указанную таблицу и все ее данные, индексы и триггеры (даже в другой схеме), делает невалидными все зависимые объекты, а также отменяются все разрешения на таблицу. Для секционированных таблиц удаляются все секции и подсекции. При удалении индексно-организованной таблицы удаляются и таблицы соответствия идентификаторов. От удаленной таблицы отвязываются все статистические типы. Если для таблицы были созданы журналы материализованных представлений, то они тоже удаляются.

Оператор *DROP TABLE* применяется к обычным, индексно-организованным и объектным таблицам. Удаленная таблица не удаляется полностью, а помещается в корзину, если только явно не указана опция *PURGE*, позволяющая сразу полностью освободить место, занимаемое таблицей. (В Oracle отдельно поддерживается оператор *PURGE*, с помощью которого можно удалить таблицу из корзины.) Для внешних таблиц *DROP TABLE* удаляет только метаданные, а файл с данными нужно удалять командами операционной системы.

Используйте фразу *CASCADE CONSTRAINTS* для удаления всех внешних ключей, ссылающихся на первичные и уникальные ключи удаляемой таблицы. Вы не сможете удалить таблицу, используемую в ограничениях ссылочной целостности, без использования *CASCADE CONSTRAINTS*.

В следующем примере удаляется таблица **job_desc** в схеме **emp**, а затем таблица **job** и все внешние ключи, ссылающиеся на первичный ключ таблицы **job**:

```
DROP TABLE emp.job_desc;
DROP TABLE job CASCADE CONSTRAINTS;
```

TRIGGER имя_триггера

Удаляет из базы данных указанный триггер.

TYPE [BODY] имя_типа [{FORCE|VALIDATE}]

Удаляет спецификацию и тело пользовательского объектного типа данных, вложенной таблицы или *VARRAY*. В случае если тип является супертипом, имеет связанный статистический тип или зависимости любого другого вида, то для удаления такого типа нужно использовать опцию *FORCE*. В этом случае все подтипы и статистические типы становятся невалидными. Oracle удаляет все публичные синонимы, связанные с удаляемым типом. Можно использовать опцию *BODY* для того, чтобы удалить только тела типа, но оставить его спецификацию. *BODY* нельзя использовать совместно с *FORCE* и *VALIDATE*. Используйте опцию *VALIDATE*, чтобы при удалении подтипа проверить существование экземпляров этого подтипа в любом из его супертипов. Удаление произойдет только при отсутствии сохраненных экземпляров. Пример удаления типа:

```
DROP TYPE salesperson_type;
```

VIEW имя_представления [CASCADE CONSTRAINTS]

Удаляет указанное представление, а также делает невалидными все зависимые представления, синонимы и материализованные представления. Используйте необязательную опцию *CASCADE CONSTRAINTS* для удаления всех ссылочных ограничений целостности, зависящих от представления. Без

использования этой опции и при наличии ссылочных ограничений целостности представление не будет удалено. Например, следующий оператор удаляет представление **active_employees** в схеме **hr**:

```
DROP VIEW hr.active_employees;
```

В операторе *DROP* перед именем объекта можно также указать схему. Если схема не указана, то используется схема по умолчанию для текущей сессии. Следующий оператор удаляет представление из схемы **sales_archive**:

```
DROP VIEW sales_archive.sales_1994;
```

А если ваша схема **scott**, то в следующем операторе подразумевается удаление представления **scott.sales_1994**:

```
DROP VIEW sales_1994;
```

Oracle поддерживает удаление большого числа объектов, не описанных в стандарте SQL3:

```
DROP { CLUSTER | CONTEXT | DATABASE LINK | DIMENSION |
  DIRECTORY | DISKGROUP | FLASHBACK ARCHIVE | INDEXTYPE |
  JAVA | LIBRARY | MATERIALIZED VIEW |
  MATERIALIZED VIEW LOG | OPERATOR | OUTLINE | PACKAGE |
  PROFILE | RESTORE POINT | ROLLBACK SEGMENT | SEQUENCE |
  SYNONYM | TABLESPACE | TYPE BODY | USER }
имя_объекта
```

Эти операторы находятся вне рамок книги. За информацией об этих операторах обращайтесь к документации (хотя базовый синтаксис всех этих операторов практически одинаковый).

PostgreSQL

В PostgreSQL не поддерживаются ключевые слова *RESTRICT* и *CASCADE*, описанные в стандарте. Поддерживает широкий набор вариантов *DROP* в соответствии со следующим синтаксисом:

```
DROP { DATABASE | DOMAIN | FUNCTION | INDEX | ROLE |
       SCHEMA | TABLE | TRIGGER | TYPE | VIEW }
[IF EXISTS]
имя_объекта
[ CASCADE | RESTRICT ]
```

Синтаксис для каждого оператора DROP из стандарта следующий:

DATABASE имя_базы_данных

Удаляет указанную базу данных и очищает все каталоги с данными удаленной базы. Этот оператор может быть выполнен только владельцем базы данных, причем пользователь должен быть подключен не к удаляемой базе. Например, мы можем удалить базу данных **sales_archive**:

```
DROP DATABASE sales_archive;
```

DOMAIN имя_домена [, ...] [CASCADE | RESTRICT]

Удаляет один или несколько указанных доменов, принадлежащих текущему пользователю. *CASCADE* позволяет автоматически удалить все объекты, за-

висящие от домена, а *RESTRICT* предотвращает удаление, если найдены зависящие объекты. Если не указана ни одна опция, то по умолчанию подразумевается *RESTRICT*.

FUNCTION имя_функции ([тип_данных1[, ...]) [CASCADE | RESTRICT]

Удаляет указанную пользовательскую функцию. Так как в PostgreSQL допускается создание функций с одинаковыми именами, отличающихся только входными параметрами, то вам может потребоваться указать типы входных параметров для точного указания функции, которую вы хотите удалить. PostgreSQL не выполняет проверки зависимостей при удалении пользовательских функций. (Если вы попытаетесь использовать объект, зависящий от удаленной функции, то получите сообщение об ошибке.) Например:

```
DROP FUNCTION median_distribution (int, int, int, int);
```

INDEX имя_индекса [, ...] [CASCADE | RESTRICT]

Удаляет один или несколько указанных индексов. Например:

```
DROP INDEX ndx_titles, ndx_authors;
```

ROLE имя_роли[, ...]

Удаляет одну или несколько указанных ролей. В PostgreSQL нельзя удалить роль, на которую ссылается хотя бы один объект в базе данных. Поэтому вам необходимо удалить либо поменять владельца для всех объектов, принадлежащих удаляемой роли, используя *REASSIGN OWNED* или *DROP OWNED*, а также отозвать у роли все привилегии.

SCHEMA имя_схемы[, ...] [CASCADE | RESTRICT]

Удаляет из текущей базы данных одну или несколько схем. Схема может быть удалена только суперпользователем базы данных или владельцем схемы (при том, что владелец схемы может и не владеть всеми объектами в схеме).

TABLE имя_таблицы[, ...] [CASCADE | RESTRICT]

Удаляет одну или несколько таблиц вместе с индексами и триггерами. Например:

```
DROP TABLE authors, titles;
```

TRIGGER имя_триггера ON имя_таблицы [CASCADE | RESTRICT]

Удаляет из базы данных указанный триггер. При удалении триггера требуется указывать также имя таблицы, так как в PostgreSQL имя триггера должно быть уникально только в пределах таблицы, к которой прикреплен триггер. То есть возможно наличие нескольких триггеров с именем **insert_trigger**, каждый из которых создан для своей таблицы. Например:

```
DROP TRIGGER insert_trigger ON authors;
```

TYPE имя_типа [, ...] [CASCADE | RESTRICT]

Удаляет из базы данных один или несколько пользовательских типов данных. PostgreSQL не проверяет влияние удаления типа на зависимые объекты, такие как функции, агрегаты и таблицы, вы должны проверять зависимости объектов самостоятельно. (Не удаляйте стандартные типы, поставляемые вместе с PostgreSQL!) Имейте в виду, что реализация типов в PostgreSQL

отличается от стандарта ANSI. Обратитесь к разделу *CREATE/ALTER TYPE* за дополнительной информацией.

VIEW имя_представления [, ...] [CASCADE | RESTRICT]

Удаляет одно или несколько указанных представлений.

CASCADE | RESTRICT

CASCADE автоматически удаляет все объекты, зависящие от основного удаляемого объекта. *RESTRICT* запрещает удаление при обнаружении зависимых объектов. По умолчанию используется поведение опции *RESTRICT*. Так как эти опции допустимы не в каждом варианте оператора *DROP*, то мы указали их в синтаксисе тех операторов, где возможно их использование.

IF EXISTS

Подавляет сообщение об ошибках при попытках удаления несуществующих объектов. Может использоваться с любыми вариантами оператора *DROP*.

Обратите внимание, что в PostgreSQL нельзя указать базу данных, для которой выполняется удаление (за исключением оператора *DROP DATABASE*). Поэтому для удаления объекта вам нужно подключиться к той базе данных, в которой этот объект находится.

PostgreSQL поддерживает удаление некоторых объектов, не описанных в стандарте SQL3:

```
DROP { AGGREGATE | CAST | CONVERSION | GROUP | LANGUAGE |
OPERATOR [CLASS] | RULE | SEQUENCE | TABLESPACE | USER } имя_объекта
```

Эти операторы находятся вне рамок книги. За информацией об этих операторах обращайтесь к документации (хотя базовый синтаксис всех этих операторов практически одинаковый).

SQL Server

В SQL Server поддерживается несколько описанных в SQL3 вариантов оператора *DROP*:

```
DROP { DATABASE | FUNCTION | INDEX | PROCEDURE | ROLE |
SCHEMA | TABLE | TRIGGER | TYPE | VIEW } имя_объекта
```

Вот полный синтаксис каждого варианта:

DATABASE имя_базы_данных[, ...]

Удаляет указанную базу данных и стирает с диска все ее файлы. Оператор может быть выполнен только из базы **master**. Реплицируемые базы данных перед удалением должны быть удалены из своих цепочек репликации. Вы не можете удалить используемую базу данных и любую из системных базы данных (**master**, **model**, **msdb**, **temp**). Например, мы можем удалить базы **northwind** и **pubs** одной командой:

```
DROP DATABASE northwind, pubs
GO
```

FUNCTION [схема.]имя_функции[, ...]

Удаляет из текущей базы данных одну или несколько пользовательских функций.

INDEX имя_индекса ON имя_таблицы_или_представления[, ...] [WITH {MAXDOP = целое_число | ONLINE = {ON | OFF} | MOVE TO местоположение [FILESTREAM_ON местоположение] }]

Удаляет один или несколько индексов таблицы или индексированного представления. Этот оператор не следует использовать для удаления первичного ключа или уникального ключа: их следует удалять оператором *ALTER TABLE ... DROP CONSTRAINT*. При удалении кластерного индекса таблицы удаляются все его некластерные индексы. При удалении кластерного индекса можно использовать фразу *WITH*. *MAXDOP* определяет максимальную степень параллелизма, используемую при удалении. Значение *MAXDOP* может быть равно 1 (для последовательного выполнения операции), 0 (для автоматического определения степени параллелизма), либо быть целым число больше 1 (для явного указания числа параллельных процессов). Если для параметра *ONLINE* установлено значение *ON*, то при удалении индексов допускаются запросы и обновления таблиц, а при значении *OFF* на таблицы накладываются блокировки на время удаления индексов. *MOVE TO* указывает файловую группу или секцию, в которую перемещается кластерный индекс.[1] В новое место кластерный индекс[2] перемещается уже в форме кучи.

PROC[EDURE] имя_процедуры[, ...]

Удаляет из текущей базы данных одну или несколько хранимых процедур. В SQL Server допускается наличие нескольких версий одной хранимой процедуры, но их нельзя удалять по отдельности: все они удаляются одновременно. Например:

```
DROP PROCEDURE calc_sales_quota
GO
```

ROLE имя_роли[, ...]

Удаляет из текущей базы данных одну или несколько ролей. Роли не должны принадлежать объекты, иначе удаление завершится с ошибкой. Эти объекты предварительно следует удалить или изменить им владельца.

SCHEMA имя_схемы

Удаляет пустую схему. Если в схеме находятся объекты, то их следует удалить или переместить в другую схему.

TABLE [имя_базы_данных.][имя_схемы.]имя_таблицы[, ...]

Удаляет указанную таблицу, все ее данные, индексы, триггеры и ограничения. (Вы можете указать только имя таблицы, если она принадлежит текущему пользователю и находится в текущей базе данных, или явно указать базу данных и схемы.) Представления, функции и хранимые процедуры, использующие удаленную таблицу, не удаляются и не помечаются как невалидные, но вернут ошибку при попытке их вызова. Вы не сможете удалить

---

[1] Вероятно, авторы имели в виду: «MOVE TO определяет место, куда будут перемещаться строки данных, находящиеся на конечном уровне кластеризованного индекса». – *Прим. науч. ред.*

[2] Вероятно, авторы имели в виду: «данные перемещаются уже в форме кучи». – *Прим. науч. ред.*

таблицу, на которую ссылается внешний ключ, если не удалите предварительно сам ключ. Аналогично, вы не сможете удалить реплицируемую таблицу, пока не исключите ее из репликационной схемы. Любые пользовательские правила и значения по умолчанию отвязываются от удаленной таблицы. Их потребуется привязывать к таблице заново, если она будет пересоздана.

TRIGGER имя_триггера[, ...] [ON {DATABASE|ALL SERVER}]

Удаляет из текущей базы данных один или несколько триггеров. Фраза [*ON {DATABASE|ALL SERVER}]* может быть использована при удалении триггеров на DDL, а фраза *[ON ALL SERVER]* используется также при удалении триггеров на *LOGON*. *ON DATABASE* означает, что областью действия удаляемого триггера является база данных, и эту фразу следует использовать, если эта же фраза использовалась при создании триггера. *ON ALL SERVER* означает, что областью действия триггера на DDL или *LOGON* является весь текущий сервер.

TYPE [имя_схемы.]имя_типа[, ...]

Удаляет один или несколько пользовательских типов данных из текущей базы данных.

VIEW [имя_схемы.]имя_представления[, ...]

Удаляет из базы данных обычное или индексированное представление и освобождает занимаемое им пространство.

В SQL Server поддерживается большое число объектов, расширяющих стандарт ANSI и для удаления которых используются более или менее стандартные формы оператора *DROP*:

```
DROP { AGGREGATE | APPLICATION ROLE | ASSEMBLY |
  ASYMMETRIC KEY | BROKER PRIORITY | CERTIFICATE |
  CONTRACT | CREDENTIAL | CRYPTOGRAPHIC PROVIDER |
  DATABASE AUDIT SPECIFICATION | DATABASE ENCRYPTIIN KEY |
  DEFAULT | ENDPOINT | EVENT NOTIFICATION | EVENT SESSION |
  FULLTEXT CATALOG | FULLTEXT INDEX | FULLTEXT STOPLIST |
  LOGIN | MASTER KEY | MESSAGE TYPE | PARTITION FUNCTION |
  PARTITION SCHEME | QUEUE | REMOTE SERVICE BINDING |
  ESOURCE POOL | ROUTE | SERVER AUDIT |
  SERVER AUDIT SPECIFICATION | SERVICE | SIGNATURE |
  STATISTICS | SYMMETRIC KEY | SYNONYM | USER |
  WORKLOAD GROUP | XML SCHEMA COLLECTION }
имя_объекта
```

Эти операторы находятся вне рамок книги. За информацией об этих операторах обращайтесь к документации (хотя базовый синтаксис всех этих операторов практически одинаковый).

См. также

CALL
CONSTRAINTS
CREATE/ALTER FUNCTION/PROCEDURE/METHOD
CREATE SCHEMA
CREATE/ALTER TABLE

CREATE/ALTER VIEW
DELETE
DROP
GRANT
INSERT
RETURN
SELECT
SUBQUERY
UPDATE

EXCEPT

Оператор *EXCEPT* формирует из результата двух или более операторов *SELECT* множество строк, включающее все строки из результата первого запроса, которые отсутствуют в результатах последующих запросов. В то время как фраза *JOIN* возвращает соответствующие друг другу строки из двух или более запросов, *EXCEPT* фильтрует строки, оставляя те, которые есть только в одной из нескольких однотипных таблиц.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с ограничениями
PostgreSQL	Поддерживается с ограничениями
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

Формально нет ограничений на число запросов, которое можно комбинировать посредством оператора *EXCEPT*. Общий синтаксис следующий:

```
{SELECT запрос1 | VALUES (выражение1[, ...])}
EXCEPT [ALL | DISTINCT]
[CORRESPONDING [BY (столбец1, столбец2, ...)]]
{SELECT запрос2 | VALUES (выражение2[, ...])}
EXCEPT [ALL | DISTINCT]
[CORRESPONDING [BY (столбец1, столбец2, ...)]]
...
```

Ключевые слова

VALUES (выражение1[, ...])

Создает множество строк с явно указанными значениями. По сути это множество можно рассматривать как результат оператора *SELECT*, но без использования синтаксиса *SELECT ... FROM*. Это называется конструктором строк, так как строки явным образом конструируются вручную. В соответствии со стандартом ANSI несколько конструкторов строк должны быть заключены в скобки и разделены запятыми.

EXCEPT

Определяет, какие строки исключаются из результирующего множества.

ALL | DISTINCT

ALL учитывает повторяющиеся строки во всех результирующих наборах. *DISTINCT* удаляет дубликаты из всех результирующих наборов до оператора *EXCEPT*. Столбцы со значениями NULL считаются одинаковыми. (*DISTINCT* используется по умолчанию, если явно не указана ни одна опция.)

CORRESPONDING [BY (столбец1, столбец 2, . . .)]

Указывает, что возвращаются только значения перечисленных столбцов, даже если в запросах используется звездочка.

Общие правила

Нужно помнить только одно важное правило, касающееся оператора *EXCEPT*: количество и порядок столбцов во всех запросах должны быть одинаковыми, а типы столбцов должны быть из одинаковых категорий.

Типы данных не должны быть идентичными, но должны быть совместимыми. Например, типы *CHAR* и *VARCHAR* совместимы. По умолчанию для каждого столбца будет использоваться наибольший (с точки зрения размерности) из типов столбца в каждом запросе. Например, если в запросах для столбца используются типы *VARCHAR(10)* и *VARCHAR(15)*, то в результирующем наборе столбец будет иметь типа *VARCHAR(15)*.

Советы и хитрости

Ни в одной из платформ не поддерживается фраза *CORRESPONDING BY*.

Если платформа не поддерживает *EXCEPT*, то вы можете использовать подзапрос с *NOT IN*. Однако подзапросы с *NOT IN* по-другому работают с пустыми значениями и могут давать другие результаты.

В соответствии с ANSI операторы *UNION* и *EXCEPT* имеют одинаковый приоритет. Однако *INTERSECT* выполняется раньше других операторов работы с множествами. Мы рекомендуем использовать скобки для явного определения порядка выполнения операторов.

В соответствии с ANSI допускается только одна фраза *ORDER BY* для всего запроса. Она должна быть указана после последнего оператора *SELECT*. Для исключения неоднозначности в именах столбцов используйте одинаковые псевдонимы для всех столбцов во всех запросах. Например:

```
SELECT au_lname AS 'lastname', au_fname AS 'firstname'
FROM authors
EXCEPT
SELECT emp_lname AS 'lastname', emp_fname AS 'firstname'
FROM employees
ORDER BY lastname, firstname
```

Хотя наборы столбцов запросов могут иметь совместимые типы данных, в некоторых платформах могут быть различия в обработке длин столбцов. Например,

если столбец **au_lname** из первого запроса имеет значительно большую длину, чем столбец **emp_name** из второго запроса, то правила выбора длины столбца для конечного результата могут быть различными в разных платформах. Хотя обычно просто выбирается тип с наибольшей длиной (наименее ограниченный).

Помните, что в качестве альтернативы вы можете использовать операторы *NOT IN* и *NOT EXISTS* с коррелированными подзапросами. Следующие запросы демонстрируют, как получить функциональность *EXCEPT* при помощи *NOT IN* и *NOT EXISTS*:

```
SELECT DISTINCT a.city
FROM pubs..authors AS a
WHERE NOT EXISTS
  (SELECT *
   FROM pubs..publishers AS p
   WHERE a.city = p.city)
```

```
SELECT DISTINCT a.city
FROM pubs..authors AS a
WHERE a.city NOT IN
  (SELECT p.city
   FROM pubs..publishers AS p
   WHERE p.city IS NOT NULL)
```

В общем случае *NOT EXISTS* работает быстрее, чем *NOT IN*. Также имеется неприятный момент в правилах обработки значений NULL, отличающихся для *IN*/*NOT IN* и *EXISTS*/*NOT EXISTS*. Для обхода возможных проблем просто используйте в *WHERE* фразу *IS NOT NULL*, как показано в предыдущем примере.

Каждая СУБД может иметь свои правила именования столбцов в случае, когда столбцы в запросах имеют разные названия. Обычно используются названия столбцов из первого запроса.

MySQL

EXCEPT не поддерживается в MySQL. Однако, как сказано в предыдущем разделе, вы можете в качестве альтернативы использовать *NOT IN* или *NOT EXISTS*.

Oracle

Oracle не поддерживает оператор *EXCEPT*. Однако он поддерживает оператор *MINUS* с аналогичной функциональностью:

```
<SELECT оператор1>
MINUS
<SELECT оператора2>
MINUS
...
```

MINUS DISTINCT и *MINUS ALL* не поддерживаются. *MINUS* является функциональным эквивалентом *MINUS DISTINCT*. Oracle не поддерживает *MINUS* для следующих запросов:

- Запросов со столбцами типов *LONG*, *BLOB*, *CLOB*, *BFILE*, *VARRAY*.
- Запросов с фразой *FOR UPDATE*.
- Запросов с коллекциями *TABLE*.

Если первый запрос содержит выражения в списке столбцов, то для этих выражений при помощи *AS* должны быть заданы псевдонимы. Только последний запрос может содержать фразу *ORDER BY*.

Например, получим список идентификаторов всех магазинов, для которых нет записей в таблице **sales**:

```
SELECT stor_id FROM stores
MINUS
SELECT stor_id FROM sales
```

Оператор *MINUS* функционально эквивалентен подзапросу с *NOT IN*. Следующий запрос получает тот же самый результат:

```
SELECT stor_id FROM stores
WHERE stor_id NOT IN
  (SELECT stor_id FROM sales)
```

PostgreSQL

PostgreSQL поддерживает операторы *EXCEPT* и *EXCEPT ALL* в соответствии с базовым синтаксисом ANSI:

```
<SELECT оператор1>
EXCEPT [ALL]
<SELECT оператор2>
EXCEPT [ALL]
...
```

PostgreSQL не поддерживает *EXCEPT* и *EXCEPT ALL* для запросов с фразой *FOR UPDATE*. *EXEPT DISTINCT* не поддерживается, но *EXCEPT* функционально ему эквивалентен. Также не поддерживается фраза *CORRESPONDING*.

Первый запрос не может содержать фразы *ORDER BY* и *LIMIT*, но последующие подзапросы могут содержать эти фразы, при этом эти подзапросы должны быть заключены в скобки. В противном случае самые правые вхождения *ORDER BY* и *LIMIT* будут применены к конечному результату.

PostgreSQL выполняет операторы *SELECT* сверху вниз, если только иной порядок явно не определен скобками.

Обычно повторяющиеся строки исключаются из результатов обоих запросов, если только вы не используете *EXCEPT ALL*. Например, вы можете получить список всех заголовков из таблицы **authors**, не имеющих вхождений в таблицу **sales**:

```
SELECT title_id
FROM authors
EXCEPT ALL
SELECT title_id
FROM sales;
```

SQL Server

EXCEPT поддерживается в SQL Server, но не поддерживаются подфразы *CORRESPONDING*, *ALL* и *DISTINCT*. При сравнении наборов строк SQL Server считает, что NULL равно NULL. При использовании *SELECT...INTO* фраза *INTO* может встречаться только в первом запросе. *ORDER BY* допускается только в конце

всего оператора, но не в отдельных запросах. *GROUP BY* и *HAVING* допускаются только в отдельных запросах, но не для финального результата. В запросах внутри *EXCEPT* также нельзя использовать *FOR BROWSE*.

См. также

INTERSECT
SELECT
UNION

EXISTS

Оператор *EXISTS* тестирует подзапрос на наличие строк.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
SELECT ...
WHERE [NOT] EXISTS (подзапрос)
```

Ключевые слова и параметры следующие:

WHERE [NOT] EXISTS

Тестирует подзапрос на наличие строк. Если подзапрос содержит хотя бы одну строку, то оператор возвращает булево значение *TRUE*. Если используется *NOT*, то оператор возвращает *TRUE* в случае, когда подзапрос возвращает пустой результат.

подзапрос

Определяет подзапрос, который тестируется на наличие строк.

Общие правила

Оператор *EXISTS* проверяет для каждой строки запроса наличие строк в подзапросе.

Например, если мы хотим узнать, существуют ли названия должностей, на которых не работает ни один сотрудник, то мы можем выполнить следующий запрос:

```
SELECT *
FROM jobs
WHERE NOT EXISTS
  (SELECT * FROM employee
  WHERE jobs.job_id = employee.job_id)
```

В этом примере проверяется отсутствие строк в подзапросе при помощи дополнительного ключевого слова *NOT*. В следующем примере запрос возвращает строки, для которых найдены строки в подзапросе:

```
SELECT au_lname
FROM authors
WHERE EXISTS
  (SELECT *
  FROM publishers
  WHERE authors.city = publishers.city)
```

Советы и хитрости

EXISTS в большинстве запросов работает так же, как *ANY* (фактически, операторы *EXISTS* и *ANY* эквивалентны). *EXISTS* обычно наиболее эффективен при использовании коррелированных подзапросов.

Есть два варианта написания подзапроса при использовании *EXISTS*. Первый вариант – использовать звездочку (т. е. *SELECT * FROM*...) и не указывать какой-то конкретный столбец. В этом случае звездочка означает «любой столбец». Второй вариант – указать только один столбец в подзапросе (т. е. *SELECT* `au_id` *FROM*...). В некоторых СУБД возможны подзапросы, извлекающие более одного столбца (т. е. *SELECT* `au_id, au_lname` *FROM* ...). Тем не менее, такая возможность встречается достаточно редко, и следует избегать ее использования, если код должен быть переносимым между платформами.

Платформенные особенности

Все СУБД поддерживают оператор *EXISTS* в описанном виде.

См. также

ALL/ANY/SOME

SELECT

WHERE

FETCH

Оператор *FETCH* – это одна из четырех команд (наравне с *DECLARE*, *OPEN* и *CLOSE*), используемых при работе с курсорами. Курсоры позволяют обрабатывать результаты запроса не одним массивом, а построчно. *FETCH* позиционирует курсор на конкретной строке и извлекает строку из результирующего множества.

Использование курсоров очень полезно, так как реляционные СУБД оперируют множествами строк, а большинство клиентских приложений работают с данными построчно. Курсоры дают возможность обрабатывать в каждый момент времени только одну запись, что и требуется для клиентских программ.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Поддерживается с ограничениями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
FETCH [  {NEXT | PRIOR | FIRST | LAST |
{ ABSOLUTE целое число | RELATIVE целое число }}
FROM ] имя курсора
[INTO переменная1[,…]]
```

Ключевые слова

NEXT

Используется для перехода к строке, которая следует сразу за текущей строкой курсора. *FETCH NEXT* является командой по умолчанию для *FETCH*. Если *FETCH* выполняется для курсора первый раз, то возвращается первая строка.

PRIOR

Используется для перехода к предыдущей строке относительно текущей строки курсора. *FETCH PRIOR* не извлекает строку, если это первая команда *FETCH*, выполненная для этого курсора.

FIRST

Используется для перехода к первой строке курсора.

LAST

Используется для перехода к последней строке курсора.

ABSOLUTE n

Используется для перехода к *n*-й строке курсора (отсчет ведется от начала при положительном *n* или от конца – при отрицательном). Если *n* равно нулю, то строка из курсора не извлекается. Если *n* больше числа записей в курсоре, то курсор позиционируется за последней записью (при положительном *n*) или перед первой записью (при отрицательном *n*).

RELATIVE n

Используется для перехода к записи, являющейся *n*-й по счету после текущей записи (при положительном *n*) или перед текущей записью (при отрицательном *n*). Если *n* выводит курсор за границы результирующего множества, то курсор позиционируется за последней записью (при положительном *n*) или перед первой записью (при отрицательном *n*).

[FROM] имя курсора

Указывает имя курсора, из которого нужно извлечь запись. Курсор должен быть предварительно создан и открыт с помощью команд *DECLARE* и *OPEN*. Слово *FROM* необязательно для использования, но желательно.

[INTO] переменная1[,...]

Используется для сохранения значений столбцов извлекаемой из курсора записи в локальных переменных. Для каждого столбца курсора в *INTO* должна быть указана переменная соответствующего типа, столбцы и переменные соотносятся по порядку следования.

Общие правила

Основными шагами работы с курсором являются:

1. Создание курсора (команда *DECLARE*).
2. Открытие курсора с помощью (команда *OPEN*).
3. Работа с курсором (команда *FETCH*).
4. Закрытие курсора (команда *CLOSE*).

Выполняя эти шаги, можно создать результат, аналогичный команде *SELECT*, и при этом работать с каждой строкой результата отдельно.

В каждой СУБД есть свои правила использования переменных. Например, в SQL Server используется префикс в виде символа (@), в Oracle и PostgreSQL префикс не нужен и т. д. Стандарт SQL указывает, что использование двоеточия (:) в качестве префикса необходимо для языков вроде C и COBOL, но префикс не нужен для процедурного SQL.

Курсор может быть позиционирован либо на строке, либо в позиции перед первой строкой, либо после последней строки. Когда курсор позиционирован на строке, эта строка называется текущей. Можно выполнить *UPDATE* или *DELETE* текущей строки, используя конструкцию *WHERE CURRENT OF*.

Важно помнить, что в курсоре нельзя перемещаться по кругу. Например, если в курсоре 10 строк и выполняется переход на 12 строк вперед, то курсор не пройдется до конца, а потом еще раз с начала на 2 записи. Вместо этого курсор остановится после последней записи при движении вперед либо перед первой записью при движении назад. К примеру, на SQL Server:

```
 FETCH RELATIVE 12 FROM employee_cursor
INTO @emp_last_name, @emp_first_name, @emp_id
```

При извлечении значений из курсора в переменные проверьте, что количество столбцов в курсоре и количество переменных совпадают и также совпадают их типы. В противном случае вы получите ошибку. Например, следующая команда выполнится с ошибкой, т. к. в курсоре три столбца, а в команде *FETCH* используется только две переменных.

```
FETCH PRIOR FROM employee_cursor
INTO @emp_last_name, @emp_id
```

Советы и хитрости

Наиболее частой ошибкой при работе с курсором с помощью оператора *FETCH* является несоответствие в количестве, типах или порядке между переменными и столбцами курсора. Так что при написании *FETCH* проверьте, что вам известны количество и типы всех столбцов курсора.

Обычно база данных блокирует как минимум текущую строку курсора, но иногда и все строки. В соответствии с ANSI-стандартом блокировки не сохраняются при выполнении *ROLLBACK* или *COMMIT*, хотя в поведении в каждой СУБД может иметь свои особенности.

Хотя оператор *FETCH* и рассматривается в этом разделе сам по себе, он всегда должен использоваться вместе с *DECLARE*, *OPEN* и *CLOSE*. Например, при каждом открытии курсора используется память сервера базы данных. Если вы забудете закрыть курсор, это может привести к проблемам с нехваткой памяти. Так что всегда проверяйте, что каждый открытый курсор корректно закрывается.

Курсоры часто используются в хранимых процедурах для пакетной обработки данных. Причиной является необходимость выполнять какие-то действия над каждой строкой в отдельности, а не над всем множеством сразу. Но из-за того что курсоры работают с отдельными записями, а не с множествами записей, они часто оказываются медленнее, чем другие способы доступа к данным. Важно критически анализировать необходимость курсоров в каждой ситуации. Многие задачи, такие как заковыристые *DELETE* или сложные *UPDATE*, можно решать с помощью грамотного использования *JOIN* и *WHERE*, а не с помощью курсоров.

MySQL

MySQL поддерживает только минимальные возможности оператора *FETCH*, описанные в SQL3:

```
FETCH имя_курсора INTO переменная1[, ...]
```

MySQL извлекает из открытого курсора текущую строку (если она имеется) и перемещает указатель курсора на следующую строку. Если в курсоре больше нет строк, то MySQL устанавливает *SQLSTATE* в значение '02000' и порождает событие *NO DATA*, что позволяет вам корректно обработать такую ситуацию.

Oracle

Курсоры в Oracle являются однонаправленными и позволяют передвигаться за один шаг только на одну запись. Поэтому с точки зрения стандарта ANSI курсоры в Oracle поддерживают только *FETCH NEXT 1*. Данные из курсора можно извлекать либо построчно в локальные переменные, либо с помощью фразы *BULK COLLECT* извлекать сразу набор записей в массив. Ключевые слова *PRIOR*, *ABSOUTE* и *RELATIVE* не поддерживаются. Однако и однонаправленные, и прокручиваемые курсоры поддерживаются через программный интерфейс Oracle Call Interface (OCI). В OCI поддерживаются слова *PRIOR*, *ABSOLUTE* и *RELATIVE* для необновляемых курсоров, чьи наборы данных представляют собой мгновенные снимки для согласованного чтения.

Синтаксис *FETCH* в Oracle следующий:

```
FETCH имя_курсора
{ INTO переменная1[, ...] | BULK COLLECT INTO
  коллекция[, ...] [LIMIT целое_число] }
```

где:

BULK COLLECT INTO коллекция

Извлекает из курсора весь набор данных или определенное число записей (*LIMIT*) в указанный массив или переменную-коллекцию на стороне клиента.

LIMIT целое_число

Ограничивает число записей (значение должно быть неотрицательной целой константой или переменной), извлекаемое при использовании *BULK COLLECT*.

Oracle поддерживает динамические курсоры, чей текст можно определять во время выполнения. Имя курсора может передаваться через переменные или параметры. Также вы можете либо сами определять тип переменных для сохранения записей курсора, либо использовать *%ROWTYPE*. Все это позволяет создавать гибкие динамические курсоры, работающие как шаблоны. Например:

```
DECLARE
  TYPE namelist IS TABLE OF employee.lname%TYPE;
  names namelist;
  CURSOR employee_cursor IS SELECT lname FROM employee;
BEGIN
  OPEN employee_cursor;
  FETCH employee_cursor BULK COLLECT INTO names;
  ...
  CLOSE employee_cursor;
END;
```

Оператор *FETCH* часто используется в паре с оператором цикла *FOR* (или с каким-либо другим оператором цикла) для итерации по всем записям курсора. Для обнаружения конца курсора следует использовать атрибуты *%FOUND* и *%NOTFOUND*. Например:

```
DECLARE
  TYPE employee_cursor IS REF CURSOR
    RETURN employee%ROWTYPE;
  employee_cursor EmpCurTyp;
  employee_rec employee%ROWTYPE;
BEGIN
  LOOP
    FETCH employee _cursor INTO employee_rec;
    EXIT WHEN employee _cursor%NOTFOUND;
    ...
  END LOOP;
  CLOSE employee_cursor;
END;
```

В этом примере используется стандартный цикл PL/SQL, а когда в курсоре заканчиваются записи, для выхода из цикла используется оператор *EXIT*.

PostgreSQL

В PostgreSQL поддерживаются курсоры, по которым можно перемещаться в любом направлении, а также расширенный по сравнению с SQL3 набор режимов работы. Синтаксис оператора FETCH в PostgreSQL следующий:

```
FETCH { FORWARD [ {ALL | целое_число} ] |
  BACKWARD [ {ALL | целое_число } ] |  ABSOLUTE целое_число
  | RELATIVE целое_число | целое_число | ALL |
```

```
    NEXT | PRIOR | FIRST | LAST }
  { IN | FROM } имя_курсора
  [ INTO :переменная1 [, ...] ]
```

где:

FORWARD [{ALL | n}]

Если не указаны опциональные ключевые слова, то просто извлекает из курсора следующую запись (так же как *NEXT*). *FORWARD ALL* извлекает все записи, находящиеся после текущей позиции курсора, и позиционирует курсор за последней записью. *FORWARD n* извлекает *n* записей, находящихся после текущей позиции курсора, и позиционирует курсор после последней извлеченной записи.

BACKWARD [{ALL | n}]

Если не указаны опциональные ключевые слова, то просто извлекает из курсора предыдущую запись. *BACKWARD ALL* извлекает все записи, находящиеся перед текущей позицией курсора, и позиционирует курсор перед первой записью. *BACKWARD n* извлекает *n* записей, находящихся до текущей позиции курсора, и позиционирует курсор перед последней извлеченной записью.

ABSOLUTE n

Извлекает запись в позиции с указанным абсолютным номером.

RELATIVE n

Извлекает указанно число последующих (при положительном значении) или предыдущих записей (при отрицательном)

целое_число

Это число определяет, на сколько записей осуществляется переход. При положительном значении переход осуществляется вперед, при отрицательном – назад.

ALL

Извлекает из курсора все оставшиеся записи.

NEXT

Извлекает из курсора одну следующую запись.

PRIOR

Извлекает из курсора одну предыдущую запись.

IN | FROM имя_курсора

Указывает курсор, из которого извлекаются данные.

INTO переменная

Указывает переменную, в которой сохраняется извлеченное из курсора значение. Как и в случае со стандартными курсорами ANSI, столбцы курсора и переменные должны соответствовать друг другу по количеству, типу и порядку.

Для использования курсоров в PostgreSQL необходимо явно начинать транзакцию с помощью *BEGIN*, а для закрытия курсора использовать *COMMIT* или *ROLLBACK*.

В следующем примере из таблицы **employee** извлекаются и выводятся 5 строк:

```
FETCH FORWARD 5 IN employee_cursor;
```

В PostgreSQL есть отдельный оператор *MOVE* для перехода по курсору. Он отличается от *FETCH* только тем, что не извлекает значения в переменные:

```
MOVE { [ FORWARD | BACKWARD | ABSOLUTE | RELATIVE] }
  { [целое_число | ALL | NEXT | PRIOR ] }
{ IN | FROM } имя_курсора
```

В следующем примере открывается курсор, первые пять записей пропускаются и извлекается значение шестой записи:

```
BEGIN WORK;
  DECLARE employee_cursor CURSOR FOR SELECT * FROM employee;
  MOVE FORWARD 5 IN employee_cursor;
  FETCH 1 IN employee_cursor;
  CLOSE employee_cursor;
COMMIT WORK;
```

Этот код вернет из курсора **employee_cursor** только одну строку.

SQL Server

Поддержка оператора *FETCH* в SQL Server очень близка к стандарту ANSI:

```
FETCH [ { NEXT | PRIOR | FIRST | LAST |
{ ABSOLUTE целое_число| RELATIVE целое_число } } ]
[FROM] [GLOBAL] имя_курсора
[INTO @переменная1[, ...]]
```

Отличия между реализацией SQL Server и стандартом ANSI состоят в следующем. Во-первых, SQL Server позволяет вместо имени курсора и конкретных значений для перехода использовать переменные. Во-вторых, SQL Server позволяет создавать глобальные курсоры, которые могут быть использованы любой сессией или пользователем, а не только создателем.

Есть некоторые правила использования оператора *FETCH*, зависящие от того, какие параметры курсора вы использовали в *DECLARE CURSOR*:

- Если вы объявили курсор типа *SCROLL* и использовали синтаксис стандарта SQL92, то поддерживаются все варианты *FETCH*. Для других курсоров в формате SQL92 поддерживается только *FETCH NEXT (также поддерживается альтернативный стиль объявления курсора с помощью фразы DECLARE CURSOR)*.
- Курсоры *DYNAMIC SCROLL* поддерживают все варианты *FETCH*, за исключением *ABSOLUTE*.
- Курсоры *FORWARD_ONLY* и *FAST_FORWARD* поддерживают только *FETCH NEXT*.
- Курсоры *KEYSET* и *STATIC* поддерживают все варианты *FETCH*.

В SQL Server кроме закрытия курсора оператором *CLOSE* требуется также освобождать используемые курсором ресурсы при помощи оператора *DEALLOCATE*.

Вот полный пример, в котором курсор сперва объявляется и открывается, затем из него извлекаются несколько записей и наконец, курсор закрывается:

```
DECLARE @vc_lname VARCHAR(30),
        @vc_fname VARCHAR(30), @i_emp_id CHAR(5)
DECLARE employee_cursor SCROLL CURSOR FOR
  SELECT lname, fname, emp_id
  FROM employee
  WHERE hire_date <= 'FEB-14-2004'
OPEN employee_cursor
-- Извлекаем последнюю запись курсора.
FETCH LAST FROM employee_cursor
-- Извлекаем из курсора предыдущую запись.
FETCH PRIOR FROM employee_cursor
-- Извлекаем из курсора пятую по счету запись.
FETCH ABSOLUTE 5 FROM employee_cursor
-- Извлекаем запись, находящуюся через одну от текущей.
FETCH RELATIVE 2 FROM employee_cursor
-- Извлекаем 8 предыдущих записей в переменные.
FETCH RELATIVE -8 FROM employee_cursor
INTO @vc_lname, @vc_fname, @i_emp_id
CLOSE employee_cursor
DEALLOCATE employee_cursor
GO
```

Помните, что в SQL Server нужно не только закрывать курсоры, но и освобождать их ресурсы. В некоторых редких случаях может потребоваться переоткрыть закрытый курсор. В таком случае вы можете повторно использовать курсор, который был закрыт, но ресурсы которого не были освобождены. Курсор полностью удаляется только при освобождении его ресурсов.

См. также

CLOSE
DECLARE CURSOR
OPEN

GRANT

Оператор *GRANT* предоставляет привилегии пользователям и ролям, позволяя им получить доступ и использовать объекты базы данных. Также в большинстве платформ оператор *GRANT* используется для предоставления прав на создание объектов, выполнение хранимых процедур, функций и т. д. Другими словами, в большинстве платформ *GRANT* используется как для объектных привилегий, так и для разрешений на использование.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

Согласно стандарту SQL2003 с помощью оператора *GRANT* пользователю предоставляются объектные привилегии и роли:

```
GRANT { {объектная_привилегия[, ...] | роль[, ...]} }
[ON объект_базы_данных]
[TO получатель[, ...]]
[WITH HIERARCHY OPTION] [WITH GRANT OPTION]
[WITH ADMIN OPTION] [FROM {CURRENT_USER | CURRENT_ROLE}]
```

Ключевые слова

GRANT объектная_привилегия

Предоставляет привилегию на выполнение различных SQL операторов. Можно выдать одним оператором несколько привилегий, разделив их запятыми. *ALL PRIVILEGES* нельзя совмещать с другими привилегиями, но остальные привилегии можно комбинировать любым образом. Возможны следующие привилегии:

ALL PRIVILEGES

Предоставляет пользователю все привилегии на определенные объекты базы данных, которые есть у пользователя, выполняющего *GRANT*. Этот вариант использовать не рекомендуется, так как он способствует небрежности при работе с привилегиями.

EXECUTE

Предоставляет привилегию на вызов хранимой процедуры, пользовательской функции или метода.

SELECT | INSERT | UPDATE | DELETE

Предоставляет привилегию на выполнение соответствующего оператора для таблицы или представления. Вы можете также указать в скобках список столбцов, на которые распространяется привилегия (кроме привилегии на *DELETE*).

REFERENCES

Позволяет использовать столбцы таблицы в любых ограничениях целостности. Во фразе *REFERENCES* также можно указывать список столбцов, на которые выдается привилегия. Наконец, *REFERENCES* дает привилегию на создание внешних ключей, ссылающихся на указанный объект как родительский.

TRIGGER

Дает привилегию на создание триггера для указанной таблицы.

UNDER

Дает привилегию на создание подтипов и типизированных таблиц.

USAGE

Дает привилегию на использование домена, пользовательского типа, кодировки, схемы упорядочения, последовательности или перевода.

GRANT роль[,...]

Выдает пользователю указанную роль. Эта роль должна быть доступна пользователю или роли, указанной во *FROM*. Например, администратор базы данных мог бы создать роль *Reporter*, имеющую доступ на чтение нескольких таблиц. Последующая выдача пользователю этой роли дает ему привилегии, которые есть у роли.

ON объект_базы_данных

Указывает объект, на который выдается привилегия. Фраза *объект_базы_данных* имеет следующий синтаксис:

```
{[TABLE] имя_объекта | DOMAIN имя_объекта |
COLLATION имя_объекта | CHARACTER SET имя_объекта |
TRANSLATION имя_объекта | SPECIFIC ROUTINE имя_объекта }
```

TO получатель

Определяет пользователя или роль, получающих привилегию. Вы можете выдать привилегию нескольким пользователям или ролям, перечислив их через запятую. Привилегию можно выдать роли *PUBLIC*, тогда выбранные привилегии будут выданы всем пользователям базы данным, как существующим, так и будущим.

WITH HIERARCHY OPTION

Позволяет выполнять *SELECT* не только для указанной таблицы, но и для всех ее подтаблиц. Используется только при выдаче привилегий.

WITH GRANT OPTION

Позволяет пользователю передавать дальше полученные привилегии. Используется только при выдаче привилегий.

WITH ADMIN OPTION

Дает пользователю возможность назначать полученную роль другим пользователям.

FROM {CURRENT_USER|CURRENT_ROLE}

Указывает, что привилегии выдаются либо от имени текущего пользователя (*CURRENT_USER*), либо от имени текущей роли (*CURRENT_ROLE*). Эта фраза опциональна и предполагает использование текущего пользовательского контекста.

Общие правила

Вы можете одним оператором выдать несколько привилегий, но не следует в одном операторе *GRANT* одновременно использовать *ALL PRIVILEGES* вместе с другими ключевыми словами. Вы можете выдать отдельную привилегию на отдельный объект базы данных отдельному пользователю, используя следующий синтаксис:

```
GRANT привилегия ON объект TO пользователь
```

Например:

```
GRANT SELECT ON employee TO Dylan;
```

Вы можете выдать определенные привилегии сразу всем пользователям при помощи *PUBLIC*. Если привилегия выдана для *PUBLIC*, то эта привилегия доступна всем пользователям. Например:

```
GRANT SELECT ON employee TO PUBLIC;
```

Для выдачи привилегий нескольким пользователям просто перечислите пользователей через запятую:

```
GRANT SELECT ON employee TO Dylan, Matt, PUBLIC
```

Предыдущий пример показывает, что одним оператором можно выдать привилегию нескольким пользователям, в данном случае это пользователи *Dylan*, *Matt* и *PUBLIC*. При выдаче привилегий на таблицу вы можете ограничить привилегии определенным набором столбцов, перечислив их после имени таблицы в скобках через запятую:

```
GRANT SELECT
  ON employee(emp_id, emp_fname, emp_lname, job_id)
TO Dylan, Matt
```

В этом примере показано, что пользователи *Dylan* и *Matt* получают привилегию не на все столбцы таблицы **employee**.

Советы и хитрости

В зависимости от платформы привилегии представлений могут быть зависимыми или независимыми от привилегий базовых таблиц.

Все необязательные фразы *WITH* относятся к привилегиям, выдаваемым в том же операторе. Например, этот оператор выдает пользователю привилегию на чтение таблицы **employee**:

```
GRANT SELECT ON employee TO Dylan;
```

А следующий оператор выдает привилегию на чтение таблицы плюс возможность выдать эту привилегию другому пользователю:

```
GRANT SELECT ON employee TO Dylan
WITH GRANT OPTION;
```

Аналогичным образом вы можете использовать оператор *REVOKE* для отзыва только опции *WITH GRANT OPTION*, отдельной привилегии *SELECT* или и того и другого.

В большинстве платформ различаются привилегии, выданные пользователю напрямую и через роли. Например, если пользователь является членом двух ролей, то отдельная объектная привилегия может быть выдана ему четырежды: один раз напрямую, один раз через *PUBLIC* и отдельно через каждую роль. В этой ситуации будьте внимательны: для полного отзыва у пользователя привилегий на объект потребуется исключить пользователя из роли *PUBLIC*[1], а также отозвать привилегию у самого пользователя и у всех его ролей.

---

[1] Возможно, авторы имели в виду отозвать привилегию у роли PUBLIC. – *Прим. науч. ред.*

MySQL

MySQL предоставляет широкий набор привилегий, в основном относящихся к работе с объектами базы данных. Оператор *GRANT* поддерживается, начиная с версии 3.22.11. Синтаксис следующий:

```
GRANT [ { ALL [PRIVILEGES] |
  {SELECT | INSERT | UPDATE} [ (столбец[, ...]) ] | DELETE |
  REFERENCES [ (столбец[, ...]) ] } |
  {[{ALTER | CREATE | DROP} [опция_dml]] | [EVENT] |
  [EXECUTE] | [FILE] | [INDEX] | [LOCK TABLES] | [PROCESS] |
  [RELOAD] | [REPLICATION {CLIENT | SLAVE}] |
  [SHOW DATABASES] | [SHOW VIEW] | [SHUTDOWN] |
  [SUPER] | [TRIGGER] | [USAGE]}[, ...]
ON [ {TABLE | FUNCTION | PROCEDURE} ]
  { [база_данных.]имя_таблицы | * | *.* | база_данных.* }
TO получатель [IDENTIFIED BY [PASSWORD] 'пароль'][, ...]
[REQUIRE параметры_безопасности]
[WITH опции_with[...]]
```

где:

ALL [PRIVILEGES]

Дает пользователю все привилегии (кроме *WITH GRANT OPTION*), применимые для одного или нескольких указанных объектов базы данных. Например, *GRANT ALL ON *.** глобально дает все привилегии кроме *WITH GRANT OPTION*.

SELECT | INSERT | UPDATE | DELETE | REFERENCES

Дает разрешение соответственно на чтение, вставку, модификацию и удаление данных из таблицы (подфраза *REFERENCES* не реализована). Разрешения на уровне столбцов можно выдавать на *SELECT*, *INSERT* и *UPDATE*, но не на *DELETE*. Использование разрешений на уровне столбцов может немного замедлить работу MySQL, так как при этом требуется проверять большее число привилегий.

{ALTER | CREATE | DROP} [опция_dml]

Дает возможность создавать, модифицировать и удалять таблицы и другие объекты базы данных. Если используется *CREATE*, то ключевое слово *ON* для указания имения таблицы не обязательно. Используйте *ALTER*, *CREATE* или *DROP* и имя объекта (в этом случае под объектом подразумевается таблица). Вы можете детализировать эту подфразу следующим образом:

{CREATE | ALTER | DROP} ROUTINE

Дает разрешение на создание, модификацию и удаление хранимых процедур и функций.

CREATE TEMPORARY TABLE

Дает разрешение на выполнение оператора *CREATE TEMPORARY TABLE*.

CREATE USER

Дает привилегии на создание, переименование и удаление пользователей, а также на выполнение оператора *REVOKE ALL PRIVILEGES*.

CREATE VIEW

Дает разрешение на выполнение оператора *CREATE VIEW*.

EVENT

Дает привилегии на создание событий для планировщика событий.

EXECUTE

Дает разрешения на вызов хранимых процедур и функций.

FILE

Дает возможность читать данные из файлов и писать в файлы, используя операторы *SELECT INTO* и *LOAD DATA*.

INDEX

Дает привилегии на создание и удаление индексов.

LOCK TABLES

Дает возможность блокировать оператором *LOCK TABLES* те таблицы, на которые пользователь имеет привилегию *SELECT*.

PROCESS

Дает права на просмотр исполняющихся процессов с помощью оператора *SHOW PROCESSLIST*.

RELOAD

Дает разрешение на исполнение команд *FLUSH* и *RESET*.

REPLICATION {CLIENT | SLAVE}

Дает пользователю возможность читать метаданные о репликационных процессах (*REPLICATION CLIENT*) или дает подчиненному процессу читать журналы из репликационного мастер-процесса (*REPLICATION SLAVE*).

SHOW DATABASES

Дает пользователю привилегию на выполнение команды *SHOW DATABASES*.

SHOW VIEW

Дает разрешение на выполнение *SHOW CREATE VIEW*.

SHUTDOWN

Дает права на использование команды *MYSQLADMIN SHUTDOWN* для принудительного завершения серверных процессов.

SUPER

Дает пользователю возможность подключаться к серверу, даже если число сессий пользователя превысило *MAX_CONNECTIONS*. Пользователи с привилегией *SUPER* также могут выполнять такие важные команды, как *CHANGE MASTER*, *KILL*, *MYSQLADMIN DEBUG*, *PURGE [MASTER] LOGS* и *SET GLOBAL*.

TRIGGER

Дает возможность создавать и удалять триггеры на конкретных таблицах. (До версии 5.1.6 для создания и удаления триггеров требовалась привилегия *SUPER*.)

USAGE

Создает пользовательскую учетную запись без каких бы то ни было привилегий.

*ON {[база_данных.]имя_таблицы | * | *.* | база_данных.*}*

Выдает привилегии либо на указанную таблицу, либо на все таблицы в текущей базе данных(*), либо на все таблицы во всех базах данных (*.*), либо на все таблицы в указанной базе данных (*база_данных*.*).

TO получатель [IDENTIFIED BY [PASSWORD] 'пароль'][, ...]

Указывает одного или нескольких пользователей, получающих привилегии. Слово *PASSWORD* необязательно. Можно указать несколько пользователей через запятую. Если привилегии выдаются новому пользователю, то с помощью *IDENTIFIED BY* этому новому пользователю можно сразу назначить пароль.

REQUIRE параметры_безопасности = { NONE | {SSL | X509} [CIPHER 'имя_шифра' [AND]] [ISSUER 'имя_издателя' [AND]] [SUBJECT 'тема'] }

Указывает, должен ли пользователь подключаться с определенными параметрами безопасности:

REQUIRE NONE

Учетная запись не имеет специальных требований по использованию SSL и X509. Это является поведением по умолчанию, если фраза *REQUIRE* не используется.

REQUIRE SSL

Допускает только подключения, защищенные с помощью SSL.

REQUIRE X509

Указывает, что клиент должен иметь действующий сертификат, хотя издатель сертификата и его предмет не имеют значения. При использовании X509 вы можете дополнительно наложить ограничения на издателя сертификата, предмет сертификата и на используемый шифр. Слово *AND* можно использовать между параметрами безопасности.

REQUIRE CIPHER 'имя_шифра'

Гарантирует, что при подключении используется конкретный шифр с определенной длиной ключа.

REQUIRE ISSUER 'имя_издателя'

Разрешает использование сертификатов X509 только с определенным издателем.

REQUIRE SUBJECT 'предмет'

Разрешает подключения только с сертификатами с указанной темой.

WITH опции_with

Позволяет выдать одну или несколько дополнительных привилегий:

GRANT OPTION

Дает получателю привилегии право на передачу полученной привилегии (а точнее, любой имеющейся у пользователя привилегии) другому пользователю.

MAX_QUERIES_PER_HOUR *целое_число*

Ограничивает число запросов пользователя, выполняемых сервером за один час. (Считаются только запросы, результат которых не берется из кэша.) Значение 0 означает отсутствие ограничения на число выполняемых запросов.

MAX_UPDATES_PER_HOUR *целое_число*

Ограничивает число операторов *UPDATE*, выполняемых сервером за один час. Значение 0 означает отсутствие ограничения на число выполняемых обновлений.

MAX_CONNECTIONS_PER_HOUR *целое_число*

Ограничивает число соединений, которые пользователь может открыть за один час. Значение 0 означает, что максимальное число одновременных соединений ограничивается системной переменной *MAX_USER_CONNECTION*.

Так как в MySQL значительное внимание уделяется скорости, то вы можете использовать функции сервера, предназначенные для достижения максимальной производительности. Например, вы можете активировать опцию *SKIP_GRANT_TABLES* для отключения проверки привилегий. Это может ускорить выполнение запросов, однако при этом каждый пользователь имеет полный доступ ко всем ресурсам базы данных.

Для таблиц можно использовать следующие привилегии доступа: *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *CREATE*, *DROP*, *GRANT*, *WITH GRANT OPTION*, *CREATE VIEW*, *SHOW VIEW*, *INDEX*, *TRIGGER* и *ALTER*. Привилегии *INSERT*, *SELECT* и *UPDATE* можно выдавать и отзывать на уровне отдельных столбцов. Например, вы можете выдать разрешение на *SELECT* определенного столбца определенной таблицы.

На уровне хранимых процедур и функций используются привилегии *ALTER ROUTINE*, *EXECUTE* и *WITH GRANT OPTION*. Привилегия *CREATE ROUTINE* не является привилегией уровня отдельной процедуры, потому что эта привилегия нужна для предоставления пользователю принципиальной возможности создавать процедуры и функции.

Некоторые привилегии выдаются только глобально (т. е. с использованием синтаксиса *ON *.**): *FILE*, *PROCESS*, *RELOAD*, *REPLICATION CLIENT*, *REPLICATION SLAVE*, *SHOW DATABASES*, *SHUTDOWN*, *SUPER* и *CREATE USER*.

Имя таблицы, базы данных и столбца может каждое иметь длину до 64 символов, а имя хоста может быть до 60 символов в длину. Пользователь (получатель привилегии) может иметь имя до 16 символов.

MySQL поддерживает также возможность выдачи привилегий для конкретного пользователя определенного хоста, при этом имя пользователя нужно указать в формате *USER@HOST*. Можно использовать групповые символы, чтобы предоставлять привилегии сразу нескольким пользователям. Отсутствующее имя хоста считается равносильным применению символа «%». Рассмотрим следующий пример оператора *GRANT*:

```
GRANT SELECT ON employee TO katie@% IDENTIFIED BY 'audi',
  annalynn IDENTIFIED BY 'cadillac',
  cody@us% IDENTIFIED BY 'bmw';
```

Этот оператор дает разрешение на чтение данных таблицы **employee** пользователю *katie* с любого хоста, пользователю *annalynn* (также с любого хоста, что подразумевается, так как хост не указан) и пользователю *cody* с любого хоста, имя которого начинается на US.

Имя пользователя не должно быть больше 16 символов в длину. После имени пользователя можно использовать *IDENTIFIED BY* для указания пароля. В следующем примере выдаются привилегии трем пользователям с указанием их паролей:

```
GRANT SELECT ON employee TO dylan IDENTIFIED BY 'porsche',
  kelly IDENTIFIED BY 'mercedes',
  emily IDENTIFIED BY 'saab';
```

Если вы выдаете привилегии пользователю, который не существует, то MySQL автоматически создает этого пользователя. Первый из двух следующих операторов создает пользователя с глобальными привилегиями, а второй – пользователя без привилегий:

```
GRANT SELECT ON *.* TO tony IDENTIFIED BY 'humbolt';
GRANT USAGE ON sales.* TO alicia IDENTIFIED BY 'dakota';
```

Если в операторе *GRANT* для еще несуществующего пользователя вы не указали пароль с помощью фразы *IDENTIFIED BY*, то новый пользователь будет создан без пароля. Это весьма небезопасно. В этом случае вы можете (и, скорее всего, вам следует) установить для этого пользователя пароль с помощью оператора *SET PASSWORD*.

В MySQL можно выдавать привилегии для пользователей с удаленных хостов, при этом в имени хоста можно использовать групповой символ %. Например, следующий оператор выдает все привилегии на все объекты пользователю *annalynn* из всех хостов домена **notforprofit.org**:

```
GRANT ALL ON * TO annalynn@'%.notforprofit.org';
```

Привилегии записываются в системные таблицы на четырех уровнях:

Глобальный уровень

Глобальные привилегии относятся ко всем базам данных на текущем сервере и хранятся в системной таблице **mysql.user**.

Уровень баз данных

Привилегии уровня базы данных относятся ко всем объектам базы данных и хранятся в системной таблице **mysql.db**.

Уровень таблиц

Привилегии уровня таблицы относятся ко всем столбцам таблицы и хранятся в системной таблице **mysql.tables_priv**.

Уровень столбцов

Привилегии уровня столбцов относятся к определенным столбцам таблицы и хранятся в системной таблице **mysql.columns_priv**.

Механизм выдачи привилегий на разных уровнях позволяет вам выдать дублирующиеся привилегии на разных уровнях. Например, вы можете выдать пользователю привилегию *SELECT* на конкретную таблицу, а затем выдать глобальную привилегию *SELECT* на ту же таблицу всем пользователям базы данных. Для того чтобы пользователь не мог читать данные из этой таблицы, нужно отдельно отобрать у пользователя привилегии, выданные ему на разных уровнях. MySQL не отбирает никакие привилегии при удалении таблицы, однако при удалении процедуры или функции соответствующие привилегии отзываются.

Системные таблицы не защищены каким-то особенным образом, поэтому можно изменять привилегии, используя операторы *INSERT*, *UPDATE* и *DELETE* вместо *GRANT* и *REVOKE*. Обычно информация о привилегиях считывается в память при запуске MySQL. Привилегии на уровне баз данных, таблиц и столбцов вступают в действие сразу после выполнения оператора *GRANT*. Пользовательские привилегии вступают в силу при следующем подключении пользователя. Если же вы напрямую модифицируете привилегии в системных таблицах, то эти изменения вступят в силу либо при следующем запуске MySQL, либо после выполнения команды *FLUSH PRIVILEGES*.

Если вы хотите использовать хорошо защищенные соединения, то вам поможет фраза *REQUIRE*. Вам не требуется использовать все возможности фразы *REQUIRE* для простой защиты соединения с помощью SSL или X509. Однако, если у вас есть особенные требования, то вы можете использовать дополнительные настройки. В следующем примере мы даем привилегии пользователю *Tony* локального сервера MySQL и требуем для подключения сертификат X509 с определенным издателем, шифром и темой:

```
GRANT SELECT ON *.* TO 'tony'@'localhost'
  IDENTIFIED BY 'humbolt'
  REQUIRE SUBJECT '/C=EE/ST=CA /L=Frisco
    /O=MySQL demo client certificate
    /CN=Tony  Tubs/Email=tont@myorg.com'
  AND ISSUER '/C=FI/ST=CA /L=UC /O=MySQL
    /CN=Tony Tubs/Email=tont@myorg.com'
  AND CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

Oracle

В Oracle оператор *GRANT* поддерживает огромное число вариаций и сочетаний. Синтаксис оператора следующий:

```
GRANT { [объектная_привилегия][, ...] |
  [системная_привилегия][, ...] | [роль][, ...]}
[ ON { [схема.][объект][, ...] |
[DIRECTORY каталог] |
[JAVA [ { SOURCE | RESOURCE } ] [схема.][объект]] } ]
TO {получатель[, ...] | роль[, ...] | PUBLIC}
[WITH { GRANT | HIERARCHY } OPTION]
[IDENTIFIED BY пароль] [WITH ADMIN OPTION];
```

Одним оператором вы можете выдать несколько привилегий, но все они должны быть одного типа (объектной, системной привилегией или ролью).

Например, вы могли бы одним оператором *GRANT* выдать пользователю три привилегии на определенную таблицу, отдельным оператором выдать определенной роли две системные привилегии, а затем выдать несколько ролей пользователю, но вы не можете сделать это все одним оператором.

Оператор *GRANT* имеет следующие параметры:

GRANT объектная_привилегия

Используется для предоставления пользователю или роли привилегий на определенный объект схемы (например, таблицу или представление). В одном операторе вы можете комбинировать несколько привилегий, объектов и получателей. Однако вы не можете в одном операторе с объектными привилегиями выдавать системные привилегии или роли. Возможны следующие объектные привилегии:

ALL [PRIVILEGES]

Дает все возможные привилегии на указанный объект схемы. Эта опция допустима для таблиц.

ALTER

Дает привилегии на изменение существующего объекта с помощью оператора *ALTER*. Эту опцию можно использовать с таблицами и последовательностями.

DEBUG

Дает возможность использовать таблицу в отладчике. Выданные привилегии относятся к триггерам таблицы и информации SQL, напрямую относящейся к таблице. Можно использовать с таблицами, представлениями, процедурами, функциями, пакетами, объектами Java и типами.

EXECUTE

Дает привилегии на выполнение хранимой процедуры, пользовательской функции, библиотеки, пакета, индексного типа, пользовательского оператора, объекта Java.

INDEX

Дает привилегии на создание индекса по таблице.

{ON COMMIT REFRESH | QUERY REWRITE }

Дает привилегии на создание на базе определенной таблицы материализованных представлений, обновляемых после каждой транзакции или используемых для переписывания запросов. Используется только для материализованных представлений.

QUERY REWRITE

Дает привилегии на создание на базе определенной таблицы материализованных представлений, используемых для переписывания запросов. Используется только для материализованных представлений.

{READ | WRITE }

Дает привилегии на чтение и запись файлов в указанном каталоге. Так как Oracle позволяет хранить данные в файлах вне базы данных, то серверный процесс должен выполняться от имени пользователя, имеющего доступ к соответствующим каталогам. Вы можете использовать специаль-

ный механизм для разграничения прав доступа на уровне пользователей. Имейте в виду, что *WRITE* используется только вместе с внешними таблицами, такими как журнальный файл или файл ошибок.

REFERENCES

Дает привилегии на создание ограничений ссылочной целостности. Используется только для таблиц.

{SELECT | INSERT | DELETE | UPDATE }

Дает привилегии на выполнение соответствующего оператора по отношению к определенному объекту схемы. Может применяться для таблиц и представлений, а также для последовательностей и материализованных представлений (только *SELECT*). Обратите внимание, что если пользователю или роли требуется привилегия *DELETE*, то ей также нужно выдать и привилегию *SELECT*. Детализировать на уровне отдельных столбцов (столбцы перечисляются в скобках через запятую) можно только привилегии *INSERT*, *REFERENCES* и *UPDATE* для таблиц и представлений.

UNDER

Дает привилегии на создание дочернего представления для указанного представления. Используется только для типизированных представлений.

GRANT системная_привилегия

Выдает указанную системную привилегию пользователю или роли. Например, вы можете выдать пользователю привилегию *CREATE TRIGGER* или *ALTER USER*: в каждом из этих случаев пользователь получает возможность выполнять одноименные операторы. Полный список системных привилегий Oracle приводится далее в этом разделе в табл. 3.2.

роль

Назначает указанную роль пользователю или другой роли. В дополнение к пользовательским ролям существует несколько предопределенных системных ролей:

CONNECT, RESOURCE, DBA

Обеспечивают обратную совместимость с предыдущими версиями Oracle. Не используйте эти роли, так как в будущем они могут быть удалены.

DELETE_CATALOG_ROLE, EXECUTE_CATALOG_ROLE, SELECT_CATALOG_ROLE

Дает члену роли возможность читать (*SELECT*) и удалять (*DELETE*) данные из словаря данных и выполнять (*EXECUTE*) пакеты словаря данных.

EXP_FULL_DATABASE, IMP_FULL_DATABASE

Дает членам этих ролей разрешение на использование утилит импорта (*IMP*) и экспорта (*EXP*).

AQ_USER_ROLE, AC_ADMINISTRATOR_ROLE

Дает членам этих ролей разрешение на использование (*USER*) и администрирование (*ADMINISTRATOR*) опции Advanced Queueing.

SNMPAGENT

Роль выдается только Oracle Enterprise Manager и Intelligent Agent.

RECOVERY_CATALOG_OWNER

Дает привилегию на создание пользователей со своими собственными каталогами восстановления.

HS_ADMIN_ROLE

Дает доступ к областям словаря данных, используемым для поддержки гетерогенных сервисов Oracle.

ON имя_схемы

Дает пользователю или роли привилегии на объект схемы. Объекты базы данных включают таблицы, представления, последовательности, хранимые процедуры, пользовательские функции, пакеты, материализованные представления, пользовательские типы, пакеты, индексные типы, пользовательские операторы и синонимы для всех этих объектов. Если вы не указываете имя схемы, то Oracle использует текущую схему пользователя. Поддерживаются два дополнительных ключевых слова для специальных случаев:

DIRECTORY

Предоставляет привилегии на каталог, являющийся специальным объектом базы данных, соответствующим каталогу файловой системы.

JAVA

Дает привилегии на Java-объекты *SOURCE* и *RESOURCE*, *являющиеся частью схемы пользователя*.

TO {получатель | роль | PUBLIC} [WITH {GRANT|HIERARCHY } OPTION]

Указывает пользователя или роль, получающую указанную привилегию. Ключевое слово *PUBLIC* можно использовать для выдачи и отзыва привилегий у роли *PUBLIC*. Через запятую можно перечислить несколько получателей.

WITH GRANT OPTION

Позволяет получателю объектных привилегий передавать их другим пользователям или роли *PUBLIC*, но не любым другим ролям.[1]

WITH HIERARCHY OPTION

При получении привилегий на объект с помощью этой опции выдаются также привилегии на все подчиненные объекты, включая все объекты, которые будут созданы в будущем. Вы можете использовать эту опцию только при выдаче привилегии *SELECT*.

IDENTIFIED BY пароль

Устанавливает или меняет пароль, который используется для активации полученной пользователем роли.[2]

---

[1] Позволяет получателю передавать объектные привилегии другим пользователям и ролям. Опция WITH GRANT OPTION может быть применена, только когда получатель объектной привилегии – пользователь или роль PUBLIC, но не какая-то другая роль. – *Прим. науч. ред.*

[2] В данном случае эта фраза применяется для задания пароля пользователя, которому передаются привилегии. – *Прим. науч. ред.*

WITH ADMIN OPTION

Позволяет получателю привилегии или роли назначать и отзывать их у других пользователей и ролей (кроме роли *GLOBAL*), а также удалять роль или изменять авторизацию, требующуюся для доступа к роли.

Выданные пользователю привилегии вступают в силу сразу же. Аналогичным образом сразу же начинают действовать привилегии, выданные роли, если роль активирована. В противном случае привилегии роли начинают действовать при активации роли. Обратите внимание, что вы можете назначать роль как пользователям, так и другим ролям, включая *PUBLIC*. Например:

```
GRANT sales_reader TO sales_manager;
```

Для выдачи привилегий на представление вы должны иметь привилегии с опцией *WITH GRANT OPTION* на все базовые таблицы представления.

Если вы хотите, чтобы привилегия была доступна всем пользователям, то выдайте ее роли *PUBLIC*:

```
GRANT SELECT ON work_schedule TO public;
```

Однако есть некоторые ограничения на выдачу системных привилегий и ролей:

- Привилегия или роль не может быть указана в одном операторе *GRANT* более одного раза.
- Роли не могут быть назначены сами себе.
- Роли не могут быть назначены рекурсивным образом. Например, если вы назначили роль *sales_reader* роли *sales_manager*, то вы не можете назначить роль *sales_manager* роли *sales_reader*.

Вы можете выдать несколько привилегий одним оператором при условии, что все эти привилегии одного типа:

```
GRANT UPDATE (emp_id, job_id), REFERENCES (emp_id),
ON employees
TO sales_manager;
```

Кстати, любые объектные привилегии на таблицу позволяют блокировать эту таблицу в любом режиме при помощи оператора *LOCK TABLE*.

В недавних версиях Oracle была добавлена поддержка специальных объектов для бизнес-анализа. Хотя бизнес-анализ выходит за рамки рассматриваемого в этой книге, заметим, что привилегии на эти специальные объекты выдаются таким же образом, как и на любые другие типы объектов. Объекты для бизнес-анализа включают в себя *MINING MODEL*, *CUBE*, *CUBE MEASURE FOLDER*, *CUBE DIMENSIONS* и *CUBE BUILD PROCESS*. Oracle также поддерживает очереди сообщений. Синтаксис выдачи привилегий для использования этой функциональности такой же, как и для объектов, и состоит из ролей *AQ_USER_ROLE* и *AQ_ADMINISTRATOR_ROLE*. Дополнительную информацию по бизнес-анализу и очередям сообщений в Oracle ищите в документации.

Практически на любой оператор или любую опцию в Oracle можно выдать привилегию оператором *GRANT* (табл. 3.2). Привилегии можно выдавать не только на объекты базы данных (такие как таблицы или представления) и системные

команды (такие как *CREATE ANY TABLE*), но и на объекты схемы (такие как *DIRECTORY*, *JAVA SOURCE* и *RESOURCE*). Ключевое слово *ANY* (*например*, *CREATE ANY TABLE*) означает, что привилегия относится к объектам указанного типа, которыми владеют любые пользователи схемы. Без *ANY* (*например*, *CREATE TABLE*) привилегия относится только к объектам, которыми владеет сам пользователь.

Таблица 3.2. Системные привилегии Oracle

Привилегия	Описание
CREATE | ALTER | DROP	
CREATE CLUSTER	Позволяет пользователю создавать кластеры в его собственной схеме.
{CREATE | ALTER | DROP} ANY CLUSTER	Позволяет создавать, изменять или удалять, соответственно, кластеры в любой схеме.
{CREATE | DROP} ANY CONTEXT	Позволяет создавать или удалять, соответственно, любой контекст.
CREATE DIMENSION	Позволяет пользователю создавать измерения в своей схеме.
{CREATE | ALTER | DROP} ANY DIMENSION	Позволяет пользователю создавать, изменять или удалять измерения в любой схеме.
{CREATE | DROP} ANY DIRECTORY	Позволяет создавать и удалять объекты-каталоги.
{CREATE | ALTER | DROP | EXECUTE[a]} [ANY] INDEX	Позволяет пользователю создавать, изменять, удалять или использовать индекс в своей или любой (*ANY*) схеме.
CREATE INDEXTYPE	Позволяет пользователю создавать индексный тип в своей схеме.
{CREATE | DROP | EXECUTE} ANY INDEXTYPE	Позволяет создавать, удалять или использовать индексный тип в любой схеме.
CREATE DATABASE LINK	Позволяет создавать указатели на базы данных в схеме пользователя.
CREATE EXTERNAL JOB	Позволяет создавать в своей схеме задачи, исполняемые по расписанию в операционной системе.[b]
CREATE [ANY] JOB	Позволяет создавать, изменять или удалять задачи, расписания и программы в своей схеме или в любой схеме (*ANY*). Выдавайте эту привилегию с осторожностью, так как она позволяет выполнять любой код, как если бы это делала внешняя программа.
{CREATE | DROP} [ANY] LIBRARY	Позволяет создавать или удалять внешние библиотеки процедур и функций в своей или любой (*ANY*) схеме.

[a] Привилегии *EXECUTE [ANY] INDEX* не существует. – *Прим. науч. ред.*

[b] Привилегия позволяет создавать задачи, запускающие внешнее приложение или скрипт ОС. – *Прим. науч. ред.*

Привилегия	Описание
{CREATE | ALTER | DROP} [ANY] MATERIALIZED VIEW	Дает привилегии на создание, изменение или удаление материализованных представлений в своей или любой (*ANY*) схеме.
CREATE OPERATOR	Дает привилегии на создание оператора и его привязок в схеме пользователя.
{CREATE | DROP | ALTER | EXECUTE} ANY OPERATOR	Дает привилегии на создание оператора и его привязок в любой схеме.
CREATE PUBLIC DATABASE LINK	Дает права на создание публичных указателей на базу данных.
{CREATE | ALTER | DROP} [ANY] ROLE	Дает привилегии на создание (не используйте *ANY* вместе с *CREATE*), изменение и удаление ролей.
{CREATE | DROP | ALTER} ANY OUTLINE	Дает привилегии на создание, удаление и изменение любого хранимого плана выполнения запроса, который может быть использован в любой схеме, использующей хранимые планы выполнения запросов.
SELECT ANY OUTLINE	Дает привилегии на создание приватных хранимых планов выполнения запросов копированием публичных.
{CREATE | ALTER | DROP | EXECUTE} [ANY] PROCEDURE	Дает привилегии на создание, изменение, удаление и выполнение хранимых процедур и функций в любой схеме (*ANY*) либо в схеме пользователя.
{CREATE | ALTER | DROP} PROFILE	Дает привилегии на создание, изменение и удаление профилей пользователей.
ALTER RESOURCE COST	Позволяет устанавливать стоимости ресурсов сессии.
{CREATE | ALTER | DROP | GRANT} ANY ROLE	Дает привилегии на создание, изменение, удаление и назначение ролей в базе данных.
{CREATE | ALTER | DROP} ROLLBACK SEGMENT	Дает привилегии на создание, изменение и удаление сегментов отката.
{CREATE | ALTER | DROP | SELECT} [ANY] SEQUENCE	Дает привилегии на создание, изменение, удаление и использование последовательностей.
CREATE SNAPSHOT	Дает привилегии на создание моментальных снимков (материализованных представлений) в схеме пользователя.
{CREATE | ALTER | DROP} ANY SNAPSHOT	Дает привилегии на создание, изменение и удаление мгновенных снимков в любой схеме.
CREATE SYNONYM	Дает привилегии на создание синонимов в схеме пользователя.
{CREATE | DROP} ANY SYNONYM	Дает привилегии на создание и удаление синонимов в любой схеме.
{CREATE | DROP} PUBLIC SYNONYM	Дает привилегии на создание и удаление публичных синонимов.

Таблица 3.2 (продолжение)

Привилегия	Описание
Выполнение	
EXECUTE ANY PROGRAM	Дает привилегии на выполнение любой программы (планировщика) из схемы пользователя в заданиях.
EXECUTE ANY CLASS	Дает привилегии на выполнение любого класса заданий в схеме пользователя.
Сессии	
CREATE SESSION	Дает привилегии на подключение к базе данных.
ALTER SESSION	Дает привилегии на выполнение оператора *ALTER SESSION*.
RESTRICTED SESSION	Дает пользователю возможность подключаться к экземпляру, запущенному в режиме *STARTUP RESTRICT*.
DEBUG CONNECT SESSION	Дает привилегии на подключение сессии к отладчику Java Debug Wire Protocol (JDWP).
DEBUG ANY PROCEDURE	Дает привилегии на отладку любых PL/SQL объектов базы данных.
DROP PUBLIC DATABASE LINK	Дает привилегии на удаление публичных указателей (dblink) на базы данных.
FLASHBACK ARCHIVE ADMINISTRATOR	Дает привилегии на создание, изменение и удаление любых flashback data archive.
GRANT ANY ROLE	Дает привилегии на назначение любой роли любому пользователю.
MANAGE SCHEDULER	Дает все привилегии на задания (jobs): создание, изменение и удаление заданий любого класса (права на создание, изменение, удаление любых классов заданий, окон и групп окон).
ON COMMIT REFRESH	Дает привилегии на создание материализованных представлений, обновляемых после фиксации транзакции, а также на преобразование материализованных представлений, обновляемых по требованию в обновляемые после фиксации транзакции.
Таблицы и табличные пространства	
FLASHBACK ANY TABLE	Дает привилегии на выполнение ретроспективных запросов по любой таблице, представлению и материализованному представлению в любой схеме. Эта роль не требуется для использования процедур пакета *DBMS_FLASHBACK*.
CREATE ANY TABLE	Дает привилегии на создание таблиц в любой схеме. Владелец схемы, в которой создается таблица, должен иметь достаточное свободное пространство в соответствующем табличном пространстве для хранения таблицы.
ALTER ANY TABLE	Дает привилегии на изменения любой таблицы или представления в любой схеме.

Привилегия	Описание
BACKUP ANY TABLE	Дает привилегии на использование утилиты экспорта для инкрементального экспорта любого объекта в любой схеме.
DELETE ANY TABLE	Дает привилегии на удаление строк из таблиц, секций таблиц и представлений в любой схеме.
DROP ANY TABLE	Дает привилегии на удаление и очистку таблиц или секций таблиц в любой схеме.
INSERT ANY TABLE	Дает привилегии на вставку строк в таблицы и представления в любой схеме.
LOCK ANY TABLE	Дает привилегии на блокирование таблиц и представлений в любой схеме.
UPDATE ANY TABLE	Дает привилегии на обновление строк в таблицах и представлениях в любой схеме.
SELECT ANY TABLE	Дает привилегии на выполнение запросов к таблицам, представлениям и материализованным представлениям в любой схеме.
{CREATE | ALTER | DROP} TABLESPACE	Дает привилегии на создание, изменение и удаление табличных пространств.
MANAGE TABLESPACE	Дает привилегии на перевод табличных пространств в активный или неактивный режим, а также на запуск и завершение процессов резервного копирования табличных пространств.
UNLIMITED TABLESPACE	Дает неограниченные привилегии на использование места в любых табличных пространствах. Эта привилегия отменяет любые квоты на отдельные табличные пространства. Если вы отзываете эту привилегию у пользователя, то все объекты пользователя сохраняются, но дальнейшее использование табличных пространств определяется квотами. Вы не можете выдавать эту системную привилегию ролям.
Триггеры	
ADMINISTER DATABASE TRIGGER	Дает привилегии на создание триггеров на уровне базы данных. (Пользователь также должен иметь привилегию *CREATE TRIGGER* или *CREATE ANY TRIGGER*.)
{CREATE | ALTER | DROP} [ANY] TRIGGER	Дает привилегии на создание, изменение и удаление триггеров в своей или в любой (*ANY*) схеме.
Типы	
{CREATE | ALTER | DROP} [ANY] TYPE	Дает привилегии на создание, изменение и удаление объектных типов и их тел в своей или любой (*ANY*) схеме.
EXECUTE ANY TYPE	Если привилегия выдана пользователю, то ему разрешается использовать объектные типы и типы-коллекции, а также вызывать методы объектных типов в любой схеме. Если привилегия выдана роли, то отдельные участники этой роли не смогут вызывать методы объектных типов в любой схеме.

Таблица 3.2 (продолжение)

Привилегия	Описание
Пользователи	
CREATE USER	Дает привилегии на создание пользователей, определение квот на любые табличные пространства, установку временного табличного пространства и табличного пространства, используемого по умолчанию, а также на указание профиля пользователя.
ALTER USER	Дает привилегии на изменение пользователя, а именно: изменение пароля и способа аутентификации, определение квот на любые табличные пространства, установку временного табличного пространства и табличного пространства, используемого по умолчанию, а также указание профиля пользователя.
BECOME USER	Дает привилегии на подключение от имени другого пользователя (требуется для выполнения полного импорта базы данных).
DROP USER	Дает привилегии на удаление пользователей.
UNDER ANY TYPE	Дает привилегии на создание подтипа на базе любого типа.
Представления	
{CREATE | DROP} [ANY] VIEW	Дает привилегии на создание и удаление представлений в своей или любой (*ANY*) схеме.
UNDER ANY VIEW	Дает привилегии на создание дочерних представлений на базе любого представления.
MERGE ANY VIEW	Дает привилегии на использование оптимизатором слияния представлений для ускорения выполнения запросов пользователя.
Дополнительные привилегии	
ADMINISTER [ANY] SQL TUNING SET	Дает привилегии на управление с помощью пакета *DBMS_SQLTUNE* наборами оптимизации sql-запросов – собственными или принадлежащими любым пользователям (*ANY*).
ADVISOR	Дает привилегии на использование инфраструктуры помощников из PL/SQL.
ALTER DATABASE	Дает привилегии на изменение базы данных.
ALTER SYSTEM	Дает привилегии на выполнение оператора *ALTER SYSTEM*.
ANALYZE ANY	Дает привилегии на сбор статистики по таблице, кластеру или индексу в любой схеме.
ANALYZE ANY DICTIONARY	Дает привилегии на сбор статистики по любому объекту словаря данных.
AUDIT ANY	Дает привилегии на выполнение аудита любого объекта базы данных при помощи оператора *AUDIT*.
AUDIT SYSTEM	Дает привилегии на выполнение операторов *AUDIT*.
{DELETE | EXECUTE | SELECT} CATALOG_ROLE	Дает привилегии на доступ к представлениям и пакетам словаря данных.

Привилегия	Описание
CHANGE NOTIFICATION	Дает привилегии на регистрацию и получение уведомлений о запросах и изменениях в базе данных.
COMMENT ANY TABLE	Дает привилегии на создание комментариев к таблицам, столбцам и представлениям в любой схеме.
EXEMPT ACCESS POLICY	Позволяет обходить детальный контроль доступа (fine-grained audit). Использовать эту привилегию следует аккуратно, так как она позволяет обходить политики безопасности, определенные для приложений.
FORCE ANY TRANSACTION	Позволяет принудительно зафиксировать или откатить любую сомнительную (in-doubt) распределенную транзакцию в локальной базе данных и завершить распределенную транзакцию.
FORCE TRANSACTION	Позволяет принудительно зафиксировать или откатить собственную сомнительную (in-doubt) распределенную транзакцию в локальной базе данных.
{EXP|IMP} FULL_DATABASE	Дает привилегии на использование утилит импорта (*IMP*) и экспорта (*EXP*).[a]
GRANT ANY PRIVILEGE	Дает привилегии на выдачу любых системных привилегий.
GRANT ANY OBJECT PRIVILEGE	Дает привилегии на выдачу любых объектных привилегий.
RECOVERY_CATALOG_OWNER	Дает привилегии на создание каталога восстановления.
RESUMABLE	Дает возможность использовать функцию Resumable Space Allocation.
SELECT ANY DICTIONARY	Дает привилегии на чтение любого объекта словаря данных в схеме **SYS**.
SELECT ANY TRANSACTION	Дает привилегии на доступ к представлению *FLASHBACK_TRANSACTION_QUERY*. (Выдавайте эту привилегию с осторожностью, так как она позволяет читать любые данные в базе данных.)
SYSDBA	Дает пользователю привилегию *RESTRICTED SESSION* и позволяет выполнять операции *STARTUP* и *SHUTDOWN*, *CREATE* и *ALTER DATABASE*, *OPEN/MOUNT/BACKUP*, *ARCHIVELOG* и *RECOVERY*, менять кодировку базы данных.
SYSOPER	Дает пользователю привилегию *RESTRICTED SESSION* и позволяет выполнять операции *STARTUP* и *SHUTDOWN*, *ALTER DATABASE*, *OPEN/MOUNT/BACKUP*, *ARCHIVELOG* и *RECOVERY*.
QUERY REWRITE	Разрешает переписывание запросов с использованием материализованных представлений и создание для таблиц индексов, основанных на функциях, если эти материализованные представления и таблицы принадлежат пользователю.
GLOBAL QUERY REWRITE	Разрешает переписывание запросов и создание индексов, основанных на функциях.

[a] Для всей базы данных. – *Прим. науч. ред.*

Любая из перечисленных выше привилегий, содержащая слово *ANY*, позволяет выполнять соответствующий оператор или команду для объектов любой схемы. Если вы хотите исключить системную схему **SYS** из области действия привилегий ANY, то установите для инициализационного параметра *O7_DICTIONARY_ACCESSIBILITY* значение *FALSE (значение по умолчанию)*.

PostgreSQL

В PostgreSQL поддерживается подмножество возможностей оператора *GRANT* по сравнению со стандартом ANSI, включая поддержку объектных привилегий для таблиц, последовательностей, функций и т. д. Синтаксис оператора *GRANT* в PostgreSQL следующий:

```
GRANT { роль | { ALL [PRIVILEGES] | SELECT | INSERT |
  DELETE | UPDATE | RULE | REFERENCES| TRIGGERS | CREATE |
  USAGE }[, ...] }
ON { [TABLE | SEQUENCE | DATABASE | FUNCTION | LANGUAGE |
  SCHEMA | TABLESPACE]
  объект[, ...] }
[WITH GRANT OPTION]
TO {получатель | GROUP группа | PUBLIC}[, ...]
[WITH ADMIN OPTION]
```

где:

GRANT роль

Назначает пользователю предварительно созданную роль. Используется синтаксис *GRANT роль TO получатель [WITH ADMIN OPTION]*. Пользователь получает все привилегии, выданные роли. Через роли нельзя выдавать привилегии роли *PUBLIC*.

ALL [PRIVILEGES]

Выдает все привилегии, которые имеет право выдать пользователь, выполняющий оператор *GRANT*. Использование *ALL* – нерекомендуемый подход, так как он провоцирует небрежное отношение к вопросам безопасности.

CREATE

Дает пользователю привилегии на создание объекта определенного типа. Можно использовать совместно с *ON DATABASE*, *ON SCHEMA* и *ON TABLESPACE*. *ON DATABASE* позволяет создавать новые схемы в базе данных. *ON SCHEMA* позволяет создавать и переименовывать объекты в схеме. *ON TABLESPACE* позволяет создавать таблицы и индексы в указанном табличном пространстве.

CONNECT | TEMP[ORARY]

Позволяет подключать к указанной базе данных или создавать в ней временные таблицы.

SELECT | INSERT | DELETE | UPDATE

Дает привилегии на выполнение соответствующего оператора для определенного объекта базы данных, такого как таблица или представление. При желании вы можете в скобках указать список столбцов, к которым относятся указанные привилегии.

RULE

Дает привилегии на создание и удаление правил на таблицах и представлениях.

REFERENCES

Дает привилегии на создание и удаление внешних ключей, ссылающихся на указанную таблицу.

TRIGGERS

Дает привилегии на создание триггеров на определенные таблицы и их столбцы.

USAGE

Дает привилегии на использование следующих объектов: CHARACTER *SET*, *COLLATION*, *TRANSLATION* и *DOMAIN* (пользовательский тип данных). Также вы можете выдать привилегию *USAGE* на последовательности, языки и схемы.

ON { [TABLE] | SEQUENCE | DATABASE | FUNCTION | LANGUAGE | SCHEMA | TABLESPACE имя_объекта[, ...] }

Указывает объект, на который выдаются привилегии. Ключевое слово *TABLE* является значением по умолчанию. Вы можете выдать привилегии на несколько объектов одного типа, перечислив их через запятую.

WITH GRANT OPTION

Позволяет пользователю передавать полученные привилегии другим пользователям.

TO {получатель | GROUP группа | PUBLIC}[, ...]

Указывает пользователя, получающего привилегии. В PostgreSQL *PUBLIC* является синонимом для всех пользователей. Привилегии можно выдавать группе (то же, что и роль), указав ее имя.

WITH ADMIN OPTION

Дает участнику роли возможность назначать и отзывать эту роль у других пользователей.

Одним оператором можно выдать несколько привилегий, перечислив их через запятую. Но при этом требуется, чтобы все привилегии были одного типа. Нельзя сочетать *ALL PRIVILEGES* с другими привилегиями, но большинство остальных привилегий можно комбинировать произвольным образом.

В PostgreSQL не поддерживается фраза *WITH* и разрешения на уровне столбцов. Создатель объекта имеет все права на него. У создателя объекта можно отозвать практически все привилегии на объект, кроме права на раздачу привилегий. Аналогично, у создателя объекта нельзя отозвать привилегию на удаление этого объекта. Другие пользователи не имеют никаких привилегий на объект, пока эти привилегии не будут им явно выданы.

В зависимости от типа объекта, на объект могут быть выданы различные привилегии:

TABLE

Могут быть выданы привилегии *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *REFERENCES*, *TRIGGER* и *ALL [PRIVILEGES]*.

SEQUENCE

Могут быть выданы привилегии *USAGE*, *SELECT* и *ALL [PRIVILEGES]*.

DATABASE

Могут быть выданы привилегии *CONNECT*, *CREATE*, *TEMP[ORARY]* и *ALL [PRIVILEGES]*.

FUNCTION

Могут быть выданы привилегии *EXECUTE* и *ALL [PRIVILEGES]*.

LANGUAGE

Могут быть выданы привилегии *USAGE* и *ALL [PRIVILEGES]*.

SCHEMA

Могут быть выданы привилегии *CREATE*, *USAGE* и *ALL [PRIVILEGES]*.

TABLESPACE

Могут быть выданы привилегии *CREATE* и *ALL [PRIVILEGES]*.

Поддержка оператора *GRANT* в PostgreSQL достаточно слабая. В следующем примере привилегия *INSERT* на таблицу **publishers** выдается роли *PUBLIC*, а привилегии *SELECT* и *UPDATE* на таблицу **sales** выдаются пользователям *Emily* и *Dylan*:

```
GRANT INSERT ON TABLE publishers TO PUBLIC;
GRANT SELECT, UPDATE ON sales TO emily, dylan;
```

В следующем примере пользователю *Katie* предоставляется привилегия управления базой данных, а затем *Katie* и *Anna* добавляются в новую группу:

```
GRANT ALL ON DATABASE publishing TO katie;
GRANT manager_role ON katie, anna WITH ADMIN OPTION;
```

SQL Server

Реализация оператора *GRANT* в SQL Server несколько отличается от стандарта ANSI. SQL Server не поддерживает фразу *FROM* и опции *HIERARCHY* и *ADMIN*, однако поддерживает выдачу специальных системных привилегий и выдачу привилегий из-под контекста безопасности другого пользователя. Синтаксис оператора следующий:

```
GRANT {[объектная_привилегия][,...]|[системная_привилегия] }
[ON { [область_определения::[[схема.]объект]
  [(столбец[, ...])] }]
TO { получатель[, ...] | роль[, ...] | PUBLIC | GUEST }
[WITH GRANT OPTION]
[AS {группа | роль}]
```

где:

GRANT объектная_привилегия

Выдает на объекты различные привилегии, которые можно комбинировать произвольным образом (за исключением привилегии *ALL PRIVILEGE*). Объ-

ектные привилегии можно выдавать на таблицы, представления, функции (табличные, скалярные и агрегатные), процедуры (хранимые и расширенные), очереди обработки и синонимы. Привилегии включают:

ALL [PRIVILEGES]

Выдает все привилегии, которые есть на текущий момент у пользователя, выполняющего *GRANT*. Этот способ выдачи привилегий использовать не рекомендуется, так как он способствует небрежному обращению с привилегиями. *ALL* может использоваться только членами ролей *SYSADMIN* и *DB_OWNER*, либо владельцем объекта. В SQL Server 2008 привилегия *ALL* запрещена к использованию и включена только для обратной совместимости. *GRANT ALL* на текущий момент является короткой записью для следующих привилегий:

GRANT ALL для баз данных

Выдает привилегии *BACKUP DATABASE*, *BACKUP LOG*, *CREATE DATABASE*, *CREATE DEFAULT*, *CREATE FUNCTION*, *CREATE PROCEDURE*, *CREATE RULE*, *CREATE TABLE* и *CREATE VIEW*.

GRANT ALL для функций

Выдает привилегии *EXECUTE* и *REFERENCES* для скалярных функций и *DELETE*, *INSERT*, *REFERENCES*, *SELECT* и *UPDATE* для табличных функций.

GRANT ALL для процедур (включает хранимые и расширенные хранимые процедуры)

Выдает привилегию *EXECUTE*.

GRANT ALL для таблиц и представлений

Выдает привилегии *DELETE*, *INSERT*, *REFERENCES*, *SELECT* и *UPDATE*.

{SELECT | INSERT | DELETE | UPDATE}

Дает привилегии на выполнение соответствующего оператора для определенного объекта базы данных, такого как таблица или представление. Можно указать в скобках список столбцов, для которых будет действовать указанная привилегия.

REFERENCES

Дает привилегии на создание и удаление внешних ключей, ссылающихся на определенный объект базы данных как на родительский.

EXECUTE

Дает привилегии на выполнение хранимой и расширенной процедуры или пользовательской функции.

GRANT системная_привилегия

Дает привилегии на выполнение соответствующих операторов (*EXECUTE*) или чтение представлений в схеме **SYS** (*SELECT*). Дополнительная информация приводится в табл. 3.3.

Привилегия на выполнение оператора *CREATE* также подразумевает, что выдается разрешение на соответствующие операторы *ALTER* и *DROP*. Однако возможны ситуации, когда наличие одной привилегии не означает наличие других необходимых привилегий. Например, привилегия *EXECUTE* на

хранимую процедуру sp_*addlinkedserver* не позволяет пользователю создавать присоединенные серверы, если только пользователь не включен также в роль *sysadmin*.

ON [область_определения::][[схема.]объект] [(столбец[, ...])]

Указывает объект, на который выдаются привилегии. Эта фраза не используется при выдаче системных привилегий. *Область_определения* определяет тип объекта, на который выдаются привилегии. Это может быть таблица, представление, *APPLICATION ROLE*, *ASSEMBLY*, *ASYMMETRIC KEY*, *CERTIFICATE*, *ENDPOINT*, *FULLTEXT CATALOG*, *LOGIN*, *ROLE*, *SCHEMA*, объекты брокера служб (*CONTRACT*, *MESSAGE TYPE*, *REMOTE SERVICE BINDING*, *ROUTE* и *SERVICE*), *SYMMETRIC KEY*, *TYPE*, *USER* и *XML SCHEMA COLLECTION*. При выдаче привилегий на таблицы и представления вы можете указать список столбцов, на которые распространяются привилегии. На таблицу и представление вы можете выдать привилегии *SELECT*, *INSERT*, *UPDATE*, *DELETE* и *REFERENCES*, но на столбцы таблицы, представления и пользовательской функции вы можете выдать только привилегии *SELECT* и *UPDATE*. На хранимые процедуры можно выдавать привилегии *EXECUTE*, и *REFERENCES* на пользовательские функции. Привилегии *REFERENCES* также требуются для функций и представлений, созданных с параметром *WITH SCHEMABINDING*.

TO {получатель[, ...] | роль[, ...] | PUBLIC | GUEST}

Указывает пользователя или роль, получающую привилегии. Через запятую можно перечислить несколько получателей привилегий. Используйте ключевое слово *PUBLIC* для выдачи привилегий роли *PUBLIC*, включающей в себя всех пользователей. SQL Server также поддерживает роль *GUEST*, соответствующую всем пользователям, не имеющим никаких других ролей. Так как SQL Server поддерживает две модели безопасности (на основе базы данных либо на основе Windows), то получателем привилегий может быть пользователь базы данных, пользователь Windows, группа Windows или роль SQL Server.

WITH GRANT OPTION

Позволяет пользователю передавать полученные привилегии другим пользователям. Эта опция используется только с объектными привилегиями.

AS {группа | роль}

Указывает альтернативного пользователя или группу, имеющую право на выполнение оператора GRANT. Вы можете использовать фразу *AS* для выдачи привилегий, как если бы сессия, выдающая привилегии, была бы членом другой роли или группы.

Привилегии могут быть выданы только в текущей базе данных, поэтому нельзя выдать привилегии сразу в нескольких базах данных.

Модель безопасности в SQL Server отличается от других описанных в книге платформ (и от стандарта ANSI). В соответствии со стандартом ANSI SQL Server поддерживает оператор *GRANT* для выдачи привилегий пользователям и ролям, а также оператор *REVOKE* для отзыва привилегий. Однако в SQL Server этот список пополнен

также оператором *DENY*, использующим тот же синтаксис, что и *REVOKE*.

Оператор *DENY* позволяет администратору базы данных запретить пользователю или роли получение определенных привилегий. Любые привилегии, запрещенные оператором *DENY*, должны быть отозваны оператором *REVOKE*, перед тем как их можно будет выдать пользователю. *DENY* имеет приоритет над *GRANT* и *REVOKE*. *DENY* можно использовать в качестве эффективного способа запрещения пользователю определенных привилегий, которые он мог бы иначе получить через членство в группах Windows или ролях SQL Server.

В SQL Server используется механизм старшинства привилегий. Это означает, что если у пользователя есть привилегия, выданная ему напрямую и через членство в группе, и затем привилегия отзывается у группы, то привилегия отзывается и на уровне пользователя тоже.

Системные привилегии *CREATE* или *ALTER* на определенный объект включают в себя и привилегию *DROP* на этот объект. Системные привилегии *CREATE* на объект дополнительно включают в себя привилегию *DROP*. Полный список системных привилегий приводится в табл. 3.3.

Таблица 3.3. Системные привилегии SQL Server

Привилегия	**Описание**
ADMINISTER BULK OPERATIONS	Дает привилегии на выполнение в базе данных массовых операций, таких как *BULK INSERT*.
ALTER ANY {APPLICATION ROLE | DATABASE DDL TRIGGER | DATASPACE |USER }	Дает привилегии на изменение в базе данных любого объекта определенного типа.
{ALTER [ANY] | CREATE} ASSEMBLY	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанной или любой (*ANY*) сборки в базе данных.
{ALTER [ANY] | CREATE} ASYMMETRIC KEY	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) несимметричного ключа безопасности в базе данных.
{ALTER [ANY] | CREATE} CERTIFICATE	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) сертификата безопасности в базе данных.
{ALTER [ANY] | CREATE} CONTRACT	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) контракта брокера сервисов в базе данных.
{ALTER [ANY] | CREATE} DATABASE DDL EVENT NOTIFICATION	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) уведомления о событиях DDL в базе данных.
{ALTER [ANY] | CREATE} FULLTEXT CATALOG	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) полнотекстового каталога в базе данных.

Таблица 3.3 (продолжение)

Привилегия	Описание
CREATE FUNCTION	Дает привилегии на созданий функций.
{ALTER [ANY] | CREATE} MESSAGE TYPE	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) типа сообщений брокера в базе данных.
{ALTER [ANY] | CREATE} REMOTE SERVICE BINDING	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанной или любой (*ANY*) привязки сервисов брокера сервисов в базе данных.
{ALTER [ANY] | CREATE} ROLE	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанной или любой (*ANY*) роли в базе данных.
{ALTER [ANY] | CREATE} ROUTE	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) маршрута брокера сервисов в базе данных.
{ALTER [ANY] | CREATE} SCHEMA	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанной или любой (*ANY*) схемы в базе данных.
{ALTER [ANY] | CREATE} SERVICE	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) сервиса брокера сервисов в базе данных.
{ALTER [ANY] | CREATE} SYMMETRIC KEY	Дает привилегии на создание (*CREATE*) или изменение (*ALTER*) указанного или любого (*ANY*) симметричного ключа в базе данных.
AUTHENTICATE [SERVER]	Дает привилегии на подключение и аутентификацию серверов.
BACKUP {DATABASE | LOG}	Дает привилегии на выполнение операторов *BACKUP DATABASE* и *BACKUP LOG*.
CHECKPOINT	Дает пользователю или роли привилегии на выполнение оператора *CHECKPOINT*.
CONNECT	Дает брокеру сервисов привилегии на подключение к экземпляру SQL Server.
CONNECT REPLICATION	Дает репликационной схеме разрешение на подключение к экземпляру SQL Server.
CONTROL [SERVER]	*CONTROL SERVER* дает все привилегии на все объекты в сервере (*CONTROL* – на все объекты указанной базы либо на конкретный объект), а также разрешение на передачу привилегий другим пользователям.
CREATE {AGGREGATE | DATABASE | DEFAULT | PROCEDURE | QUEUE | RULE | SYSNONYM | TABLE | TYPE | VIEW | XML SCHEMA COLLECTION}	Дает привилегии на создание объектов соответствующего типа в сервере, базе данных или схеме.

Привилегия	Описание
IMPERSONATE	Дает пользователю привилегии на действие от имени другого пользователя, например: *GRANT IMPERSONATE ON USER::katie TO manager123*
RECEIVE	Дает пользователю привилегии на получение сообщений из указанной очереди брокера сервисов, например: *GRANT RECEIVE ON query_notification_errorqueue TO emily;*.
SEND	Дает привилегии на посылку уведомлений очереди из очереди сервиса сообщений указанному пользователю, например: *GRANT SEND ON SERVICE:://mysvc TO dylan;*.
SHOWPLAN	Дает привилегии на просмотр планов выполнения запросов.
SHUTDOWN	Дает привилегии на остановку сервера.
SUBSCRIBE QUERY NOTIFICATIONS	Дает привилегии на получение уведомлений из очереди уведомлений запроса, например: *GRANT SUBSCRIBE QUERY NOTIFICATIONS TO sam;*
TAKE OWNERSHIP	Дает привилегии на изменение владельца коллекции XML-схем с одного пользователя на другого.
VIEW {DATABASE STATE | DEFINITION}	Дает привилегии на просмотр метаданных всей базы данных, вне зависимости от того, является ли пользователь владельцем базы данных (для *DATABASE STATE*) или метаданных сервера, базы данных, схемы или конкретного объекта (для *DEFINITION*).

В SQL Server также есть набор системных ролей с предустановленными привилегиями как на объекты, так и на команды. Системные роли SQL Server следующие:

SYSADMIN

Может выполнять любые действия в базе данных и имеет доступ ко всем объектам.

SERVERADMIN

Может конфигурировать любые опции сервера, а также останавливать и запускать сервер.

SETUPADMIN

Может выполнять процедуры и использовать присоединенные серверы.

SECURITYADMIN

Может читать журналы ошибок, менять пароли, администрировать учетные записи и выдавать привилегию *CREATE DATABASE*.

PROCESSADMIN

Может администрировать процессы, выполняющиеся в SQL Server.

DBCREATOR

Может создавать, изменять и удалять базы данных.

DISKADMIN

Может администрировать диски и файлы.

BULKADMIN

Может использовать утилиту BCP (Bulk Copy Program) и выполнять операторы *BULK INSERT* (а также обычные операторы *INSERT*) для таблиц базы данных.

Роль *SYSADMIN* позволяет выдавать любые привилегии в любой базе данных сервера. Члены ролей *DB_OWNER* и *DB_SECURITYADMIN* могут выдавать любые привилегии на оператор или объект в базе данных, которыми они владеют. Члены ролей *DB_DDLADMIN*, *SYSADMIN* и владельцы баз данных могут выдавать системные привилегии.

Члены системных ролей могут добавлять других пользователей в члены этих ролей. Но для этого нужно использовать не оператор *GRANT*, а системную хранимую процедуру **sp_addsrvrolemember**.

В дополнение к серверным системным ролям в SQL Server есть также набор ролей уровня баз данных. То есть серверные системные роли дают привилегии на все базы данных в сервере, а системные роли из следующего списка дают привилегии только в пределах той базы данных, в которой они выданы:

DB_OWNER

Включает привилегии всех основных ролей и разрешение на выполнение действий по конфигурации и администрированию базы данных.

DB_ACCESSADMIN

Дает привилегии на добавление и удаление из базы данных пользователей и групп SQL Server и Windows.

DB_DATAREADER

Дает привилегии на чтение любой таблицы в базе данных.

DB_DATAWRITER

Дает привилегии на чтение, вставку, модификацию и удаление данных любой таблицы в базе данных.

DB_DDLADMIN

Дает привилегии на создание, изменение и удаление объектов в базе данных, а также на выполнение любых операторов DDL.

DB_SECURITYADMIN

Дает разрешение на администрирование пользователей и ролей, а также объектных и системных привилегий.

DB_BACKUPOPERATOR

Дает привилегии на резервное копирование базы данных.

DB_DENYDATAREADER

Запрещает чтение данных.

DB_DENYDATAWRITER

Запрещает модификацию данных.

Как и в случае с серверными системными ролями, системные роли уровня базы данных выдаются не с помощью оператора *GRANT*, а с помощью хранимой процедуры **sp_addrolemember**.

В следующем примере мы выдаем пользователям *Emily* и *Sarah* системные привилегии *CREATE DATABASE* и *CREATE TABLE*. Затем несколько привилегий на таблицу **titles** выдаются группе *editors* с возможностью передавать эти привилегии дальше:

```
GRANT CREATE DATABASE, CREATE TABLE TO emily, sarah
GO
GRANT SELECT, INSERT, UPDATE, DELETE ON titles
TO editors
WITH GRANT OPTION
GO
```

В следующем примере привилегии выдаются пользователю базы данных *sam* и пользователю Windows *jacob*:

```
GRANT CREATE TABLE TO sam, [corporate\jacob]
GO
```

В последнем примере показано, как выдаются привилегии с использованием ключевого слова *AS*. В этом примере пользователь *emily* владеет таблицей **sales_detail**, и она выдает привилегию *SELECT* на эту таблицу роли *sales_manager*. Пользователь *kelly*, являющийся членом роли *sales_manager*, хочет передать привилегию *SELECT* пользователю *sam*, но он не может этого сделать, так как привилегия была выдана не ему лично, а группе. Для решения этой проблемы можно использовать слово *AS*:

```
-- Первоначальная выдача привилегии
GRANT SELECT ON sales_detail TO sales_manager
WITH GRANT OPTION
GO
-- kelly выдает привилегию пользователю sam от имени роли sales_manager
GRANT SELECT ON sales_detail TO sam AS sales_manager
GO
```

См. также

REVOKE

IN

Оператор *IN* позволяет сформировать список значений (либо явным перечислением значений, либо по результату подзапроса), а затем проверить конкретное значение на вхождение в этот список во фразе *WHERE* или *HAVING*. Другими словами, этот оператор позволяет отвечать на вопросы вида «Входит ли значение A в указанный список значений?».

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
{WHERE | HAVING | {AND | OR}} значение
  [NOT] IN ({значение1, значение2[, ...] | подзапрос})
```

Ключевые слова

{WHERE | HAVING | {AND | OR}} значение

IN можно использовать во фразах *WHERE* и *HAVING*. Также оператор *IN* можно комбинировать с другими условиями с помощью *AND* и *OR*. Проверяемое значение может иметь любой тип данных: обычно это столбец таблицы, используемой в транзакции, или переменная.

NOT

Указывает, что нужно возвращать те значения, которые *не* содержатся в списке.

IN ({значение1, значение2[, ...] | подзапрос})

Указывает список значений, по которому проверяется исходное значение. Тип значений списка должен быть совместим с типом исходного значения. При перечислении значений используются обычные правила: строковые значения должны быть указаны в кавычках, а для числовых значений разделители (кавычки) не требуются. Вместо перечисления значений вы можете использовать подзапрос, возвращающий одно или несколько значений подходящего типа.

В следующем примере для SQL Server мы извлекаем из таблицы **employee** базы **hr** записи о сотрудниках, живущих в штатах Джорджия, Теннеси, Алабама или Кентукки:

```
SELECT *
FROM hr..employee
WHERE home_state IN ('AL','GA','TN','KY')
```

Аналогичным образом мы можем найти в таблице **employee** всех сотрудников, являющихся авторами согласно базе **pubs**:

```
SELECT *
FROM hr..employee
WHERE emp_id IN (SELECT au_id FROM pubs..authors)
```

Мы можем использовать ключевое слово *NOT* для формирования выборки на основании элементов, отсутствующих в списке. В следующем примере штаб-квартира компании находится в Нью-Йорке и многие сотрудники ездят на работу из соседних штатов. Мы хотим отфильтровать всех таких сотрудников:

```
SELECT *
FROM hr..employee
WHERE home_state
  NOT IN ('NY','NJ','MA','CT','RI','DE','NH')
```

Обратите внимание, что Oracle полностью поддерживает функциональность ANSI и при этом расширяет оператор *IN* возможностью использования составных значений. Например, в Oracle можно использовать следующий оператор:

```
SELECT *
FROM hr..employee e
WHERE (e.emp_id, e.emp_dept) IN
( (242, 'sales'), (442, 'mfg'), (747, 'mkt') )
```

См. также

ALL/ANY/SOME
BETWEEN
EXISTS
LIKE
SELECT
SOME/ANY

INSERT

Оператор *INSERT* вставляет строки в таблицу или представление.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Оператор *INSERT* позволяет вставлять строки в таблицу одним из следующих способов:

- Можно вставлять строки со значениями по умолчанию, определенными в операторах *CREATE/ALTER TABLE*.
- Можно явно указать значения, вставляемые в каждый столбец (это наиболее стандартный способ).
- Можно вставить в таблицу сразу несколько строк, возвращаемых оператором *SELECT*.

Синтаксис SQL2003

```
INSERT INTO [ONLY] {имя_таблицы | имя_представления}
   [(столбец1[, ...])]
[OVERRIDE {SYSTEM | USER} VALUES]
{DEFAULT VALUES|VALUES (значение1[, ...])|оператор_select}
```

Ключевые слова

ONLY

Используется только с типизированными таблицами для того, чтобы строки вставлялись только в целевую таблицу, но не в подтаблицы.

{имя_таблицы | имя_представления} [(столбец1[, ...])]

Указывает таблицу или представление, в которое вставляются строки. Вы должны иметь привилегию *INSERT* на таблицу или хотя бы на столбцы, для которых вы указываете значения. Если не задана схема, то используется схема по умолчанию. При необходимости вы можете указать список столбцов, в которые записываются значения.

OVERRIDE {SYSTEM | USER} VALUES

Ключевое слово *SYSTEM* позволяет вставить значение в столбец, который в обычном случае заполняется сервером, например в столбец с автоматически генерируемой последовательностью идентификаторов. *OVERRIDE USER VALUES* делает ровно противоположное: вставляет сгенерированное системой значение, даже если пользователь явно указал свое значение.

DEFAULT VALUES

Вставляет во все столбцы значения по умолчанию, там где они определены с помощью *DEFAULT*, и пустые значения в столбцы, для которых не определены значения по умолчанию. (Конечно, значением по умолчанию можно сделать и пустое значение.) Этот оператор вставляет единственную запись. В зависимости от ограничений *PRIMARY KEY* и *UNIQUE* на целевой таблице, этот оператор может привести к ошибке.

VALUES (значение1[, ...])

Указывает значения, вставляемые в целевую таблицу. Значения из списка должны полностью соответствовать по числу, типу и размеру столбцам, перечисленным в *INSERT INTO*. То есть в первый столбец вставляется первое значение из списка, во второй столбец – второе значение и т. д. При необходимости вы можете использовать ключевые слова *DEFAULT* (для вставки в столбец значения по умолчанию) и *NULL* (для вставки в столбец пустого значения).

оператор_select

Вставляет в таблицу или представление строки, сформированные оператором *SELECT*. Список столбцов в операторе *SELECT* должен полностью соответствовать списку столбцов в операторе *INSERT*. Целевая таблица или представление не могут использоваться в операторе *SELECT* во фразах *FROM* и *JOIN*.

Общие правила

Вы можете вставлять строки в таблицы, а также в представления, построенные на базе одной таблицы. Оператор в формате *INSERT...VALUES* вставляет одну строку из явно заданных значений. В следующем примере мы вставляем информацию об авторе **Jessica Rabbit** в таблицу **authors**:

```
INSERT INTO authors (au_id, au_lname, au_fname, phone,
    address, city, state, zip, contract )
```

```
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT,
    '1717 Main St', NULL, 'CA', '90675', 1)
```

В каждый столбец вставляется явно указанное значение за исключением столбца **phone**, для которого используется значение по умолчанию, и столбца **city**, в который вставляется *NULL*.

Важно помнить, что вы можете пропустить некоторые столбцы, оставив в них пустые значения (если эти столбцы допускают пустые значения). Такие вставки называются *частичными* вставками. Вот пример частичной вставки, делающей то же, что и оператор в предыдущем примере:[1]

```
INSERT INTO authors (au_id, au_lname, au_fname, phone, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT, 1)
```

Оператор *INSERT*, комбинируемый с оператором *SELECT*, позволяет быстро вставить в таблицу несколько записей. При использовании *INSERT...SELECT* важно убедиться, чтобы значения, возвращаемые *SELECT*, имели совместимые типы данных со столбцами целевой таблицы. Вот пример загрузки данных из таблицы **sales** в таблицу **new_sales**:

```
INSERT INTO sales (stor_id, ord_num, ord_date, qty,
  payterms, title_id)
SELECT
  CAST(store_nbr AS CHAR(4)),
  CAST(order_nbr AS VARCHAR(20)),
  order_date,
  quantity,
  SUBSTRING(payment_terms,1,12),
  CAST(title_nbr AS CHAR(1))
FROM new_sales
WHERE order_date >= 01-JAN-2005
-- Извлекаем только свежие записи
```

Вы можете указать в скобках через запятую список столбцов, в которые вставляются значения. Можно не указывать список столбцов, но тогда подразумевается, что вставка производится во все столбцы таблицы. Если для какого-либо столбца значение не указано, то используется значение по умолчанию либо пустое значение. Столбцы в списке могут иметь любой порядок, но они не могут повторяться. Столбцы и вставляемые в них значения должны соответствовать по типу и размеру данных.

В первом из последующих примеров не используется список столбцов, а во втором используются только значения по умолчанию:

```
INSERT INTO authors
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT,
  '1717 Main St', NULL, 'CA', '90675', 1)

INSERT INTO temp_details
DEFAULT VALUES
```

[1] Не совсем то же самое, например столбец ZIP, в отличие от предыдущего случая, получит значения по умолчанию либо NULL, если значение по умолчанию не задано. – *Прим. науч. ред.*

Первый оператор будет выполнен успешно, только если вставляемые значения соответствуют по порядку, количеству и типу данных столбцам таблицы. Любое несоответствие приведет к ошибке. Второй оператор будет успешно выполнен, только если для каждого столбца таблицы определено значение по умолчанию либо столбец допускает значения *NULL*.

Использование операторов *INSERT* без списка столбцов считается очень плохим тоном, так как эти операторы могут перестать корректно работать при небольших изменениях таблиц (например, добавлении нового столбца).

Советы и хитрости

Оператор *INSERT* вызывает ошибку в следующих ситуациях:

- При несоответствии типов данных столбца и вставляемого в столбец значения.
- Когда в столбец с ограничением *NOT NULL* вставляется пустое значение.
- Когда в столбцы с ограничениями *UNIQUE* и *PRIMARY KEY* вставляются повторяющиеся значения.
- Когда вставляемые значения не удовлетворяют ограничениям *CHECK*.
- Когда значение, вставляемое в столбец с ограничением *FOREIGN KEY*, не найдено в первичном ключе соответствующей таблицы.

Наиболее частой ошибкой при выполнении оператора *INSERT* является несоответствие между числом столбцов таблицы и числом вставляемых значений. Если вы случайно пропустите значение, соответствующее определенному столбцу, то весьма вероятно, что вы получите сообщение об ошибке.

Оператор *INSERT* также завершается с ошибкой, если вставляемое значение не соответствует по типу столбцу таблицы. Например, попытка вставки строки типа «Hello, world!» в столбец целого типа приведет к ошибке. С другой стороны, иногда базы данных могут неявно конвертировать значения между некоторыми типами. Например, SQL Server автоматически сконвертирует дату в строку при вставке в столбец типа *VARCHAR*.

Еще одна типичная проблема при использовании *INSERT* – несоответствие длины вставляемого значения и длины столбца таблицы. Например, вставка длинной строки в столбец типа *CHAR(5)* или вставка большого числа в столбец типа *TINYINT* может привести к проблемам. В зависимости от платформы это может быть либо фатальная ошибка с откатом транзакции, либо просто отбрасывание части данных. Ни то ни другое не желательно. Аналогичная проблема может возникнуть при попытке вставки значения *NULL* в столбец, объявленный как *NOT NULL*.

Большинство проблем при использовании INSERT возникают из-за того, что программисты имеют недостаточно информации о таблице. Внимательно изучите таблицу или представление, прежде чем писать операторы INSERT.

MySQL

MySQL поддерживает для оператора *INSERT* несколько опций, подтверждающих репутацию MySQL как высокопроизводительной платформы:

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] [[база_данных.]владелец.]таблица [(столбец1[, ...])]
{VALUES ( {значение1 | DEFAULT}[, ...] ) | оператор_select |
  SET [ON DUPLICATE KEY UPDATE] столбец1=значение1[, ...]}
```

где:

LOW_PRIORITY | DELAYED | HIGH_PRIORITY

LOW_PRIORITY откладывает выполнении *INSERT* до того момента, когда все пользователи закончат чтение таблицы. Эта опция может привести к достаточно долгому ожиданию. Ее не следует использовать с таблицами MyISAM, так как при этом невозможны параллельные вставки. Ключевое слово *DELAYED* позволяет пользователю продолжить работу, даже если оператор *INSERT* еще не завершился. *DELAYED* игнорируется при использовании в формате *INSERT...SELECT* и *INSERT...ON DUPLICATE*. *HIGH_PRIORITY* просто восстанавливает приоритет оператора до нормального уровня при использовании в серверах с низким приоритетом, в остальном приоритет и скорость работы оператора не повышается.

IGNORE

Позволяет игнорировать и не пытаться вставлять строки, приводящие к дубликатам в уникальных и первичных ключах. Без *IGNORE* такие строки при попытке вставки приводят к завершению всего оператора с ошибкой. Если при использовании фразы *IGNORE* встретились дублирующие строки, то они игнорируются, а все остальные вставляются как обычно.

SET столбец = значение

Альтернативный синтаксис, позволяющий указать значения для столбцов по имени.

ON DUPLICATE KEY UPDATE

Позволяет обновить текущую запись, если она приводит к дубликатам в уникальных и первичных ключах.

MySQL не поддерживает описанную в стандарте ANSI фразу *OVERRIDE*.

MySQL обрезает данные при вставке значений, не подходящих по типу или размеру. Например, при вставке значения '10.23 X' в числовой столбец часть 'X' будет отброшена. Если вы попытаетесь вставить в числовой столбец значение, выходящее за границы диапазона типа, то значение будет обрезано. Вставка некорректной даты или времени приведет к нулевому значению в столбце таблицы. При вставке строки «Hello, World!» в столбец типа *CHAR(5)* строка будет урезана до первых пяти символов. Урезание строк применяется к типам *CHAR*, *VARCHAR*, *TEXT* и *BLOB*.

MySQL поддерживает оператор *REPLACE* с синтаксисом, аналогичным синтаксису *INSERT*. *REPLACE* при вставке дубликатов позволяет перезаписать текущие значение и по сути является аналогом *INSERT...IGNORE*.

Oracle

Реализованный в Oracle оператор *INSERT* позволяет вставлять данные в таблицы, представления, секции, подсекции и объектные таблицы. Также поддерживаются расширения, позволяющие вставку строк сразу в несколько таблиц и условную вставку. Синтаксис оператора следующий:

```
-- Стандартный оператор INSERT
INSERT [INTO] {таблица
  [ [SUB]PARTITION { (секция) | (значение_ключа) } ] |
  (подзапрос) [WITH {READ ONLY | CHECK OPTION
  [CONSTRAINT имя_ограничения]}] |
  TABLE (коллекция) [ (+) ] } [псевдоним]
  [(столбец1[, ...])]
{VALUES (значение1[, ...]) [RETURNING выражение1[, ...]
  INTO переменная1[, ...]] |
  оператор_select [WITH {READ ONLY |
    CHECK OPTION [CONSTRAINT имя_ограничения]}]}
-- Условный оператор INSERT
INSERT {[ALL | FIRST]} WHEN условие
  THEN стандартный_оператор_insert
ELSE стандартный_оператор_insert
[LOG ERRORS [INTO [схема.]таблица] [( выражение )]
  [REJECT LIMIT {целое_число | UNLIMITED}]]
```

где:

INSERT [INTO]

Вставляет строку или строки в таблицу, представление, материализованное представление или подзапрос. Ключевое слово *INTO*[1] необязательно. Одиночные записи вставляются с помощью фразы *VALUES*, а наборы записей вставляются с помощью подзапросов.

таблица [[SUB]PARTITION { (секция) | (значение_ключа) }]

Указывает целевую таблицу, в которую вставляются строки. Вы можете полностью квалифицировать имя таблицы, указав схему и связь с другой базой данных; по умолчанию подразумеваются текущая схема и база данных. Если вы вставляете строки не в объектную таблицу или объектное представление, то вы можете вставлять данные в определенную секцию или подсекцию, указав либо название секции, либо значение ключа секционирования.

подзапрос

Позволяет вставить в таблицу сразу целый набор строк, возвращаемый стандартным оператором *SELECT*. Фактически вы с помощью подзапроса на лету создаете представление, так же как и при вставке в представление. Это основной механизм вставки сразу в несколько таблиц. Все значения, возвращаемые подзапросом, должны соответствовать столбцам, в которые производит-

[1] Обязательное ключевое слово. – *Прим. науч. ред.*

ся вставка, иначе возникнет ошибка. При вставке строк в несколько таблиц использование подзапросов обязательно. С подзапросами можно использовать следующие опции:

WITH READ ONLY

Означает, что таблицы и представления, используемые в подзапросе, нельзя обновлять до завершения оператора.

WITH CHECK OPTION [CONSTRAINT имя_ограничения]

Означает, что в таблицу или представление нельзя вставлять строки, которые не проходят указанное ограничение *CHECK*.

TABLE (коллекция) [(+)] } [псевдоним]

Позволяет работать с коллекций как с обычной таблицей для вставки независимо от того, задается эта коллекция подзапросом, столбцом, функцией или конструктором. В любом случае коллекция должна быть вложенной таблицей или *VARRAY*. Вы можете указать для коллекции псевдоним. Псевдонимы запрещено использовать при вставке в несколько таблиц.

(столбец1[, ...])

Указывает список столбцов, в которые вставляются данные. Если вы не используете список столбцов, то ожидается, что значения во фразе *VALUES* или в подзапросе полностью соответствуют столбцам целевой таблицы. Oracle вернет ошибку, если вы не укажете значения для какого-либо столбца, не имеющего значения по умолчанию и объявленного как *NOT NULL*.

VALUES (значение1[, ...]) [RETURNING выражение1[, ...] INTO переменная1[, ...]]

Указывает значения для вставки. В соответствии со стандартом ANSI требуется, чтобы значения соответствовали столбцам, в которые производится вставка. Вы можете использовать *DEFAULT* для вставки в столбец значения по умолчанию и *NULL* – для вставки пустого значения. *DEFAULT* нельзя использовать при вставке в представления. При вставке в несколько таблиц во фразе *VALUES* должны быть соответствующие значения для каждого возвращаемого подзапросом элемента.

RETURNING выражение1

Возвращает вставленные в таблицу значения. Например, вы можете использовать фразу *RETURNING* для получения значения автоматически генерируемого первичного ключа. При вставке одной строки возвращаемые значения сохраняются в переменных, а при множественной вставке нужно использовать массивы. Вы можете использовать *RETURNING* при вставке в таблицы, представления на базе одной таблицы и в материализованные представления. Фразу *RETURNING* нельзя использовать при вставке в несколько таблиц.

INTO переменная1

Указывает переменные, в которые сохраняются значения выражений из фразы *RETURNING*. Вы должны указать переменную для каждого выражения во фразе *RETURNING*. Вы не можете использовать *INTO* для значения типа *LONG*, с удаленными объектами, с представлениями, имеющими триггеры типа *INSTEAD OF*, а также при параллельных операциях *INSERT*, *UPDATE* и *DELETE*.

ALL

Выполняет вставку в несколько таблиц и используется только совместно с подзапросами. Если не указано условие *WHEN*, то выполняется вставка во все перечисленные таблицы. Если же указано условие *WHEN*, то оно проверяется для каждой таблицы, и в зависимости от результата проверки записи вставляются в каждую таблицу независимо от других. Каждый раз, когда условие оценивается как *TRUE*, Oracle выполняет соответствующую фразу *INTO*. Вставки в несколько таблиц не могут выполняться параллельно для индексно-организованных таблиц и для таблиц с битовыми индексами, а также запрещены в следующих случаях:

- При вставках в представления и материализованные представления.
- При вставках в удаленные таблицы.
- При использовании коллекций с помощью выражения *TABLE*.
- При вставке в более чем 999 столбцов.
- При использовании последовательностей в подзапросе.

FIRST

Указывает, что условия *WHEN* должны проверяться по порядку, и как только встретится условие, которое имеет результат *TRUE*, то выполняется соответствующая фраза *INTO*, а остальные фразы *WHEN* пропускаются.

WHEN `условие` *THEN* `стандартный_оператор_insert`

Определяет условие, при истинности которого выполняется оператор *INSERT*. Условие оценивается для каждой строки, возвращаемой подзапросом. Разрешается использовать до 127 фраз *WHEN*.

ELSE `стандартный_оператор_insert`

Выполняется в случае, когда ни одно условие *WHEN* не было выполнено.

LOG ERRORS [INTO [`схема`*.]*`таблица`*] [(*`выражение`*)] [REJECT LIMIT {*`целое_число`* | UNLIMITED}]*

Журналирует значения столбцов в случае возникновения ошибок DML. *INTO* указывает таблицу, в которую записывается информация об ошибках. Если имя таблицы не указано, то Oracle использует таблицы с названием **ERR$_** плюс 25 первых символов названия таблицы, в которую выполняется вставка. Вы можете указать любое выражение (например, *TO_CHAR(SYSDATE)*), которое вы хотите вставлять в таблицу ошибок. *REJECT LIMIT* позволяет задать порог числа ошибок, при достижении которого выполнение оператора *INSERT* будет остановлено. (Обратите внимание, что вы не сможете журналировать значения столбцов типов *LONG*, *LOB* и объектных типов.)

В Oracle поддерживаются стандартные варианты оператора *INSERT*, описанные в ANSI, такие как *INSERT...SELECT* и *INSERT...VALUES*. Однако поддерживаются также некоторые дополнительные варианты.

При вставке в таблицы, для которых используются последовательности, выбирайте следующее значение из последовательности с помощью синтаксиса *<имя_последовательности>.NEXTVAL*. Например, представьте, что вы хотите использовать последовательность **authors_seq** для установки значения столбца **au_id** таблицы **authors**:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES (authors_seq.nextval, 'Rabbit', 'Jessica', 1)
```

При получении значений, вставленных оператором *INSERT*, проверьте соответствие выражений в *RETURNING* переменным в *INTO*. Выражения во фразе *RETURNING* не обязательно должны состоять из столбцов, упомянутых в *VALUES*. Например, следующий оператор *INSERT* вставляет строку в таблицу **sales**, при этом в переменную сохраняется значение столбца, явно не заполняемого:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1)
RETURNING hire_date INTO :temp_hr_dt;
```

Обратите внимание, что во фразе *RETURNING* возвращается столбец **hire_date**, хотя этот столбец и не заполняется во фразе *VALUES*. (Логично предположить, что у этого столбца могло бы быть значение по умолчанию.)

Вот пример безусловной вставки строк одновременно в несколько таблиц:

```
INSERT ALL
  INTO jobs(job_id, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+1, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+2, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+3, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
SELECT job_identifier, job_title, base_pay, max_pay
FROM job_descriptions
WHERE job_status = 'Active';
```

Для более сложных ситуаций можно использовать оператор *INSERT*, вставляющий строки в несколько таблиц, в зависимости от выполнения определенных условий:

```
INSERT ALL
  WHEN job_status = 'Active'     INTO jobs
  WHEN job_status = 'Inactive'   INTO jobs_old
  WHEN job_status = 'Terminated' INTO jobs_cancelled
  ELSE INTO jobs
SELECT job_identifier, job_title, base_pay, max_pay
FROM job_descriptions;
```

Заметьте, что если бы в предыдущем примере вы пропускали некоторые столбцы целевых таблиц, то вы должны были бы использовать после каждой фразу *INTO* фразу *VALUES*. Следующий пример демонстрирует этот синтаксис:

```
INSERT FIRST
  WHEN job_status = 'Active'
    INTO jobs
    VALUES(job_id, job_desc, min_lvl, max_lvl)
  WHEN job_status = 'Inactive'
    INTO jobs_old
    VALUES(job_id, job_desc, min_lvl, max_lvl)
  WHEN job_status = 'Terminated'
```

```
      INTO jobs_cancelled
      VALUES(job_id, job_desc, min_lvl, max_lvl)
    WHEN job_status = 'Terminated'
      INTO jobs_outsourced
      VALUES(job_id, job_desc, min_lvl, max_lvl)
    ELSE INTO jobs
      VALUES(job_id, job_desc, min_lvl, max_lvl)
  SELECT job_identifier, job_title, base_pay, max_pay
  FROM job_descriptions;
```

Обратите внимание, что, так как в примере используется ключевое слово *FIRST*, то из двух условий `job_status='Terminated'` всегда будет отрабатываться только первое (вставка в **jobs_cancelled**), а следующее будет пропускаться.

Oracle позволяет вставлять данные либо в обычном режиме, либо в режиме прямой вставки. При обычной вставке Oracle проверяет ссылочную целостность и повторно использует свободное пространство. В режиме прямой вставки Oracle пишет данные в конец целевой таблицы, не заполняя пустое пространство внутри таблицы. Также в режиме прямой вставки данные пишутся сразу в файлы данных, минуя буферный кэш.

Oracle позволяет использовать подсказки оптимизатора для изменения плана выполнения оператора *INSERT*. Например, вы можете использовать подсказку *APPEND* для включения режима прямой вставки. Для получения информации о других подсказках, используемых с оператором *INSERT*, воспользуйтесь документацией.

Режим прямой вставки увеличивает скорость вставки большого числа записей. Однако Oracle будет выполнять вставку в обычном режиме, если имеет место любая из следующих ситуаций:

- Данные в целевой таблице модифицируются операторами *UPDATE* и *DELETE* в той же транзакции перед оператором *INSERT*. (Но *UPDATE* и *DELETE* можно выполнять после прямой вставки.)
- Оператор *INSERT* является распределенным.
- Целевая таблица содержит столбцы типов *LOB* или объектных типов.
- Целевая таблица организована по индексу.
- Целевая таблица имеет триггеры или ссылочные ограничения целостности.
- Целевая таблица реплицируется.
- Инициализационный параметр *ROW_LOCKING* имеет значение *INTENT*.

Также Oracle не позволит вам читать данные из таблицы, в которую вы только что вставили строки в режиме прямой вставки, пока вы не выполните *COMMIT*.

Oracle позволяет выполнять параллельную прямую вставку в несколько таблиц, но только если вы используете подзапросы, а не обычные фразы *VALUES*.

При вставка значений типа LOB и BFILE есть небольшая хитрость: перед вставкой такие значения должны быть инициализированы как NULL. Также есть особенности у столбцов типа RAW. Если вы

вставите в такой столбец обычную строку, то все последующие запросы, использующие этот столбец, будут требовать полного сканирования таблицы.

PostgreSQL

PostgreSQL поддерживает оператор *INSERT* в соответствии со стандартом ANSI, за исключением отсутствия фразы *OVERRIDE SYSTEM GENERATED VALUES* и наличия расширенной поддержки фразы *RETURNING*:

```
INSERT INTO таблица [(столбец1[, ...])]
{[DEFAULT] VALUES | VALUES {(значение1[, ...]) | DEFAULT}
  | оператор_SELECT}
[RETURNING { * | столбец [AS выходное_имя][, ...] }]
```

где:

(столбец1[, ...])

Указывает один или несколько столбцов целевой таблицы. Список должен быть заключен в скобки, а столбцы должны быть разделены запятыми.

DEFAULT

Позволяет вставить одну строку, полностью состоящую из значений по умолчанию.

RETURNING { | столбец [AS выходное_имя][, ...] }*

Возвращает значения, вставленные в результате выполнения оператора. Вы можете использовать звездочку для возврата значений всех столбцов либо явно перечислить столбцы с указанием их выходных имен. Вы можете использовать *RETURNING*, к примеру, для возврата значения автоматически генерируемого первичного ключа.

PostgreSQL пытается автоматически приводить типы значений, указанных в *VALUES* или получаемых в *SELECT*, если они не соответствуют типам столбцов в целевой таблице.

SQL Server

SQL Server имеет расширенную поддержку оператора *INSERT* по сравнению со стандартом ANSI. В частности, поддерживаются некоторые специальные функции для работы с наборами строк, а также возможность вставлять результаты работы хранимых процедур сразу в целевую таблицу. Синтаксис оператора следующий:

```
[WITH табличное_выражение[, ...]]
INSERT [TOP ( число ) [PERCENT]]
[INTO] таблица [(столбец1[, ...])]
[OUTPUT выражение INTO {@табличная_переменная | таблица}
  [ (список_столбцов[, ...]) ]]
{[DEFAULT] VALUES | VALUES (значение1[, ...]) |
  оператор_select | EXEC[UTE] процедура [[@параметр =]
  значение] [OUTPUT] [, ...]]}
```

где:

WITH табличное_выражение

Объявляет именованный временный набор строк, определяемый оператором *SELECT*.

INSERT [INTO] таблица

Определяет в качестве целевого объекта для вставки таблицу, представление или функцию, работающую с набором строк (rowset function). При вставке строк в представление должна затрагиваться только одна базовая таблица представления. Функции, работающие с наборами строк, позволяют получать данные из специфических или внешних источников данных, таких как потоки XML, файловые структуры для полнотекстового поиска (специальные структуры для хранения внутри базы данных документов типа MS Word или слайдов MS PowerPoint) и внешних источников данных, таких как таблицы Excel. Примеры приводятся позднее в этом разделе. В SQL Server для оператора *INSERT* поддерживаются две функции, работающие с наборами строк.

OPENQUERY

Выполняет указанный передаваемый запрос на связанном сервере. Позволяет эффективно выполнить оператором *INSERT* вставку записей в источник, внешний по отношению к SQL Server. Источник данных должен быть объявлен как связанный сервер.

OPENROWSET

Выполняет указанный передаваемый запрос на внешнем источнике данных. Эта функция похожа на *OPENDATASOURCE*, за исключением того, что *OPENDATASOURCE* лишь открывает источник данных, но не передает данные оператору *INSERT*. Функция *OPENROWSET* предназначена для эпизодического нерегламентированного использования.

TOP (число) [PERCENT]

Указывает точное число строк или процент от общего числа строк, которые должны быть вставлены. Если число указано не литералом, а выражением, то оно должно быть заключено в скобки. Если используется ключевое слово *PERCENT*, то выражение должно иметь тип *FLOAT* и иметь значение от 0 до 100. Если *PERCENT* не используется, то выражение должно иметь тип *BIGINT*.

(столбец1[, ...])

Определяет один или несколько столбцов таблицы, в которые вставляются значения. Столбцы должны быть разделены запятыми, сам список должен быть в скобках. SQL Server автоматически заполняет столбцы с типами *IDENTITY* и *TIMESTAMP*, а также имеющие значения по умолчанию.

OUTPUT выражение INTO {@табличная_переменная | таблица}[(список_столбцов[, ...])]

Возвращает строки, вставленные командой (по умолчанию *INSERT* возвращает лишь число вставленных строк), и сохраняет их либо в табличную переменную, либо в таблицу. Таблица, в которую возвращаются вставленные сроки, не должна иметь триггеров, внешних ключей и ограничений *CHECK*. Столбцы в этой таблице должны соответствовать указанному во фразе списку столбцов либо, если список столбцов не указан, всем столбцам целевой таблицы.

DEFAULT

Позволяет вставить одну строку, полностью состоящую из значений по умолчанию.

EXEC[UTE] процедура [[@параметр =] значение] [OUTPUT][, ...]]

Дает команду на выполнение динамического оператора Transact-SQL, хранимой процедуры, удаленной процедуры (RPC) или расширенной хранимой процедуры и сохранение результата выполнения в локальной таблице. Вы можете задать значения параметров процедуры, а также пометить необходимые параметры как выходные (*OUTPUT*). Поля возвращаемого набора строк должны соответствовать столбцам целевой таблицы.

SQL Server автоматически генерирует значения для столбцов типа *IDENTITY* и *TIMESTAMP*, поэтому при вставке их можно опускать. Однако значения для столбцов типа *UNIQUEIDENTIFIER* автоматически не генерируются, и при вставке строк эти значения нужно генерировать при помощи функции *NEWID( )*:

```
INSERT INTO guid_sample (global_ID, sample_text, sample_int)
VALUES (NEWID( ), 'insert first record','10000')
GO
```

При переносе кода с платформы на платформу имейте в виду, что в SQL Server вставка в столбец типа *TEXT* или *VARCHAR* пустой строки (‘’) приводит к сохранению строки нулевой длины. Это не то же самое, что значение *NULL*, как на некоторых других платформах. При вставке строк оператором в форме *INSERT...SELECT* можно определять подзапрос при помощи *WITH*, но при этом нельзя использовать *READPAST*, *NOLOCK* и *READUNCOMMITED*.

Следующий пример иллюстрирует применение *INSERT...EXEC*. Сначала в примере создается временная таблица **#ins_exec_container**, а потом в эту таблицу вставляется список содержимого каталога *c:\temp*. Вторая команда *INSERT* вставляет в таблицу sales результат динамического оператора *SELECT*:

```
CREATE TABLE #ins_exec_container (result_text
VARCHAR(300) NULL)
GO
INSERT INTO #ins_exec_container
EXEC master..xp_cmdshell "dir c:\temp"
GO

INSERT INTO sales
EXECUTE ('SELECT * FROM sales_2002_Q4')
GO
```

Эта функциональность может быть очень полезна при построении бизнес-логики на базе хранимых процедур Transact-SQL, например для определения состояния объектов внутри или вне базы данных и обработки этих результатов в T-SQL.

SQL Server позволяет использовать подсказки оптимизатору для изменения плана выполнения оператора *INSERT*. Однако этот способ настройки рекомендуется только самым продвинутым пользователям. За информацией о подсказках оптимизатору, используемых в *INSERT*, обращайтесь к документации.

Табличные выражения можно использовать в операторах *INSERT*, *UPDATE*, *DELETE*, *SELECT* и *CREATE VIEW*. Табличные выражения позволяют определять временные именованные наборы строк, формируемые оператором SELECT и допускающие рекурсивное использование. В табличных выражениях нельзя использовать *COMPUTE*, *COMPUTE BY*, *FOR XML*, *FOR BROWSE*, *INTO*, *OPTION* и *ORDER BY*. В табличном выражении можно использовать несколько операторов SELECT, если они объединены при помощи *UNION*, *UNION ALL*, *EXCEPT* и *INTERSECT*.

Вот пример простого оператора *INSERT*, использующего табличное выражение:

```
WITH direct_reports (Manager_ID, DirectReports) AS
( SELECT manager_ID, COUNT(*)
  FROM hr.employee AS e
  WHERE manager_id IS NOT NULL
  GROUP BY manager_id )
DELETE FROM direct_reports
WHERE DirectReports <= 1;
```

Можно выполнить *INSERT* с возвратом вставленных значений при помощи фразы *OUTPUT*:

```
INSERT hr.employee
OUTPUT
  INSERTED.employee_id, INSERTED.employee_lname,
  INSERTED.employee_fname
INTO @my_temporary_table_variable
VALUES ('Insert Error', GETDATE( ) );
```

См. также

DELETE
MERGE
SELECT
UPDATE

INTERSECT

Оператор *INTERSECT* возвращает из двух или более запросов строки, которые есть в каждом из запросов. В некотором роде оператор *INTERSECT* похож на *INNER JOIN (см. секцию с описание JOIN)*.

Оператор *INTERSECT* относится к классу *операторов работы с множествами*, куда также входят операторы *EXCEPT* и *UNION*. Все эти операторы используются одинаковым образом для работы с множествами строк, отсюда название класса операторов.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с ограничениями
PostgreSQL	Поддерживается с ограничениями
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

Теоретически вы можете объединять с помощью оператора *INTERSECT* произвольное число запросов. Синтаксис оператора следующий:

```
<SELECT_оператор1>
INTERSECT [ALL | DISTINCT]
[CORRESPONDING [BY (столбец1, столбец2, ...)]]
<SELECT_оператор2>
INTERSECT [ALL | DISTINCT]
[CORRESPONDING [BY (столбец 1, столбец2, ...)]]
...
```

Ключевые слова

ALL

Включает повторяющиеся во всех запросах строки.

DISTINCT

Удаляет повторяющиеся строки из всех результирующих множеств, предшествовавших выполнению оператора *INTERSECT*. Столбцы со значениями *NULL* считаются совпадающими. (Если не указано ни *ALL*, ни *DISTINCT*, то используется *DISTINCT*)

CORRESPONDING

Указывает, что из запросов используются только столбцы с одинаковыми именами, даже если в запросах вместо списка столбцов используются звездочки.

BY

Указывает, что из запросов используются только указанные столбцы, даже если в запросах есть другие столбцы с совпадающими именами. Эту фразу нужно использовать совместно с *CORRESPONDING*.

Общие правила

При использовании *INTERSECT* необходимо помнить только одно: количество и порядок столбцов в запросах должны быть одинаковыми. Типы столбцов не обязательно должны совпадать, но должны быть совместимы (например, как *CHAR* и *VARCHAR*). Как правило, в результирующем наборе для каждого столбца будет использоваться самый большой (по размерности) тип, соответствующий этому столбцу в отдельных запросах.

Советы и хитрости

Ни в одной из платформ не поддерживается фраза *CORREPONDING BY*.

Если в платформе не поддерживается оператор *INTERSECT*, то используйте вместо него оператор *FULL JOIN*.

По стандарту ANSI оператор *INTERSECT* имеет более высокий приоритет, чем другие операторы работы с множествами. Однако не все платформы придерживаются этого правила. Вы можете явно указать порядок выполнения операторов

с помощью скобок. В противном случае СУБД будет выполнять операторы *INTERSECT* слева направо или сверху вниз.

В соответствии со стандартом при использовании *INTERSECT* допускается только одна фраза *ORDER BY* в последнем операторе *SELECT*. Для исключения неоднозначности используйте для каждого столбца псевдонимы. Например:

```
SELECT a.au_lname AS 'lastname', a.au_fname AS 'firstname'
FROM authors AS a
INTERSECT
SELECT e.emp_lname AS 'lastname', e.emp_fname AS 'firstname'
FROM employees AS e
ORDER BY lastname, firstname
```

Хотя наборы столбцов запросов могут иметь совместимые типы данных, в некоторых платформах могут быть различия в обработке длин столбцов. Например, если столбец **au_lname** из первого запроса имеет значительно большую длину, чем столбец **emp_name** из второго запроса, то правила выбора длины столбца для конечного результата могут быть различными в разных платформах. Хотя обычно просто выбирается тип с наибольшей длиной (наименее ограниченный).

Каждая СУБД может иметь свои правила именования столбцов в случае, когда столбцы в запросах имеют разные названия. Обычно используются названия столбцов из первого запроса.

MySQL

Не поддерживается.

Oracle

Oracle поддерживает базовую функциональность операторов *INTERSECT* и *INTERSECT ALL* в соответствии со следующим синтаксисом:

```
<SELECT_оператор1>
INTERSECT
<SELECT_оператор2>
INTERSECT
...
```

Фраза *CORRESPONDING* не поддерживается. Вместо *INTERSECT DISTINCT* следует использовать функционально эквивалентный *INTERSECT*. Нельзя использовать оператор *INTERSECT* в следующих случаях:

- Если запросы содержат столбцы типов LONG, BLOB, CLOB, BFILE и VARRAY.
- Если запросы содержат фразы *FOR UPDATE* и выражения *TABLE*.

Если первый запрос содержит выражения в списке столбцов, то для этих выражений при помощи *AS* должны быть заданы псевдонимы. Также только последний запрос может содержать фразу *ORDER BY*.

Например, мы можем найти идентификаторы магазинов, в которых были продажи, с помощью следующего запроса:

```
SELECT stor_id FROM stores
INTERSECT
SELECT stor_id FROM sales
```

PostgreSQL

PostgreSQL поддерживает операторы *INTERSECT* и *INTERSECT ALL* в соответствии со следующим синтаксисом:

```
<SELECT_оператор1>
INTERSECT [ALL]
<SELECT_оператор2>
INTERSECT [ALL]
...
```

PostgreSQL не поддерживает оператор *INTERSECT* для запросов, содержащих фразу *FOR UPDATE*, и не поддерживает фразу *CORRESPONDING*. Вместо *INTERSECT DISTINCT* следует использовать функционально эквивалентный *INTERSECT*.

Самый первый запрос не может содержать фразы *LIMIT* и *ORDER BY*. Последующие запросы могут содержать эти фразы, но такие запросы должны быть заключены в скобки. Иначе самое последнее вхождение *LIMIT* и *ORDER BY* будет применено к конечному результату. Например, вы могли бы получить список авторов, являющихся сотрудниками и фамилии которых начинаются на 'P', следующим запросом:

```
SELECT a.au_lname
FROM authors AS a
WHERE a.au_lname LIKE 'P%'
INTERSECT
SELECT e.lname
FROM employee AS e
WHERE e.lname LIKE 'P%';
```

SQL Server

SQL Server поддерживает оператор *INTERSECT*, используя базовый синтаксис ANSI:

```
<SELECT_оператор1>
INTERSECT [ALL]
<SELECT_оператор2>
INTERSECT [ALL]
...
```

В качестве названий столбцов результирующего множества используются названия столбцов из первого запроса. Любые имена или псевдонимы столбцов, используемые в *ORDER BY*, должны быть указаны в первом запросе. При использовании операторов *INTERSECT* (или *EXCEPT*) для нескольких запросов выполнение происходит следующим образом: сначала выполняется множественный оператор для первой пары запросов, потом промежуточный результат соединяется со следующим запросом и т. д. Порядок выполнения определяется в первую очередь скобками, затем выполняется *INTERSECT*, а самый низкий приоритет – у операторов *EXCEPT* и *UNION*.

Также обратите внимание, что в качестве альтернативы вы можете использовать операторы *NOT IN* и *NOT EXISTS* вместе с коррелированными подзапросами. Пример приводится в описаниях операторов *IN* и *EXISTS*.

См. также

EXCEPT
SELECT
UNION

IS

Оператор *IS* используется для проверки значения на NULL.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
{WHERE | {AND | OR}} выражение IS [NOT] NULL
```

Ключевые слова

{WHERE | {AND | OR}} выражение IS NULL

Возвращает булево значение *TRUE*, если выражение имеет значение *NULL*, и *FALSE* в противном случае. Проверку на *NULL* можно использовать во фразе *WHERE* или сочетать с другими выражениями с помощью *AND* и *OR*.

NOT

Инвертирует предикат: оператор возвращает *TRUE*, если выражение не равно *NULL*, и *FALSE* в противном случае.

Общие правила

Так как *NULL* соответствует неизвестному значению, то для проверки на *NULL* вы не можете использовать обычные операторы сравнения. Например, ни выражение *X=NULL*, ни *X<>NULL* не могут быть истинными, так как неизвестно, равно *X* или не равно неизвестному значению.

Вместо обычных операторов сравнения следует использовать оператор *IS NULL*. Будьте внимательны: не пишите слово *NULL* в кавычках, так как при этом оно будет воспринято как литерал «NULL», а не как специальное значение NULL.

Советы и хитрости

В некоторых платформах для проверки на пустое значение можно использовать обычные операторы сравнения. Но в любом случае, стандартный оператор *IS [NOT] NULL* поддерживается всеми платформами.

Иногда проверка на *NULL* может усложнить фразу *WHERE*. Например, вместо простого предиката, проверяющего поле **stor_id**:

```
SELECT stor_id, ord_date
FROM sales
WHERE stor_id IN (6630, 7708)
```

вам может понадобиться добавить второй предикат для корректной обработки пустых значений в **stor_id**:

```
SELECT stor_id, ord_date
FROM sales
WHERE stor_id IN (6630, 7708)
OR stor_id IS NULL
```

См. также

SELECT

WHERE

JOIN

Подфраза *JOIN* позволяет извлечь строки из двух или более логически связанных таблиц. Вы можете использовать различные условия соединения таблиц и типы соединений, при этом типы поддерживаемых соединений могут значительно варьироваться между платформами.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

```
FROM таблица [AS псевдоним] {CROSS JOIN |
[NATURAL] [тип_соединения] JOIN
  соединяемая_таблица [[AS] псевдоним]
{ ON условие1 [{AND | OR} условие2] [...] |
USING (столбец1[, ...]) }}
[...]
```

Ключевые слова

FROM таблица

Указывает первую таблицу или представление, участвующую в соединении.

NATURAL

Указывает, что соединение (неважно, внешнее или внутреннее) выполняется по равенству значений во всех столбцах с одинаковыми названиями. При этом вам не нужно использовать фразы *ON* и *USING*. Запрос с натуральным соединением будет выполнен с ошибкой, если в соединяемых таблицах не окажется столбцов с одинаковыми именами.

[тип_соединения] JOIN соединяемая_таблица

Указывает тип используемого соединения и вторую (или последующую) соединяемую таблицу. Для соединяемых таблиц можно указать псевдонимы. Возможны следующие типы соединений:

CROSS JOIN

Кросс-соединение (также известное как картезианское произведение) соединяет каждую строку первой таблицы с каждой строкой второй таблицы. Такое поведение также достигается при отсутствии условия соединения. Прибегать к использованию кросс-соединений не рекомендуется.

[INNER] JOIN

При *внутреннем соединении* из каждой таблицы отсеиваются строки, к которым не присоединяются строки из другой таблицы. Этот тип соединения используется по умолчанию.

LEFT [OUTER] JOIN

Левое внешнее соединение возвращает все строки из таблицы, указанной слева от слова *JOIN*. Если для некоторых строк из левой таблицы нет соответствующих строк в правой, то они все равно не отбрасываются, при этом в столбцы, соответствующие правой таблице, помещаются значения *NULL*. Хорошим тоном считается не смешивать использование правых и левых соединений, а использовать всегда только левое соединение.

RIGHT [OUTER] JOIN

Правое внешнее соединение возвращает все строки из таблицы, указанной справа от слова *JOIN*. Если для некоторых строк из правой таблицы нет соответствующих строк в левой, то они все равно не отбрасываются, при этом в столбцы, соответствующие левой таблице, помещаются значения *NULL*.

FULL [OUTER] JOIN

Полное внешнее соединение возвращает все строки из обеих таблиц, вне зависимости от того, есть ли для определенной строки соответствующая строка в противоположной таблице. Для строк, не соединенных со строками противоположной таблицы, столбцам противоположной таблицы присваиваются пустые значения.

UNION JOIN

Возвращает все столбцы и все строки из обеих таблиц. Для строк, не соединенных со строками противоположной таблицы, столбцам противоположной таблицы присваиваются пустые значения.

[AS] псевдоним

Указывает псевдонимы для соединяемых таблиц. Слово *AS* необязательно.

ON условие_соединения

Соединяет строки из таблицы, указанной во *FROM*, и таблицы, указанной в *JOIN*. Вы можете указывать несколько операторов *JOIN*, основанных на общем наборе значений. Эти значения обычно содержатся в столбцах с одинаковыми названиями и типом, присутствующих в обеих таблицах. Обычно (но не всегда) ключ соединения является первичным ключом в одной таблице

и внешним ключом в другой. Если данные в столбцах совпадают, то строки соединяются.

Синтаксис условия соединения следующий (мы намеренно не детализируем тип соединения):

```
FROM таблица1
JOIN таблица2
  ON таблица1.столбец1 = таблица1.столбец2
    [{AND|OR} таблица1.столбец3= таблица3.столбец4]
    [...]
JOIN таблица3
  ON таблица1.столбец1 = таблица3.столбец2
    [{AND|OR} таблица1.столбец3 = таблица3.столбец4]
    [...]
[JOIN...]
```

Используйте операторы *AND* и *OR* для соединений по нескольким условиям. Также неплохо заключать в скобки каждую пару соединяемых таблиц, так как это облегчает чтение запроса.

USING (столбец1[,...])

Указывает, что условием соединения является равенство значений в указанных столбцах, имеющихся в каждой таблице. Написание фразы *USING* может быть быстрее, чем *ON таблица1.столбецA=таблица2.столбецA*, но результат при этом одинаковый.

Общие правила

Соединения позволяют получать в результирующем множестве строки, составленные из логически связанных строк двух или более таблиц. Вы можете использовать либо ANSI-синтаксис соединений, либо тэта-соединения. Тэта-соединения являются старым способом соединений, при котором условия соединения пишутся во фразе *WHERE*.

К примеру, у вас может быть таблица **employee** с информацией о всех сотрудниках компании. Однако в этой таблице нет полной информации о должности сотрудника, а есть только идентификатор должности **job_id**. Описание должности (поля **description** и **title**) хранится в таблице **job**. Используя соединение, вы легко можете получить результат, содержащий столбцы из обеих таблиц. Следующие примеры иллюстрируют тэта-соединения и соединения в синтаксисе стандарта ANSI:

```
/* Тэта-соединение */
SELECT emp_lname, emp_fname, job_title
FROM employee, jobs
WHERE employee.job_id = jobs.job_id;

/* ANSI соединение */
SELECT emp_lname, emp_fname, job_title
FROM employee
JOIN jobs ON employee.job_id = jobs.job_id;
```

Если вы используете в одном запросе несколько столбцов, то вы должны избегать неоднозначности. Другими словами, либо имена столбцов должны быть

уникальны, либо вместе с именем столбца нужно также указывать таблицу, из которой берется столбец. В предыдущем примере столбец с именем **job_id** есть в каждой таблице, поэтому требуется указывать имя таблицы (если столбец с определенным именем есть только в одной таблице, то указывать имя таблицы для такого столбца необязательно). Однако такие запросы может быть достаточно сложно читать, поэтому можно ссылаться на таблицы по определенным для них псевдонимам:

```
SELECT e.emp_lname, e.emp_fname, j.job_title
FROM employee AS e
JOIN jobs AS j ON e.job_id = j.job_id;
```

В предыдущих примерах демонстрировалось только *эквисоединение*, то есть соединение по условию равенства значений. Однако вы можете использовать и любые другие операторы сравнения: >, <, >= и др.

Вы не можете соединять строки по условиям на значения типов *LOB* (*BLOB*, *CLOB* и т. п.). Все остальные типы данных обычно допустимы в условиях соединения.

Использование картезианского произведения (соединения таблиц, при котором строки соединяются во всех возможных сочетаниях) является *очень плохой идеей*. Прочитайте идущее далее описание оператора *CROSS JOIN*, чтобы знать, как он выглядит, и избегать его.

Рассмотрим примеры каждого типа соединения.

CROSS JOIN

Вот несколько примеров кросс-соединений. В первом случае это тэта-соединение с отсутствующим условием соединения. Во втором случае используется фраза *CROSS JOIN*. В третьем случае это фраза *JOIN* с также отсутствующим условием.

```
SELECT *
FROM employee, jobs;
SELECT *
FROM employee
CROSS JOIN jobs;
SELECT *
FROM employee
JOIN jobs;
```

INNER JOIN

Вот пример внутреннего соединения с использованием нового синтаксиса:

```
SELECT a.au_lname AS 'first name',
  a.au_fname AS 'last name',
  p.pub_name AS 'publisher'
FROM authors AS a
INNER JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

В таблице **authors** хранится информация о многих авторах, однако только у небольшого числа авторов город проживания совпадает с городом какого-либо издательства. Предыдущий запрос, выполненный в базе **pubs** SQL Server, возвращает примерно следующий результат:

```
first name   last name   publisher
------------ ----------- ---------------------
Carson       Cheryl      Algodata Infosystems
Bennet       Abraham     Algodata Infosystems
```

Соединение называется внутренним, потому внутри результирующего множества остаются только те строки, для которых есть соответствующая строка в другой таблице. В некоторых платформах вы можете написать тот же самый запрос с использованием фразы *USING* вместо *ON*:

```
SELECT a.au_lname AS 'first name',
  a.au_fname AS 'last name',
  p.pub_name AS 'publisher'
FROM authors AS a
INNER JOIN publishers AS p USING (city)
ORDER BY a.au_lname DESC
```

Результат этого запроса будет точно таким же, как и у предыдущего.

LEFT [OUTER] JOIN

Хорошей идеей является не смешивать использование правых и левых внешних соединений, а для порядка использовать только левые соединения.

В следующем примере мы с помощью *LEFT OUTER JOIN* получаем издателя для каждого автора (вместо *ON* мы можем использовать *USING*, как показано в примере для *INNER JOIN*):

```
SELECT a.au_lname AS 'first name',
  a.au_fname AS 'last name',
  p.pub_name AS 'publisher'
FROM authors AS a
LEFT OUTER JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

Этот запрос возвращает каждого автора, а также имя его издателя, если издатель в том же городе найден, либо NULL, если издатель не найден. Например, в базе **pubs** SQL Server результат запроса мог бы быть следующим:

```
first name   last name   publisher
------------ ----------- ----------------
Yokomoto     Akiko       NULL
White        Johnson     NULL
Stringer     Dirk        NULL
Straight     Dean        NULL
...
```

Из левой таблицы (таблицы **authors**) возвращаются все данные, а если в правой таблице (таблице **publishers**) не найдено строки для соединения, то в результирующем множестве в столбцах правой строки находятся *значения NULL*.

RIGHT [OUTER] JOIN

Правое внешнее соединение работает, в целом, аналогично левому внешнему соединению, за исключением того, что все строки возвращаются не из левой, а из правой таблицы. Например, в базе **pubs** SQL Server следующий запрос

```
SELECT a.au_lname AS 'first name',
  a.au_fname AS 'last name',
  p.pub_name AS 'publisher'
FROM authors AS a
RIGHT OUTER JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

мог бы вернуть следующий результат:

```
first name   last name   publisher
------------ ----------- ----------------------
Carson       Cheryl      Algodata Infosystems
Bennet       Abraham     Algodata Infosystems
NULL         NULL        New Moon Books
NULL         NULL        Binnet & Hardley
```

Возвращаются все строки из правой таблицы (отсюда название «правое соединение»), все столбцы из левой таблицы имеют значения NULL, если соответствующие строки не были найдены.

NATURAL [INNER|{LEFT|RIGHT}[OUTER]] JOIN

При натуральных соединениях не нужно использовать фразы *ON* и *USING*.

```
SELECT a.au_lname AS 'first name',
  a.au_fname AS 'last name',
  p.pub_name AS 'publisher'
FROM authors AS a
NATURAL RIGHT OUTER JOIN publishers AS p
ORDER BY a.au_lname DESC
```

Этот запрос вернет тот же результат, что и предыдущий, при условии, что столбец **city** есть в обеих таблицах, и это единственный совпадающий по названию столбец. Вы можете использовать префикс *NATURAL* с любым типом соединения (*INNER*, *FULL*, *OUTER*).

FULL [OUTER] JOIN

При полном внешнем соединении из обеих таблиц возвращаются все данные, все зависимости от того, найдена ли для строки соответствующая строка в другой таблице. Если какая-то строка не была соединена со строками другой таблицы, то в результирующем наборе проставляются пустые значения. Ключевое слово *OUTER* необязательно. Предыдущий запрос с *FULL OUTER JOIN* выглядит следующим образом:

```
SELECT a.au_lname AS 'first name',
  a.au_fname AS 'last name',
  p.pub_name AS 'publisher'
FROM authors AS a
FULL JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

Результирующий набор строк этого запроса по сути состоит из результатов запросов *INNER*, *LEFT* и *RIGHT* (некоторые строки мы опустили для краткости):

```
first name      last name   publisher
--------------- ----------- --------------------
Yokomoto        Akiko       NULL
White           Johnson     NULL
Stringer        Dirk        NULL
...
Dull            Ann         NULL
del Castillo    Innes       NULL
DeFrance        Michel      NULL
Carson          Cheryl      Algodata Infosystems
Blotchet-Halls  Reginald    NULL
Bennet          Abraham     Algodata Infosystems
NULL            NULL        Binnet & Hardley
NULL            NULL        Five Lakes Publishin
NULL            NULL        New Moon Books
...
NULL            NULL        Scootney Books
NULL            NULL        Ramona Publishers
NULL            NULL        GGG&G
```

Как вы видите, при полном внешнем соединении вы получаете как строки с пустыми значениями в левой части, так и строки с пустыми значениями в правой части.

Советы и хитрости

Если вы не указываете явно тип соединения, то по умолчанию используется *INNER JOIN*. Имейте в виду, что существует много типов соединения, каждый со своими правилами и особенностями поведения, описанными в предыдущем разделе.

В общем случае желательно описывать условия соединения во фразе *JOIN*, а не в *WHERE*. Это не только делает ваш код чище, позволяя отделять условия соединения от условий поиска, но и позволяет избежать некоторых ошибок, встречающихся в некоторых платформах в реализации внешних соединений через условия в *WHERE*.

В целом мы не рекомендуем в целях ускорения разработки применять ключевые слова типа *NATURAL*, так как при этом подфраза с соединением не будет обновлена автоматически при изменении структуры таблиц. При изменениях таблиц без соответствующих изменений запросов запросы могут перестать работать корректно.

Не во всех платформах поддерживаются все типы соединений, поэтому детальная информация по каждой платформе приводится далее.

Соединения более чем двух таблиц могут представлять определенную трудность. В этом случае стоит рассматривать такое соединение как последовательность нескольких попарных соединений.

MySQL

MySQL поддерживает большую часть типов соединений, определенных стандартом ANSI, кроме натуральных внутренних соединений (натуральные соединения в MySQL бывают только внешние). Синтаксис соединений в MySQL следующий:

```
FROM таблица [AS псевдоним]
{[STRAIGHT_JOIN соединяемая_таблица] |
{ {[INNER] | [CROSS] |
[NATURAL] [ {LEFT | RIGHT | FULL} [OUTER] ]}
JOIN соединяемая_таблица [AS псевдоним]
{ ON условие1 [{AND|OR} условие2] [...] } |
USING (столбец1[, ...]) }}
[...]
```

где:

STRAIGHT_JOIN

Заставляет оптимизатор соединять таблицы в порядке их перечисления во фразе *FROM*, в остальном это ключевое слово эквивалентно обычному *JOIN*. Эта возможность была введена из-за того, что иногда MySQL выбирает неверный порядок соединения.

Примеры приводятся в предыдущем разделе «Общие правила».

Oracle

Oracle полностью поддерживает стандарт ANSI в плане соединений. Однако поддержка стандартного синтаксиса соединений появилась лишь начиная с 9-й версии, поэтому старый код использует только соединения при помощи условий в *WHERE*. Старый синтаксис Oracle для внешних соединений состоял в добавлении (+) к столбцам с противоположной стороны направления соединения. Причиной этого является тот факт, что к таблице, из которой выбирались только присоединившиеся строки, добавлялась строка из значений NULL.

Например, в следующем запросе выполняется *RIGHT OUTER JOIN* таблиц **authors** и **publishers**. Запрос в старом синтаксисе соединений выглядит следующим образом:

```
SELECT a.au_lname AS 'first name',
  a.au_fname AS 'last name',
  p.pub_name AS 'publisher'
FROM authors a, publishers p
WHERE a.city(+) = p.city
ORDER BY a.au_lname DESC
```

А тот же запрос в стандартом синтаксисе ANSI имеет следующий вид:

```
SELECT a.au_lname AS 'first name',
  a.au_fname AS 'last name',
  p.pub_name AS 'publisher'
FROM authors AS a
RIGHT OUTER JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

Примеры различных типов соединений приводятся в предыдущем разделе «Общие правила».

Oracle предлагает уникальную функциональность секционированных соединений, полезную при заполнении пропусков в разреженных данных. Представьте себе, что в таблице **manufacturing** мы храним информацию о производстве: каждая запись таблицы содержит дату, идентификатор продукта и количество произведенного продукта. Если в какой-то день определенный продукт не производился, то запись для этой даты и этого продукта не хранится. Это называется разреженными данными. Для целей отчетности было бы неплохо уметь делать выборки, в которых для каждого продукта была бы запись за каждый день независимо от того, производился ли продукт в этот день. Секционированные внешние соединения позволяют с легкостью решить эту задачу, определив логическую секцию и выполнив внешнее соединение для каждой секции. В следующем примере мы выполняем соединение с таблицей **times** и для каждого **product_id** выводим запись для каждого дня в указанном временном диапазоне:

```
SELECT times.time_id AS time,
       product_id    AS id,
       quantity      AS qty
FROM manufacturing
PARTITION BY (product_id)
RIGHT OUTER JOIN times
ON (manufacturing.time_id = times.time_id)
WHERE manufacturing.time_id
BETWEEN TO_DATE('01/10/05', 'DD/MM/YY')
  AND TO_DATE('06/10/05', 'DD/MM/YY')
ORDER BY 2, 1;
```

Вот результат этого запроса:

```
time      id     qty
--------- ------ ---
01-OCT-05 101    10
02-OCT-05 101
03-OCT-05 101
04-OCT-05 101    17
05-OCT-05 101    23
06-OCT-05 101
01-OCT-05 102
02-OCT-05 102
03-OCT-05 102    43
04-OCT-05 102    99
05-OCT-05 102
06-OCT-05 102    87
```

Получение того же результата без использования секционированных соединений было бы более сложным и менее эффективным.

PostgreSQL

PostgreSQL полностью поддерживает стандарт ANSI. Примеры различных типов соединений приводятся в предыдущем разделе «Общие правила».

SQL Server

SQL Server поддерживает соединения типов *INNER*, *OUTER* и *CROSS* с использованием фразы *IN*. Синтаксис *NATURAL* и *USING* не поддерживается. Синтаксис соединений в SQL Server следующий:

```
FROM таблица [AS псевдоним]
{ {[INNER] | [CROSS] | [ {LEFT | RIGHT | FULL} [OUTER] ]}
JOIN соединяемая_таблица [AS псевдоним]
{ ON условие1 [{AND|OR} условие2] [...] } }
[...]
```

Примеры различных типов соединений приводятся в предыдущем разделе «Общие правила».

См. также

SELECT

WHERE

LIKE

Оператор *LIKE* позволяет проверять строки на соответствие различным шаблонам. Преимущественно это используется во фразе *WHERE* операторов *SELECT*, *INSERT*, *UPDATE* и *DELETE*. Шаблоны строк могут содержать групповые символы, набор поддерживаемых групповых символов зависит от платформы.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
WHERE выражение [NOT] LIKE шаблон
  [ESCAPE управляющий_символ]
```

Ключевые слова

WHERE выражение LIKE

Возвращает булево значение *TRUE*, если выражение соответствует шаблону. Выражение может быть столбцом, константой, переменной, скалярной функцией или любой комбинацией перечисленного. Но выражение не может быть пользовательского типа и типа *LOB*.

NOT

Инвертирует предикат: оператор возвращает *TRUE*, если выражение не соответствует шаблону, и *FALSE* в противном случае.

ESCAPE управляющий_символ

Позволяет искать в строке символы, которые в шаблоне обычно трактуются как групповые.

Общие правила

Проверка строк по шаблонам выполняется оператором *LIKE* очень просто, но нужно помнить о нескольких правилах:

- Имеют значение все символы, включая пробелы в начале и конце строк.
- С помощью *LIKE* можно сравнивать различные типы данных, но разные типы данных по-разному хранят шаблоны. В частности, будьте внимательны с отличиями между типами *CHAR*, *VARCHAR* и *DATE*.
- *LIKE* может помешать использованию индексов или привести к менее оптимальному использованию индекса, нежели простое сравнение.

Стандарт ANSI содержит два групповых символа, поддерживаемых всеми платформами:

%

Соответствует любой строке.

_ *(подчеркивание)*

Соответствует любому одиночному символу.

Первый запрос в следующем примере возвращает список городов, в названии которых есть слово «ville». Второй запрос возвращает авторов, имена которых не Sheryl или Cheryl (или Aheryl, Bheryl, Dheryl и т. д):

```
SELECT * FROM authors
WHERE city LIKE '%ville%';
SELECT * FROM authors
WHERE au_fname NOT LIKE '_heryl';
```

На некоторых платформах поддерживаются дополнительные групповые символы. Они описываются в дальнейших разделах, посвященных отдельным платформам.

Фраза *ESCAPE* позволяет вам искать в строках групповые символы. С помощью *ESCAPE* вы можете выбрать управляющий символ – символ, который не используется в шаблоне строки. Любой групповой символ, которому предшествует управляющий символ, будет трактоваться как обычный символ. Например, мы можем с помощью следующего запроса просмотреть столбец **comments** таблицы **sales_detail**, чтобы узнать, упоминал ли кто-либо из клиентов недавно введенные скидки:

```
SELECT ord_id, comment
FROM sales_detail
WHERE comment LIKE '%~%%' ESCAPE '~'
```

В этом примере первый символ % является групповым, но второй символ % рассматривается просто как символ процента, так как перед ним идет управляющий символ.

Советы и хитрости

Основная польза оператора *LIKE* заключается в возможности использования групповых символов. Оператор *LIKE* возвращает булево значение *TRUE*, если при сравнении найдены одно или несколько совпадений.

При использовании *LIKE* важна чувствительность платформы к регистру символов. Например, SQL Server по умолчанию не чувствителен к регистру символов (хотя можно настроить и противоположное поведение), поэтому строки DAD и dad при сравнении будут считаться одинаковыми. Oracle чувствителен к регистру символов, поэтому строки DAD и dad считаются разными.[1] Вот пример, иллюстрирующий этот момент:

```
SELECT *
FROM authors
WHERE lname LIKE 'LARS%'
```

Этот запрос, будучи выполненным на SQL Server, вернет фамилии типа *larson* и *lars*, хотя шаблон для поиска задан в верхнем регистре. Oracle, однако, не вернет ни *larson*, ни *lars*, так как он выполняет сравнение с точностью до регистра символов.

MySQL

MySQL поддерживает оператор *LIKE* в соответствии со стандартом ANSI. В нем поддерживаются как оба групповых символа (% и _), так и фраза *ESCAPE*.

Также в MySQL есть специальные функции *REGEXP*, *RLIKE*, *NOT REGEXP* и *NOT RLIKE* для работы с регулярными выражениями. Начиная с версии 3.23.4 MySQL по умолчанию не чувствителен к регистру символов.

Oracle

Oracle поддерживает оператор *LIKE* в соответствии со стандартом ANSI. В нем поддерживаются как оба групповых символа (% и _), так и фраза *ESCAPE*. Синтаксис оператора *LIKE* в Oracle следующий:

```
WHERE выражение [NOT] {LIKE | LIKEC | LIKE2 |
  LIKE4} шаблон
[ESCAPE управляющий_символ]
```

Специфические элементы синтаксиса Oracle имеют следующие значения:

LIKEC

Использует полный набор символов *UNICODE*.

LIKE2

Использует *UNICODE USC2*.

LIKE4

Использует *UNICODE UCS4*.

---

[1] Аналогично можно настроить противоположное поведение. – *Прим. науч. ред.*

Так как Oracle чувствителен к регистру символов, то желательно и шаблон, и выражение оборачивать в функцию *UPPER*. Тем самым вы будете всегда сравнивать строки независимо от регистра символов.

PostgreSQL

PostgreSQL поддерживает оператор *LIKE* в соответствии со стандартом ANSI. В нем поддерживаются как оба групповых символа (% и _), так и фраза *ESCAPE*.

По умолчанию PostgreSQL чувствителен к регистру символов, однако в нем поддерживается функция *ILIKE* для регистронезависимого сравнения с шаблонами. Также вы можете использовать ~~ вместо *LIKE*, ~~* вместо *ILIKE* и *!~~* и *!~~** вместо *NOT LIKE* и *NOT ILIKE* соответственно. Эти операторы являются расширениями стандарта ANSI.

Например, следующие запросы функционально эквивалентны:

```
SELECT * FROM authors
WHERE city LIKE '%ville';

SELECT * FROM authors
WHERE city ~~ '%ville';
```

Так как шаблоны этих запросов написаны в нижнем регистре, то вы можете столкнуться с проблемами, – запросы не будут возвращать значения, хранимые в верхнем регистре, такие как *'BROWNSVILLE'*, *'NASHVILLE'* и *'HUNTSVILLE'*. Вы можете обойти эту проблему следующими способами:

```
-- Преобразуем значения в верхний регистр
SELECT * FROM authors
WHERE city LIKE UPPER('%ville');[1]

-- Используем регистронезависимый оператор
SELECT * FROM authors
WHERE city ~~* '%ville';
SELECT * FROM authors
WHERE city ILIKE '%ville';
```

В PostgreSQL также поддерживаются регулярные выражения по стандарту POSIX. Их рассмотрение выходит за рамки этой книги, поэтому за подробной информацией обращайтесь к документации.

SQL Server

SQL Server поддерживает оператор *LIKE* в соответствии со стандартом ANSI. В нем поддерживаются как оба групповых символа (% и _), так и фраза *ESCAPE*. Также дополнительно поддерживаются следующие групповые символы:

[ ]

Соответствует любому значению из указанного множества (например, *[abc]*) или диапазона (например, *[k-n]*).

[^]

Соответствует любому значению не из указанного множества или диапазона.

---

[1] А лучше так: «WHERE UPPER(city) LIKE UPPER('%ville')». – *Прим. науч. ред.*

Дополнительные групповые символы дают новые возможности. Например, вы можете найти всех авторов с фамилиями Carson, Carsen, Karson и Karsen:

```
SELECT * FROM authors
WHERE au_lname LIKE '[CK]ars[eo]n'
```

Либо вы можете найти все фамилии, заканчивающиеся на «*arsen*» или «*arson*», но не совпадающие с Larson и Larsen:

```
SELECT * FROM authors
WHERE au_lname LIKE '[A-Z^L]ars[eo]n'
```

Помните, что при использовании *LIKE* имеют значение все символы в шаблоне, включая пробелы в начале и конце строки.

См. также

SELECT
UPDATE
DELETE
WHERE

MERGE

Оператор *MERGE* является в некотором роде оператором *CASE* для DML. *MERGE* комбинирует *INSERT* и *UPDATE* в один оператор, включающий функциональность обоих.

По сути оператор *MERGE* сравнивает записи исходной и целевой таблицы. Если запись есть и в той и в другой таблице, то при заданных условиях запись целевой таблицы обновляется значениями записи исходной таблицы. Если же в целевой таблице нет записи из исходной таблицы, то такая запись вставляется в целевую таблицу. Оператор *MERGE* впервые появился в стандарте SQL2003.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается
PostgreSQL	Не поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
MERGE INTO {имя_объекта | подзапрос} [ [AS] псевдоним ]
USING источник [ [AS] псевдоним ]
ON условие_поиска
WHEN MATCHED
THEN UPDATE SET столбец = { выражение | DEFAULT }[, ...]
WHEN NOT MATCHED
THEN INSERT [( столбец[, ...] )] VALUES ( выражение[, ...] )
```

Ключевые слова

MERGE INTO {имя_объекта | подзапрос}

Указывает для оператора *MERGE* целевой объект, которым может являться таблица, обновляемое представление или вложенная таблица.

[AS] псевдоним

Указывает необязательный псевдоним для целевой таблицы.

USING источник

Указывает источник, которым может являться таблица, представление или подзапрос.

ON условие_поиска

Указывает условие, по которому сопоставляются строки из таблицы-источника и целевой таблицы. Здесь используется такой же синтаксис, как и во фразе *ON* оператора *JOIN*. Например, при работе с таблицами new_hire_emp и emp, фраза *ON* могла бы выглядеть как *ON emp.emp_id = new_hire_emp.emp_id*.

WHEN MATCHED THEN UPDATE SET столбец= { выражение | DEFAULT }[, ...]

Указывает правила обновления столбцов целевой таблицы в том случае, если найдена строка целевой таблицы, соответствующая строке таблицы-источника.

WHEN NOT MATCHED THEN INSERT [(столбец[, ...])] VALUES (выражение[, ...]

Определяет значения строки, вставляемой в целевую таблицу, если для строки исходной таблицы не найдено соответствующей строки в целевой таблице.

Общие правила

Правила использования оператора *MERGE* достаточно просты:

- Фразы *WHEN MATCHED* и *WHEN NOT MATCHED* обязательны, но каждая из них в операторе может быть указана не более одного раза.
- В качестве целевого объекта оператора *MERGE* может выступать таблица, обновляемое представление или обновляемый подзапрос.
- Если источником оператора *MERGE* является подзапрос, то его необходимо заключить в скобки.
- Условие поиска во фразе *ON* не может содержать ссылки на хранимые процедуры и пользовательские функции.
- Условие поиска во фразе *ON* может содержать несколько элементов, соединенных операторами *AND* и *OR*.
- Если во фразе *WHEN NOT MATCHED* не указан список столбцов, то подразумевается использование всех столбцов целевой таблицы в порядке их следования в таблице.

Все остальные правила оператора *MERGE* очевидны. Например, столбцы, указанные во фразе *WHEN MATCHED*, должны быть обновляемыми.

Советы и хитрости

Оператор *MERGE* иногда называют оператором «upsert», так как он позволяет обновить существующие записи (*UPDATE*) и вставить отсутствующие (*INSERT*) одной командой.

Самое сложное в использовании оператора *MERGE* – это осознать и привыкнуть к идее условного выполнения либо *INSERT*, либо *DELETE*.

Представим себе, что у нас есть две таблицы – **EMP** и **NEW_HIRE**. Таблица **EMP** содержит информацию обо всех сотрудниках компании, успешно прошедших 90-дневный испытательный срок при начале работы. Записи в таблице **EMP** могут также иметь различные статусы: активная, неактивная, завершенная. Наем каждого нового сотрудника записывается в таблицу **NEW_HIRE**, а через 90 дней запись переносится в таблицу **EMP**. Однако компания каждое лето нанимает стажеров из колледжей, поэтому возможна ситуация, что нанятые сотрудники уже есть в таблице **EMP** с прошлого года. На псевдокоде бизнес-задачу можно описать следующим образом:

```
Цикл по каждой записи таблицы NEW_HIRE
  Найти соответствующую запись в таблице EMP
  Если запись в таблице EMP найдена
    Обновить запись в таблице EMP
  Иначе
    Вставить запись в таблицу EMP
  Конец
Конец цикла
```

Мы могли бы написать достаточно сложную хранимую процедуру, проверяющую все записи таблицы **NEW_HIRE** и выполняющую *UPDATE* для бывших стажеров и *INSERT* для полностью новых сотрудников. Однако оператор *MERGE* решает эту задачу намного проще:

```
MERGE INTO emp AS e
  USING (SELECT * FROM new_hire) AS n
  ON e.empno = n.empno
WHEN MATCHED THEN
  UPDATE SET
    e.ename = n.ename,
    e.sal = n.sal,
    e.mgr = n.mgr,
    e.deptno = n.deptno
WHEN NOT MATCHED THEN
  INSERT ( e.empno, e.ename, e.sal, e.mgr, e.deptno )
  VALUES ( n.empno, n.ename, n.sal, n.mgr, n.deptno );
```

Как видите, оператор *MERGE* очень полезен для загрузки данных.

MySQL

MySQL не поддерживает оператор *MERGE*. Однако вы можете использовать функционально и синтаксически похожий оператор *REPLACE* для тех же целей.

Oracle

Oracle поддерживает оператор *MERGE* с небольшими расширениями, которые становятся очевидны при простом сравнении синтаксиса оператора в Oracle и в стандарте ANSI:

```
MERGE INTO [схема.]{имя_объекта | подзапрос} [псевдоним]
USING [схема.]источник [псевдоним]
ON ( условие_поиска )
WHEN MATCHED THEN
UPDATE SET столбец = { выражение | DEFAULT }[, ...]
WHEN NOT MATCHED THEN
INSERT ( столбец[, ...] ) VALUES ( выражение[, ...]
[LOG ERRORS [INTO [схема.]таблица] [( выражение )]
[REJECT LIMIT { целое_число | UNLIMITED}]]
```

Отличия между стандартом ANSI и реализацией Oracle включают:

- В Oracle при указании псевдонимов для таблиц не используется ключевое слово *AS*.
- Условие поиска в *ON* должно быть взято в скобки.
- В Oracle во фразе *WHEN NOT MATCHED* необходимо использовать список столбцов, необязательный по стандарту ANSI.

Oracle поддерживает журналирование ошибок оператора MERGE с помощью фразы *LOG ERRORS [INTO [схема.]таблица] [(выражение)] [REJECT LIMIT {целое_число |UNLIMITED}]*. *INTO* указывает таблицу, в которую записывается информация об ошибках. Если имя таблицы не указано, то Oracle использует таблицы с названием **ERR$_** плюс 25 первых символов названия таблицы, в которую выполняется вставка. Вы можете указать любое выражение (например, *TO_CHAR(SYSDATE)*), которое вы хотите вставлять в таблицу ошибок. *REJECT LIMIT* позволяет задать порог числа ошибок, при достижении которого выполнение оператора *INSERT* будет остановлено. (Обратите внимание, что вы не сможете журналировать значения столбцов типов *LONG*, *LOB* и объектных типов.)

Примеры *MERGE* приводятся в предыдущих разделах «Общие правила» и «Советы и хитрости».

PostgreSQL

Не поддерживается.

SQL Server

SQL Serevr поддерживает свой вариант оператора *MERGE* начиная с версии SQL Server 2008. В значительной степени реализация соответствует стандарту ANSI. Синтаксис оператора следующий:

```
[WITH табличное_выражение[, ...]]
MERGE [TOP ( число ) [PERCENT]]
[INTO] {имя_объекта | подзапрос} [ [AS] псевдоним ]
USING ( источник ) [ [AS] псевдоним ]
ON условие_поиска
WHEN MATCHED
  THEN { UPDATE SET столбец = { выражение | DEFAULT }[, ...]
```

```
      | DELETE }
  WHEN [{[TARGET] | SOURCE}] NOT MATCHED
    THEN INSERT [( столбец[, ...] )] [DEFAULT] VALUES
      (выражение[, ...] )
  [OUTPUT выражение [INTO {@табличная_переменная | таблица}
    [( список_столбцов[, ...] )]]]
```

где:

WITH табличное_выражение

Объявляет именованный временный набор строк, определяемый оператором *SELECT*.

TOP (число) [PERCENT]

Указывает точное число строк или процент от общего числа строк, которые должны быть вставлены. Если число указано не литералом, а выражением, то оно должно быть заключено в скобки. Если используется ключевое слово *PERCENT*, то выражение должно иметь тип *FLOAT* и иметь значение от 0 до 100. Если *PERCENT* не используется, то выражение должно иметь тип *BIGINT*.

WHEN {[TARGET] | SOURCE} NOT MATCHED

Определяет поведение оператора для строк, не найденных либо в таблице-источнике (*SOURCE*), либо в целевой таблице (*TARGET*). Если не указано ни *SOURCE*, ни *TARGET*, то по умолчанию подразумевается *TARGET*, то есть фраза *WHEN NOT MATCHED* эквивалентна *WHEN TARGET NOT MATCHED*. Фразу *WHEN SOURCE NOT MATCHED* следует использовать с дополнительными условиями поиска, все из которых должны быть удовлетворены. В противном случае следует использовать *WHEN TARGET NOT MATCHED*. Вы можете использовать две фразы *WHEN SOURCE NOT MATCHED*, определяющих разные условия для операций *DELETE* и *UPDATE*.

OUTPUT выражение INTO {@табличная_переменная | таблица} [(список_столбцов[,...])]

Возвращает строки, вставленные командой (по умолчанию MERGE возвращает лишь число вставленных или обновленных строк), и сохраняет их либо в табличную переменную, либо в таблицу. Таблица, в которую возвращаются вставленные сроки, не должна иметь триггеров, внешних ключей и ограничений *CHECK*. Столбцы в этой таблице должны соответствовать указанному во фразе списку столбцов, либо, если список столбцов не указан, всем столбцам целевой таблицы.

Оператор *MERGE* не только позволяет указать список столбцов для обновления или вставки. Для столбца может использоваться следующий синтаксис *{DELETED | INSERTED | таблица_источник}.{* | столбец}*, либо можно использовать ключевое слово *$ACTION* (*$ACTION* автоматически заменяется на *INSERT*, *UPDATE* или *DELETE* в зависимости от реально выполненной команды). SQL Server также поддерживает псевдотаблицы **inserted** и **deleted**, которые используются в триггерах для поддержки целостности при выполнения *MERGE*. Значения столбца для фразы *OUTPUT* можно брать из псевдотаблиц **inserted** и **deleted**. Также, если на целевой таблице созданы триггеры типа *AFTER* на действия *INSERT*, *UPDATE* и *DELETE*, то они будут запускаться соответствующими действиями оператора *MERGE*.

См. также

INSERT
JOIN
SELECT
SUBQUERY
UPDATE

OPEN

Оператор *OPEN* является одним из четырех операторов (наряду с *FETCH*, *DECLARE* и *CLOSE*) для работы с курсорами. Курсоры позволяют вместо одновременной обработки множества строк обрабатывать каждую строку отдельно. *OPEN* открывает курсор, объявленный ранее при помощи оператора *DECLARE CURSOR*.

Использование курсоров очень полезно, так как реляционные СУБД оперируют множествами строк, а большинство клиентских приложений работают с данными построчно. Курсоры дают возможность обрабатывать в каждый момент времени только одну запись, что и требуется для клиентских программ.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Не поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
OPEN имя_курсора
```

Ключевые слова

OPEN имя_курсора

Открывает указанный курсор, предварительно объявленный оператором *DECLARE CURSOR*.

Общие правила

Основными шагами работы с курсором являются:

1. Создание курсора (команда *DECLARE*).
2. Открытие курсора (команда *OPEN*).
3. Работа с курсором (команда *FETCH*).
4. Закрытие курсора (команда *CLOSE*).

Выполняя эти шаги, можно создать результат, аналогичный команде *SELECT*, и при этом работать с каждой строкой результата отдельно.

В следующем примере мы открываем курсор для получения имен всех авторов из таблицы **authors**:

```
DECLARE employee_cursor CURSOR FOR
  SELECT au_lname, au_fname
  FROM pubs.dbo.authors
  WHERE lname LIKE 'K%'
OPEN employee_cursor
FETCH NEXT FROM employee_cursor
BEGIN
  FETCH NEXT FROM employee_cursor
END
CLOSE employee_cursor
```

Советы и хитрости

Наиболее распространенной ошибкой, связанной с оператором *OPEN*, являются незакрытые курсоры. Хотя в этом разделе мы рассматриваем оператор *OPEN* отдельно, он всегда должен использоваться совместно с *DECLARE*, *FETCH* и *CLOSE*. Если вы забыли закрыть курсор, то вы не получите сообщения об ошибке, но открытый курсор будет продолжать удерживать блокировки и использовать оперативную память и другие ресурсы сервера. Незакрытые курсоры могут привести к проблемам, аналогичным утечке памяти. Если вы больше не используете курсор, он все равно занимает память, которой сервер базы данных мог бы найти лучшее применение. Стоит потратить немного дополнительного времени и удостовериться, что все открытые курсоры корректно закрываются.

Курсоры часто используются в хранимых процедурах для пакетной обработки данных. Причиной является необходимость выполнять какие-то действия над каждой строкой в отдельности, а не над всем множеством сразу. Но из-за того, что курсоры работают с отдельными записями, а не с множествами записей, они часто оказываются медленнее, чем другие способы доступа к данным. Важно критически анализировать необходимость курсоров в каждой ситуации. Многие задачи, такие как заковыристые *DELETE* или сложные *UPDATE*, можно решать с помощью грамотного использования *JOIN* и *WHERE*, а не с помощью курсоров.

MySQL

MySQL поддерживает оператор OPEN полностью в соответствии со стандартом.

Oracle

Oracle полностью поддерживает стандарт ANSI, а также позволяет при открытии курсора передавать в него параметры:

```
OPEN имя_курсора [ параметр1[, ...]]
```

PostgreSQL

PostgreSQL не поддерживает оператор *OPEN CURSOR*. В PostgreSQL курсоры неявно открываются при их объявлении.

SQL Server

В дополнение к стандартному оператору *OPEN*, в SQL Server можно открывать «глобальные» курсоры, используя следующий синтаксис:

```
OPEN [GLOBAL] имя_курсора
```

где:

имя_курсора

Указывает имя курсора (или строковую переменную, содержащую это имя), предварительно объявленного оператором *DECLARE CURSOR*.

GLOBAL

Позволяет нескольким пользователям обращаться к курсору, даже если им явно не было выдано разрешение на этот курсор. Если ключевое слово *GLOBAL* не используется, то подразумевается создание локального курсора.

В SQL Server можно создавать курсоры разного типа. Если курсор объявлен с параметрами *INSENSITIVE* и *STATIC*, то при его открытии создается временная таблица, в которой хранится результирующее множество строк курсора. Аналогично, если при объявлении курсора используется *KEYSET*, то при открытии создается временная таблица для хранения множества ключей.

См. также

CLOSE
DECLARE
FETCH
SELECT

ORDER BY

Фраза *ORDER BY* используется для определения порядка сортировки результата, возвращаемого оператором *SELECT*.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

```
ORDER BY {выражение [COLLATE схема_упорядочения]
  [ASC | DESC]}[, ...]
```

Ключевые слова

ORDER BY

Указывает порядок, в котором должны возвращаться строки результата запроса. Без *ORDER BY* вам не следует ожидать вывода строк в определенном порядке, даже если вы используете *GROUP BY* и результат выглядит отсортированным.

выражение

Указывает элемент запроса, определяющий порядок сортировки результирующего множества. Вы можете указать несколько выражений для сортиров-

ки. Обычно выражение содержит имя или псевдоним столбца запроса, однако возможны и более сложные выражения, например *salary*1.02*. Стандарт SQL92 позволял указывать порядковые номера столбцов, но в SQL2003 эта функциональность была запрещена и ее не следует использовать в запросах.

COLLATE схема_упорядочения

Позволяет использовать в *ORDER BY* схему упорядочения, отличную от используемой по умолчанию.

ASC | DESC

Указывает, что сортировка результирующего множества должна производиться по возрастанию (*ASC*) или убыванию (*DESC*) *выражения*.

Общие правила

Во фразе *ORDER BY* можно ссылаться на столбцы, перечисленные во фразе *SELECT*, причем желательно использовать псевдонимы:

```
SELECT au_fname AS first_name, au_lname AS last_name
FROM authors
ORDER BY first_name, last_name
```

ORDER BY сортирует данные поочередно по каждому выражению слева направо. То есть результирующее множество сначала сортируется по первому указанному столбцу; строки, имеющие одинаковые значения в первом столбце, сортируются по второму столбцу; строки, имеющие одинаковые значения в первом и втором столбцах, сортируются по третьему и т. д.

Атрибуты *ASC/DESC* и схема упорядочения определяются индивидуально для каждого столбца и не зависят от других столбцов. Поэтому вы можете отсортировать результат по возрастанию значений одного столбца, а затем по убыванию значений другого столбца:

```
SELECT au_fname AS first_name, au_lname AS last_name
FROM authors
ORDER BY au_lname ASC, au_fname DESC
```

Значения NULL при сортировке всегда идут подряд (то есть считаются равными). В зависимости от платформы пустые значения могут оказаться в начале отсортированного списка или в конце. Данный запрос на SQL Server:

```
SELECT title, price
FROM titles
ORDER BY price, title
```

вернет следующий результат (результат отредактирован для краткости):

```
title price
--------------------------------------------------- -------
Net Etiquette                                       NULL
The Psychology of Computer Cooking                  NULL
The Gourmet Microwave                               2.9900
You Can Combat Computer Stress!                     2.9900
Life Without Fear                                   7.0000
```

```
Onions, Leeks, and Garlic: Cooking Secrets of the Me 20.9500
Computer Phobic AND Non-Phobic Individuals: Behavior 21.5900
But Is It User Friendly?                            22.9500
```

Вы можете изменить расположение пустых значений в отсортированном списке, используя параметры *ASC* и *DESC*. Но при этом, очевидно, изменяется порядок сортировки всех непустых значений.

Стандарт ANSI позволяет использовать в *ORDER BY* выражения, основанные на столбцах, не присутствующих в списке значений *SELECT*. Рассмотрим, например, следующий запрос:

```
SELECT title, price
FROM titles
ORDER BY title_id
```

Результат этого запроса будет отсортирован по значению столбца **title_id**, хотя сам этот столбец не извлекается.

Советы и хитрости

При использовании операторов работы с множествами (*UNION*, *EXCEPT*, *INTERSECT*) только последний оператор *SELECT* может содержать *ORDER BY*. Нельзя использовать *ORDER BY* в подзапросах любого типа.

Некоторые особенности *ORDER BY* из стандарта SQL92 были отменены в SQL2003. Вам следует избегать следующего:

Использование псевдонимов таблиц

Например, *ORDER BY e.emp_id* следует заменить *ORDER BY emp_id*. При неоднозначности имен столбцов используйте псевдонимы столбцов.

Использование номеров столбцов

Используйте псевдонимы столбцов.

Вы можете сортировать результат не только по значениям столбцов, но и по выражениям на базе столбцов, и даже по литералам:

```
SELECT SUBSTRING(title,1,55) AS title, (price * 1.15) as price
FROM titles
WHERE price BETWEEN 2 and 19
ORDER BY price, title
```

При сортировке по выражениям из списка *SELECT* вам следует использовать псевдонимы выражений, чтобы упростить *ORDER BY*.

MySQL

MySQL поддерживает стандарт ANSI за исключением опции *COLLATE*.

Не следует сортировать по столбцам типа *BLOB*, так как при этом в сортировке будет участвовать только некоторое количество первых байт значений, определенное *MAX_SORT_LENGTH*. При сортировке по возрастанию значения *NULL* идут первыми, при сортировке по убыванию – последними.

Oracle

Oracle поддерживает стандарт ANSI за исключением фразы *COLLATE*, а также предлагает дополнительные опции *SIBLINGS* и *NULLS {FIRST|LAST}*. Синтаксис *ORDER BY* в Oracle следующий:

```
ORDER [SIBLINGS] BY
  {выражение [ASC | DESC] [NULLS {FIRST | LAST}]}[, ...]
```

где:

ORDER [SIBLINGS] BY выражение

Сортирует результирующее множество по указанному выражению. Выражение может быть именем столбца, псевдонимом, номером столбца или любым другим выражением. Фраза *ORDER SIBLINGS BY* используется в иерархических запросах и указывает порядок сортировки узлов-братьев.

NULLS {FIRST | LAST}

Позволяет явно указать расположение пустых значений в отсортированном множестве – либо в начале (*NULLS FIRST*), либо в конце (*NULLS LAST*). По умолчанию пустые значения идут в конце при возрастающей сортировке и в начале – при убывающей сортировке.

Вы можете эмулировать опцию *COLLATE* в сессии, используя функцию *NLSSORT* или параметр *NLS_SORT*. Для того чтобы эмулировать *COLLATE* для всех сессий, установите соответствующее значение инициализационного параметра *NLS_SORT* или *NLS_LANGUAGE*.

Oracle продолжает поддерживать некоторые отмененные функции стандарта SQL92, например указание номеров столбцов в *ORDER BY*. Однако не следует выполнять сортировку по столбцам типов *LOB*, *VARRAY* и вложенным таблицам.

PostgreSQL

PostgreSQL поддерживает стандарт ANSI за исключением фразы *COLLATE*. В *ORDER BY* можно также использовать расширение *USING*:

```
ORDER BY {выражение [ASC | DESC | USING оператор]}[, ...]
```

где:

USING оператор

Указывает используемый при сортировке оператор сравнения. Вы можете использовать операторы >, <, =, >=, <= и т. д. Сортировку по возрастанию можно выполнить, указав *USING* <, а сортировку по убыванию при помощи *USING* >.

NULL при сортировке считается наибольшим значением. Поэтому значения NULL будут идти в конце при сортировке по возрастанию, и в начале – при сортировке по убыванию.

SQL Server

SQL Server полностью поддерживает стандарт ANSI, включая опцию *COLLATE*. Например, следующий запрос возвращает имена авторов из таблицы **authors** в порядке, задаваемом схемой упорядочения *SQL_Latin1*:

```
SELECT au_fname
FROM authors
ORDER BY au_fname
COLLATE SQL_Latin1_general_cp1_ci_as
```

SQL Server продолжает поддерживать некоторые отмененные функции стандарта SQL92, например указание номеров столбцов в *ORDER BY*. NULL при сортировке считается наибольшим значением. Не следует выполнять сортировку по значениям типа *TEXT*, *NTEXT* и *IMAGE*.

См. также

SELECT

RELEASE SAVEPOINT

Оператор *RELEASE SAVEPOINT* удаляет точку сохранения, объявленную ранее в текущей транзакции.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Не поддерживается
PostgreSQL	Поддерживается
SQL Server	Не поддерживается

Синтаксис SQL2003

```
RELEASE SAVEPOINT имя_контрольной_точки
```

Ключевые слова

имя_контрольной_точки

Указывает контрольную точку, созданную ранее в этой же транзакции при помощи оператора *SAVEPOINT*. Имя контрольной точки должно быть уникально в пределах транзакции.

Общие правила

Используйте оператор *RELEASE SAVEPOINT* для удаления контрольной точки транзакции. Все контрольные точки, созданные после указанной, также удаляются.

Для иллюстрации контрольных точек мы вставим в таблицу несколько строк, создадим контрольную точку, а затем удалим ее:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1);
SAVEPOINT first_savepoint;
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('277-27-2777', 'Fudd', 'E.P.', 1);
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('366-36-3636', 'Duck', 'P.J.', 1);
```

```
RELEASE SAVEPOINT first_savepoint;
COMMIT;
```

В этом примере контрольная точка **first_savepoint** удаляется, а затем все три записи вставляются в таблицу **authors**.

В следующем примере мы выполним те же действия, но с большим числом контрольных точек:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1);
SAVEPOINT first_savepoint;
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('277-27-2777', 'Fudd', 'E.P.', 1);
SAVEPOINT second_savepoint;
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('366-36-3636', 'Duck', 'P.J.', 1);
SAVEPOINT third_savepoint;
RELEASE SAVEPOINT second_savepoint;
COMMIT;
```

В этом примере удаление контрольной точки **second_savepoint** вызывает также удаление контрольной точки **third_savepoint**, так как она была создана после **second_savepoint**.

После удаления контрольной точки ее имя может быть использовано повторно.

Советы и хитрости

Операторы *COMMIT* и *ROLLBACK* вызывают удаление всех созданных в транзакции контрольных точек. Оператор *ROLLBACK TO SAVEPOINT* возвращает транзакцию в состояние указанной контрольной точки; все контрольные точки, созданные после указанной, аннулируются.

MySQL

Поддерживает стандартный синтаксис SQL3.

Oracle

Не поддерживается.

PostgreSQL

Поддерживает стандартный синтаксис SQL3, хотя слово *SAVEPOINT* необязательно:

```
RELEASE [SAVEPOINT] имя_контрольной_точки
```

SQL Server

Не поддерживается.

См. также

ROLLBACK
SAVEPOINT

RETURN

Оператор *RETURN* завершает выполнение вызванной из SQL функции или хранимой процедуры и возвращает результат работы.

В некоторых СУБД вместо определенного стандартом оператора *RETURN* нужно использовать оператор *RETURNS*.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Поддерживается
PostgreSQL	Поддерживается с ограничениями
SQL Server	Поддерживается

Синтаксис SQL2003

```
RETURN возвращаемое_значение| NULL
```

Ключевые слова

возвращаемое_значение

Указывает возвращаемое процедурой или функцией значение. Допустимо использование различного типа значений.

NULL

Завершает функцию, не возвращая результирующее значение.

Общие правила

Используйте оператор *RETURN* в процедурном коде для завершения работы. Например, вы могли бы создать функцию, в которой входной параметр используется в сложном выражении *CASE* и из функции возвращается одиночное легко читаемое значение.

Советы и хитрости

Хотя *RETURN* является отдельным оператором в SQL, он очень тесно переплетается с операторами *CREATE FUNCTION/PROCEDURE* и почти всегда используется в них. Посмотрите разделы, посвященные реализации каждого из этих двух операторов в различных платформах, чтобы найти информацию о *RETURN* в контексте этих операторов.

MySQL

MySQL поддерживает стандартный синтаксис оператора *RETURN*, за исключением опции *NULL*:

```
RETURN возвращаемое_значение
```

Oracle

Oracle поддерживает для оператора *RETURN* стандартный синтаксис ANSI, за исключением ключевого слова *NULL*. (Oracle поддерживает возврат пустых значений, но не по синтаксису ANSI.) Oracle поддерживает *RETURN* только в пользовательских функциях и операторах. Возвращаемое значение в *CREATE OPERATOR* не может иметь тип *LONG*, *LONG RAW* и *REF*. Пользовательские функции поддерживают внутри PL/SQL булев тип данных, но вы не можете вернуть булево значение в вызывающий sql-оператор. Для хранения булевых значений используйте тип *INT* (для значений 0 и 1) либо *VARCHAR2* (для значений '*TRUE*' и '*FALSE*').

В следующем примере мы создаем функцию, возвращающую значение переменной **proj_rev** вызывающему процессу:

```
CREATE FUNCTION project_revenue (project IN varchar2)
RETURN NUMBER
AS
  proj_rev NUMBER(10,2);
BEGIN
  SELECT SUM(DECODE(action,'COMPLETED',amount,0) -
    SUM(DECODE(action,'STARTED',amount,0) +
    SUM(DECODE(action,'PAYMENT',amount,0)
  INTO proj_rev
  FROM construction_actions
  WHERE project_name = project;
  RETURN (proj_rev);
END;
```

PostgreSQL

PostgreSQL поддерживает для оператора *RETURN* стандартный синтаксис ANSI, за исключением ключевого слова *NULL*:

```
RETURNS возвращаемое_значение
```

PostgreSQL позволяет создавать пользовательские функции либо на SQL, либо на C/C++. Хранимые процедуры в PostgreSQL не поддерживаются, но вы можете эмулировать их с помощью функции. В этом обсуждении нас интересуют исключительно функции, написанные на SQL.

Возвращаемое значение может иметь базовый тип, сложный тип, тип *SETOF*, тип *OPAQUE* либо тип существующего столбца таблицы. Модификатор *SETOF* используется в *RETURNS* для возврата не одного значения, а множества значений одного типа. Модификатор *OPAQUE* означает, что *RETURNS* на самом деле не возвращает значение. *OPAQUE* используется только в триггерах.

SQL Server

SQL Server поддерживает *RETURN* в соответствии со следующим синтаксисом:

```
RETURN [целочисленное_значение]
```

Оператор *RETURN* обычно используется в хранимых процедурах и пользовательских функциях. Он позволяет незамедлительно и полностью прервать вы-

полнение подпрограммы и, при необходимости, вернуть значение. Хранимые процедуры в SQL Server неявно возвращают 0, если другое значения не указано в *RETURN* явно. Любые операторы, следующие за *RETURN*, игнорируются.

Оператор *RETURN* в следующем примере возвращает целое число с результатом вычислений:

```
CREATE FUNCTION metric_volume
-- На вход подаются размеры в сантиметрах
  (@length decimal(4,1),
   @width decimal(4,1),
   @height decimal(4,1) )
RETURNS decimal(12,3) -- Кубические сантиметры
AS
BEGIN
  RETURN ( @length * @width * @height )
END
GO
```

В этом примере создается функция, вычисляющая объем и возвращающая его вызывающему объекту.

См. также

CREATE/ALTER FUNCTION/PROCEDURE
CREATE/ALTER TRIGGER

REVOKE

Оператор *REVOKE* существует в двух видах. Первый вид оператора *REVOKE* предназначен для отзыва у пользователя или роли привилегий на выполнение определенных операторов. Второй вид оператора *REVOKE* предназначен для отзыва привилегий на объекты и ресурсы базы данных.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

Общий синтаксис оператора *REVOKE* имеет следующий вид:

```
REVOKE { [специальные_опции] |
   {привилегия[, ...] | роль[, ...]} }
ON имя_объекта_базы_данных
FROM владелец_привилегии[, ...]
[GRANTED BY {CURRENT_USER | CURRENT_ROLE}]
{CASCADE | RESTRICT}
```

Ключевые слова

специальные_опции

Можно указывать одну из трех следующих опций:

GRANT OPTION FOR

Отзывает у пользователя опцию *WITH GRANT OPTION*, то есть пользователь больше не может передавать привилегии другим пользователям. Сами привилегии остаются в силе. (Эта опция применяется только к привилегиям, но не к ролям.)

HIERARCHY OPTION FOR

Отзывает у пользователя опцию *WITH HIERARCHY OPTION*, которая позволяет выполнять оператор *SELECT* не только для таблицы, но и для ее подтаблиц. (Эта опция применяется только к привилегиям, но не к ролям.)

ADMIN OPTION FOR

Отзывает опцию, которая позволяет пользователю назначать роль другим пользователям. (Эта опция применяется только к привилегиям, но не к ролям.)

привилегия

Отзывает привилегии на различные операторы. Привилегии можно комбинировать произвольным образом.

ALL PRIVILEGES

Отзывает все привилегии, выданные указанному пользователю на указанный объект. Применение этой опции не рекомендуется, так как провоцирует небрежность при работе с привилегиями.

EXECUTE

Отзывает привилегии на выполнение хранимой процедуры, пользовательской функции или метода.

SELECT | INSERT | UPDATE | DELETE

Отзывает привилегии на выполнение соответствующего оператора на указанный объект базы данных, такой как таблица или представление. Для привилегий *UPDATE*, *INSERT* и *SELECT* вы можете указать список столбцов таблицы, которыми будет ограничено выполняемое действие.

REFERENCES

Отзывает у пользователя привилегии на создание любых ограничений, ссылающихся на указанный объект базы данных как на родительский. Вы можете указать список столбцов, привилегии на которые будут отозваны.

TRIGGER

Отзывает у пользователя привилегии на создание триггеров на указанные таблицы. Побочным эффектом такого оператора *REVOKE* служит удаление уже существующих триггеров, зависящих от этой привилегии.

UNDER

Отзывает привилегии на создание подтипов и типизированных таблиц.

USAGE

Отзывает привилегии на использование доменов, пользовательских типов, кодировок, схем упорядочения и переводов.

роль

Отзывает определенную роль у пользователя или роли, указанных во *FROM*. Например, администратор базы данных мог бы создать роль *Reporter*, имеющую доступ на чтение к нескольким таблицам. Когда эта роль назначается пользователю, пользователь получает доступ к соответствующим таблицам. Детальная информация приводится в разделе, посвященном оператору *GRANT*.

ON имя_объекта_базы_данных

Отзывает привилегии на указанный объект базы данных. SQL2003 не поддерживает системные привилегии, но такая поддержка есть во многих реализациях. (Подфраза *ON* не используется при отзыве системных привилегий и ролей.) В качестве *имени_объекта_базы_данных* могут использоваться следующие фразы:

```
{ [TABLE] имя_объекта | DOMAIN имя_объекта |
COLLATION имя_объекта |
CHARACTER SET имя_объекта | TRANSLATION имя_объекта |
TYPE имя_объекта | [SPECIFIC] {ROUTINE | FUNCTION |
PROCEDURE | METHOD}
имя_объекта }
```

FROM владелец_привилегии

Указывает пользователя или роль, которые теряют отзываемые привилегии. Слово *PUBLIC* используется для отзыва привилегий у роли *PUBLIC* (глобального списка пользователей.) Можно перечислить через запятую несколько пользователей или ролей.

GRANTED BY {CURRENT_USER|CURRENT_ROLE }

Опционально используется для указания того, кем была изначально выдана привилегия. Например, при использовании *GRANTED BY CURRENT_USER* привилегия отзывается, только если она была выдана текущим пользователем. В противном случае оператор завершается с ошибкой. Если фраза не используется, то по умолчанию подразумевается *CURRENT_USER*.

RESTRICT|CASCADE

Ограничивает оператор *REVOKE* только указанной привилегией (*RESTRICT*) либо выполняет *REVOKE* и для всех зависимых привилегий (*CASCADE*), что также может привести к удалению объектов, зависящих от привилегий. Заметьте, что *REVOKE...RESTRICT* завершится с ошибкой при наличии зависимых привилегий. Зависимые привилегии должны быть отозваны в первую очередь.

Общие правила

Конкретную привилегию можно отозвать у определенного пользователя, используя *REVOKE имя_привилегии ON объект_базы_данных FROM имя_пользователя RESTRICT*. Объектную привилегию можно отозвать у всех пользователей, используя глобальный список пользователей *PUBLIC*.

Для отзыва привилегий у нескольких пользователей просто перечислите их через запятую. Также вы можете отозвать привилегии у нескольких пользователей и у роли *PUBLIC* одним оператором. (Роль *PUBLIC* детально описана в разделе, посвященном оператору *GRANT*. Когда *GRANT* выполняется для *PUBLIC*, привилегии получают все пользователи.)

При отзыве привилегий на таблицу операцию можно ограничить определенным набором столбцов: для этого после имени таблицы перечислите в скобках через запятую необходимые столбцы.

Советы и хитрости

Большинство платформ разделяют привилегии, выданные на уровне роли, и привилегии, выданные на уровне пользователя. (Помните, что роль – это группа привилегий.) Поэтому возможна ситуация, что пользователю, являющемуся членом двух ролей, трижды выдана одна и та же привилегия – один раз напрямую и два раза посредством ролей. В этом случае для лишения пользователя привилегии нужно отозвать у него эту привилегию напрямую, а также исключить пользователя из обеих ролей.

Важным аспектом оператора *REVOKE* (а равно и парного оператора *GRANT*) является то, что одни элементы оператора предназначены для работы с объектными привилегиями, а другие – для работы с ролями и административными привилегиями. Обычно эти элементы не используются одновременно. (Отличия между объектными привилегиями и административными привилегиями подробно объясняются в разделах, посвященных конкретным платформам.) Например, вы могли бы отозвать объектные привилегии у роли *salespeople* или у конкретных пользователей *e_fudd* и *prince_edward*:

```
REVOKE SELECT ON TABLE authors FROM salespeople RESTRICT;
REVOKE ALL PRIVILEGES ON TABLE sales FROM e_fudd,
  prince_edward CASCADE;
```

Многие табличные привилегии могут выдаваться на уровне столбцов. Отзываются эти привилегии следующим образом:

```
REVOKE INSERT(au_id, au_fname, au_lname)
  ON authors FROM e_fudd;
```

Фразы со специальными опциями (*GRANT OPTION*, *ADMIN OPTION* и *HIERARCHY OPTION*) используются для запрещения пользователям возможности передавать свои привилегии и роли другим пользователям. Однако отключение этих опций не запрещает пользователям продолжать использовать сами роли и привилегии. Например, мы можем запретить всем членам роли *manager* передавать привилегии на *UPDATE* определенной таблицы другим пользователям:

```
REVOKE GRANT OPTION FOR UPDATE ON sales FROM manager CASCADE;
```

Также с помощью *REVOKE* вы можете исключить пользователя из роли:

```
REVOKE manager FROM e_fudd CASCADE;
```

Хорошей практикой считается писать самодостаточные и логичные операторы *GRANT* и *REVOKE*. В частности, нужно избегать использования *CASCADE* и *ALL*

PRIVILEGES, так как их действие может быть не очевидно при рассмотрении одного оператора.

MySQL

MySQL поддерживает большинство ключевых слов из ANSI, за исключением *TRIGGER*, *EXECUTE* и *UNDER*. Также в MySQL есть полезная возможность, облегчающая глобальную выдачу и отзыв привилегий. Описание ключевых слов, не указанных далее, ищите выше – в разделе, посвященном стандарту ANSI:

```
REVOKE [ { ALL [PRIVILEGES] |
{SELECT | INSERT | UPDATE} [ (столбец[, ...]) ] | DELETE |
REFERENCES [ (столбец[, ...]) ] } |
{ [ USAGE ] | [{ALTER | CREATE | DROP}] | [FILE] | [INDEX] |
  [PROCESS] | [RELOAD] | [SHUTDOWN] |
  [CREATE TEMPORARY TABLES] | [LOCK TABLES] |
  [REPLICATION CLIENT] | [REPLICATION SLAVE] |
  [SHOW DATABASES] | [SUPER] }[, ...]
ON [тип_объекта] {имя_таблицы | * | *.* |имя_базы_данных.*}
FROM пользователь[, ...]
```

где:

ALL [PRIVILEGES]

Синоним для *ALL PRIVILEGES*. В MySQL включает все привилегии, применимые к типу объекта, указанному в *ON*, за исключением *GRANT OPTION* (то есть включает привилегии *SELECT*, *INSERT*, *UPDATE*, *DELETE* и т. д.)

SELECT | INSERT | UPDATE | DELETE

Отзывает, соответственно, привилегии на чтение, вставку, обновление и удаление данных из таблиц (или из определенных столбцов таблицы).

REFERENCES

Не реализовано.

USAGE

Отзывает все привилегии пользователя.

{ALTER | CREATE | DROP}

Отзывает привилегии на изменение, создание и удаление таблиц и других объектов базы данных.

FILE

Отзывает привилегии на чтение и запись файлов при помощи команд *SELECT INTO* и *LOAD DATA*.

INDEX

Отзывает привилегии на создание и удаление индексов.

PROCESS

Отзывает привилегии на просмотр выполняющихся процессов при помощи *SHOW FULL PROCESSLIST*.

RELOAD

Отзывает привилегии на выполнение команды *FLUSH*.

SHUTDOWN

Отзывает привилегии на использование команды *MYSQLADMIN SHUTDOWN* для завершения серверного процесса.

CREATE TEMPORARY TABLES

Отзывает привилегии на создание временных таблиц.

LOCK TABLES

Отзывает привилегии на блокировку с помощью оператора *LOCK TABLE* таблиц, на которые у пользователя есть привилегия *SELECT*.

REPLICATION CLIENT

Отзывает привилегии на просмотр метаданных о репликации.

REPLICATION SLAVE

Отзывает привилегии на чтение при репликации журналов мастер-сервера подчиненным сервером.

SHOW DATABASES

Отзывает привилегии на выполнение команды *SHOW DATABASES*.

SUPER

Отзывает у пользователя привилегии на выполнение команд: *CHANGE MASTER*, *KILL*, *MYSQLADMIN DEBUG*, *PURGE [MASTER] LOGS* и *SET GLOBAL*, а также на открытие новых соединений даже при достижении числом соединений значения *MAX_CONNECTIONS*.

*ON [тип_объекта] {имя_таблицы | * | *.* | имя_базы_данных.*}*

Отзывает привилегии либо на указанную таблицу, либо на все таблицы в текущей базе данных (*), либо на все таблицы во всех базах данных (*.*), либо на все таблицы в указанной базе данных. Можно дополнительно указать тип объекта (*TABLE*, *FUNCTION* или *PROCEDURE*), привилегии на который отзываются.

FROM

Отзывает привилегии у одного или нескольких пользователей, перечисленных через запятую. Имена пользователей также могут включать суффикс *@host*, если вы хотите ограничить отзыв привилегий указанным хостом.

REVOKE имеет некоторые ограничения по размерам: имена пользователей не могут быть длиннее 16 символов, а имена хостов, баз данных и объектов не могут быть длиннее 60 символов. Имена пользователей могут быть привязаны к определенным хостам. Более детальная информация приводится в разделе, посвященном оператору *GRANT*.

В MySQL привилегии на удаляемые объекты не отзываются автоматически: при удалении объекта все привилегии на этот объект нужно отзывать явно. Предположим, что вы удалили таблицу, но не отозвали привилегии на нее. Если вы позднее пересоздадите таблицу, то все привилегии на нее останутся в силе. Аналогично, все пользовательские привилегии остаются в силе при удалении пользователя.

Также важно знать, что MySQL поддерживает несколько уровней привилегий. Например, у пользователя может быть привилегия на таблицу, выданная напрямую, а также привилегия на эту же таблицу, выданная на уровне базы данных или сервера. Поэтому нужно быть внимательным при отзыве привилегий, потому что глобальные привилегии могут предоставлять пользователю доступ к объекту, к которому вы хотели запретить доступ, просто отозвав привилегии на этот конкретный объект.

Первая из следующих команд отзывает все привилегии на таблицу **sales** у пользователей *emily* и *dylan*, а вторая команда отзывает привилегию *SELECT* на все таблицы в текущей базе данных у пользователя *kelly*:

```
REVOKE ALL PRIVILEGES ON sales FROM emily, dylan;
REVOKE SELECT ON * FROM kelly;
```

Первая из двух следующих команд отзывает у пользователя *kelly* возможность передавать привилегии на таблицу **sales** другим пользователям, а вторая команда отзывает все привилегии пользователя *sam* в базе данных **pubs**:

```
REVOKE GRANT OPTION ON sales FROM kelly;
REVOKE ALL ON pubs.* FROM sam;
```

Oracle

Оператор *REVOKE* может использоваться не только для отзыва объектных и системных привилегий, но и для отключения пользователя от роли (или роли от другой роли). Информация об объектных и системных привилегиях, поддерживаемых оператором *REVOKE*, приводится в разделе, описывающем оператор *GRANT*.

Формы оператора *REVOKE* для отзыва объектных привилегий и для отзыва системных привилегий являются взаимоисключающими. Не пытайтесь выполнить оба действия одним оператором. Так как полный синтаксис для обоих вариантов операторов очень длинный, то за перечнем поддерживаемых объектных и системных привилегий обратитесь к описанию оператора *GRANT*.

В Oracle есть интересная особенность работы с привилегиями. В то время как большинство платформ поддерживают для каждого пользователя несколько контекстов привилегий (привилегии получаются пользователем напрямую и через членство в ролях), Oracle сделал следующий шаг в этом направлении. Конкретная привилегия на конкретный объект может быть выдана получателю привилегий несколькими пользователями. И каждый из этих пользователей должен будет отозвать привилегию у получателя, чтобы полностью лишить его этой привилегии.[1]

В Oracle оператор *REVOKE* имеет следующий синтаксис:

```
REVOKE { [объектная_привилегия][, ...]|
         [системная_привилегия] |
         [роль] }
```

[1] Это относится только к объектным привилегиям. – *Прим. науч. ред.*

```
[ON { [схема.][объект] |
[DIRECTORY имя_объекта_каталога] |
[JAVA [ { SOURCE | RESOURCE } ] [схема.][объект]] } ]
FROM {пользователь [, ...] | роль[, ...] | PUBLIC}
[CASCADE [CONSTRAINTS]] [FORCE];
```

где:

объектная_привилегия

Отзывает у пользователя или роли (или нескольких пользователей и ролей) одну или несколько указанных привилегий на один или несколько объектов базы данных:

ALL [PRIVILEGES]

Отзывает все выданные на объект привилегии. Так как *ALL* включает также *REFERENCES*, то при этом необходимо использовать фразу *CASCADE*. (Смотрите описание *REFERENCES* ниже.)

ALTER

Отзывает привилегии на изменение существующей таблицы[1] оператором *ALTER TABLE*.

EXECUTE

Отзывает привилегии на вызов хранимой процедуры, пользовательской функции или пакета.

INDEX

Отзывает привилегии на создание индексов по таблице.

REFERENCES

Отзывает привилегии на создание ограничений ссылочной целостности. Требует использования фразы *CASCADE CONSTRAINTS*.

SELECT | INSERT | DELETE | UPDATE

Отзывает привилегии на выполнение каждого типа операторов для объекта базы данных. Помните, что привилегия *DELETE* зависит от привилегии *SELECT*.

системная_привилегия

Отзывает одну или несколько указанных системных привилегий (таких как *CREATE TRIGGER* или *ALTER USER*) у пользователя или роли. Не используйте при отзыве системных привилегий фразу *ON*. Так как системных привилегий огромное множество, то мы не будем перечислять их еще раз. Список системных привилегий Oracle приводится в описании оператора *GRANT*.

роль

Исключает пользователя или роль из членов указанной роли.

ON

Указывает объект, привилегии на который отзываются. Объектом может быть таблица, представление, последовательность, хранимая процедура, пользовательская функция, пакет, материализованное представление, пользователь-

---

[1] И не только таблицы. Например, есть объектная привилегия ALTER SEQUENCE. – *Прим. науч. ред.*

ский тип, библиотека, индексный тип, пользовательский оператор или синоним для любого из перечисленных объектов. Например, вы могли бы отозвать привилегию *SELECT* на таблицу **scott.authors**. Если вы не указываете имя схемы, то Oracle по умолчанию использует текущую схему. Поддерживаются дополнительные ключевые слова для двух случаев:

DIRECTORY имя_объекта_каталога

Указывает объект типа *DIRECTORY*, привилегии на который отзываются.

JAVA [{SOURCE | RESOURCE }] [схема.][объект]

Указывает исходный код или ресурс объекта Java, на который отзываются привилегии.

FROM {пользователь | роль | PUBLIC}

Указывает пользователя или роль, у которых отзываются привилегии. *PUBLIC* используется для отзыва привилегий у роли *PUBLIC*. Через запятую можно перечислить несколько пользователей или ролей.

CASCADE [CONSTRAINTS]

Удаляет все ссылочные ограничения целостности, зависящие от отзываемых привилегий. Эта фраза необходима только при отзыве привилегий *REFERENCES* или *ALL*.

FORCE

Требуется для отзыва привилегии *EXECUTE* на объекты пользовательских типов, имеющие зависимости.

При отзыве привилегий у одного пользователя Oracle автоматически отзывает привилегии[1] также у всех пользователей, получивших привилегии от первого. В дополнение все объекты, зависящие от отозванных привилегий (например, хранимые процедуры, триггеры, представления, зависящие от привилегии *SELECT*), становятся невалидными.

Пользователи, имеющие системную привилегию *GRANT ANY ROLE*, могут отзывать любую роль. Оператор *REVOKE* позволяет отозвать только привилегии, явно выданные оператором *GRANT*, но не привилегии, доступные через роли или операционную систему. В этом случае вы должны оператором *REVOKE* отозвать привилегию у роли, при этом все пользователи – члены роли – потеряют эту привилегию.

Следующий пример демонстрирует отзыв роли у пользователя и отзыв системной привилегии у роли:

```
REVOKE read_only FROM sarah;
REVOKE CREATE ANY SEQUENCE,
CREATE ANY DIRECTORY FROM read_only;
```

Вот пример того, как отзывается привилегия *REFERENCES* с каскадным удалением ограничений:

```
REVOKE REFERENCES
ON pubs_new_york.emp
```

[1] Касается только объектных привилегий. – *Прим. науч. ред.*

```
FROM dylan
CASCADE CONSTRAINTS;
```

Наконец, в следующем примере на определенную таблицу выдаются все привилегии, а затем часть из них отзывается:

```
GRANT ALL PRIVILEGES ON emp TO dylan;
REVOKE DELETE, UPDATE ON emp FROM dylan;
```

PostgreSQL

PostgreSQL поддерживает базовые возможности оператора *REVOKE*, касающиеся, в основном, отзыва привилегий на таблицы, представления и последовательности. Опции *HIERARCHY OPTION FOR* и *ADMIN OPTION FOR* не поддерживаются. Синтаксис оператора следующий:

```
REVOKE [GRANT OPTION FOR]
{ привилегии | { ALL [PRIVILEGES] | {SELECT | INSERT |
  DELETE | UPDATE} | RULE | REFERENCES| TRIGGERS | CREATE |
  USAGE }[, ...] }
ON { [TABLE] | SEQUENCE | DATABASE | FUNCTION | LANGUAGE |
  SCHEMA | TABLESPACE]
имя_объекта[, ...] }
FROM {пользователь | GROUP группа | PUBLIC}[, ...]
[ {CASCADE | RESTRICT} ]
```

где:

REVOKE [GRANT OPTION FOR] *привилегии*

Отзывает привилегии на различные операторы, которые можно комбинировать любым образом. *GRANT OPTION FOR* отзывает привилегию на передачу пользователем своих привилегий другим пользователям.

ALL [PRIVILEGES]

Отзывает все привилегии, выданные пользователю или роли на указанный объект. Эту опцию использовать не рекомендуется, так как она провоцирует небрежность в работе с привилегиями.

SELECT | INSERT | DELETE | UPDATE

Отзывает привилегии на выполнение соответствующих операторов. Вы можете указать после таблицы в скобках список столбцов, привилегии на которые отзываются.

RULE

Отзывает привилегии на создание правил для таблицы или представления.

REFERENCES

Отзывает привилегии на создание и удаление внешних ключей, ссылающихся на указанный объект.

TRIGGERS

Отзывает привилегии на создание и удаление триггеров на таблице.

CREATE

Отзывает привилегии на создание объектов.

USAGE

Отзывает привилегии на использование доменов[1], пользовательских типов и кодировок.

FROM пользователь[, ...] | PUBLIC | GROUP группа

Указывает пользователя или роль, у которых отзываются привилегии. Ключевое слово *PUBLIC* позволяет отозвать привилегию у роли *PUBLIC* (роли, неявно включающей всех пользователей). Несколько пользователей или групп могут быть перечислены через запятую.

CASCADE | RESTRICT

RESTRICT ограничивает операцию указанной привилегией. *CASCADE* автоматически отзывает все зависимые привилегии. Используется только вместе с *GRANT OPTION FOR*. По умолчанию используется *RESTRICT*.

В описании оператора GRANT приводится полный список привилегий, применимых к каждому типу объекта. Вы можете, соответственно, отозвать у пользователя или роли любую имеющуюся у них привилегию.

Реализация оператора *REVOKE* в PostgreSQL достаточно проста. Единственное, на что стоит обратит внимание, это то, что группы (*GROUP*) являются тем же самым, что и роли (*ROLE*). В следующем примере отзываются привилегии у групп *PUBLIC* и *READ-ONLY*:

```
REVOKE ALL PRIVILEGES ON employee FROM public;
REVOKE SELECT ON jobs FROM read-only;
```

PostgreSQL *не поддерживает* выдачу привилегий на отдельные столбцы таблицы или представления.

При отзыве опции *GRANT OPTION FOR* следует особое внимание обратить на зависимости. Если вы отзовете у одного пользователя эту опцию с параметром *RESTRICT*, то оператор завершится с ошибкой при наличии других пользователей, зависящих от первого. Если же вы отзовете у пользователя опцию *GRANT OPTION FOR* с параметром *CASCADE*, то у других пользователей, зависящих от первого, будут отозваны сами привилегии, выданные первым пользователем.

SQL Server

В SQL Server оператор *REVOKE* используется для отмены любых настроек привилегий пользователей. Этот момент очень важен, потому что в SQL Server поддерживается оператор *DENY*, предназначенный для явного запрещения пользователю доступа к определенным ресурсам. *REVOKE* можно использовать для отзыва привилегий, выданных оператором *GRANT*. Для явного запрещения пользователю определенных действий используйте *DENY*.

SQL Server не поддерживает описанные в ANSI опции *HIERARCHY OPTION* и *ADMIN OPTION*. Хотя *ADMIN OPTION* и отсутствует, поддерживаются несколько административных привилегий (*CREATE* и *BACKUP*). Оператор имеет следующий синтаксис:

---

[1] Для процедурных языков отзывает право использовать данный язык для создания функций. Для схем отзывает право просматривать список объектов в схеме. – *Прим. науч. ред.*

```
REVOKE [GRANT OPTION FOR]
{ [объектная_привилегия][, ...]|
  [системная_привилегия] }
[ON [класс::][объект] [(столбец[, ...])]]|
{TO | FROM} {пользователь[, ...] | роль[, ...] |
  PUBLIC | GUEST}
[CASCADE]
[AS {группа | роль}]
```

где:

GRANT OPTION FOR

Отзывает у пользователя привилегию на передачу собственных привилегий другим пользователям.

объектная_привилегия

Отзывает привилегии на выполнение различных операторов, привилегии можно комбинировать произвольным образом. Полный список объектных привилегий приводится в описании оператора *GRANT*.

системная_привилегия

Отзывает привилегии на выполнение определенных команд и операций. Полный список системных привилегий приводится в описании оператора *GRANT*.

ON объект [(столбец[, ...])]

Указывает объект, привилегии на который отзываются. Если объект является таблицей или представлением, то вы можете при желании отозвать привилегии на отдельные столбцы. На таблицы и представления вы можете отзывать привилегии *SELECT*, *INSERT*, *UPDATE*, *DELETE* и *REFERENCES*, но на отдельные столбцы вы можете отзывать только привилегии *SELECT* и *UPDATE*. На хранимые процедуры, пользовательские функции и расширенные хранимые процедуры можно отзывать привилегию *EXECUTE*.

{TO|FROM} {пользователь |роль|PUBLIC|GUEST}

Указывает пользователя или роль, у которых отзываются привилегии. Можно перечислить несколько пользователей или ролей через запятую. Ключевое слово *PUBLIC* можно использовать для отзыва привилегий у роли *PUBLIC* (роли, неявно включающей всех пользователей). SQL Server также поддерживает роль *GUEST*, являющейся учетной записью для всех пользователей, не имеющих своих учетных записей.

CASCADE

Отзывает привилегии у пользователей, получивших свои привилегии вследствие фразы *WITH GRANT OPTION*. Необходимо использовать вместе с фразой *GRANT OPTION FOR*.

AS {группа |роль}

Определяет полномочия, используемые при отзыве привилегий. В некоторых случаях пользователю может потребоваться временно использовать полномочия группы для отзыва привилегий. Для этого можно использовать фразу *AS*.

Две формы оператора *REVOKE* – для объектных и для системных привилегий – являются взаимоисключающими. Не пытайтесь выполнить оба типа действий

одним оператором. Ключевым отличием в синтаксисе двух этих форм является то, что при отзыве системных привилегий не используется *ON*. Вот пример отзыва системных привилегий:

```
REVOKE CREATE DATABASE, BACKUP DATABASE FROM dylan, katie
```

Если привилегии выдавались с использованием *WITH GRANT OPTION*, то отзывать эти привилегии также нужно с опциями *WITH GRANT OPTION* и *CASCADE*. Например:

```
REVOKE GRANT OPTION FOR
SELECT, INSERT, UPDATE, DELETE ON titles
TO editors
CASCADE
GO
```

REVOKE можно использовать только в текущей базе данных. Поэтому описанные в ANSI опции *CURRENT_USER* и *CURRENT_ROLE* всегда подразумеваются по умолчанию. *REVOKE* также можно использовать для отмены действий оператора *DENY*.

В SQL Server поддерживается оператор *DENY*, имеющий такой же синтаксис, как и *REVOKE*. Однако концептуально эти операторы различаются: *REVOKE* отменяет привилегии, а *DENY* явно запрещает пользователю определенные привилегии. Вы можете использовать оператор *DENY* для запрещения пользователю или роли определенных действий, даже если привилегии на эти действия были выданы явно оператором *GRANT* или через членство в роли.

Вы должны использовать оператор *REVOKE* для отмены выданных или запрещенных привилегий. Предположим, пользователь *kelly* ушла в отпуск по беременности. На время отпуска ей запрещается доступ к таблице **employee**, а по возвращению из отпуска запрет снимается:

```
DENY ALL ON employee TO kelly
GO
REVOKE ALL ON employee TO kelly
GO
```

В этом примере *REVOKE* не отбирает у пользователя привилегии, а просто отменяет действие предыдущего оператора *DENY*.

См. также

GRANT

ROLLBACK

Оператор *ROLLBACK* отменяет действия, выполненные с начала транзакции или с момента определенной точки сохранения. *ROLLBACK* также закрывает открытые курсоры.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с ограничениями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
ROLLBACK [WORK]
[AND [NO] CHAIN]
[TO SAVEPOINT точка_сохранения]
```

Ключевые слова

WORK

Необязательно ключевое слово, не несет смысловой нагрузки.

AND [NO] CHAIN

AND CHAIN завершает текущую транзакцию, но среда выполнения транзакции (например, настройка уровня изоляции транзакции) сохраняется для следующей транзакции. *AND NO CHAIN* просто завершает транзакцию (этот вариант используется по умолчанию).

TO SAVEPOINT точка_сохранения

Позволяет откатить не всю транзакцию, а только до указанной точки сохранения (выполнить частичный откат). *Точка_сохранения* может быть литералом или переменной. Если точки сохранения с указанным именем не существует, то оператор завершается с ошибкой. Если фраза *TO SAVEPOINT* опущена, то все курсоры закрываются. Если фраза *TO SAVEPOINT* используется, то закрываются все курсоры, открытые после точки сохранения.

Помимо отката операторов DML (таких как *INSERT*, *UPDATE* и *DELETE*) *ROLLBACK* откатывает транзакцию до последнего выполненного оператора *START TRANSACTION*, *SET TRANSACTION* или *SAVEPOINT*.

Общие правила

ROLLBACK используется для отмены действий транзакции. Этот оператор можно использовать как для транзакций, начатых явно с помощью *START TRAN* или другого оператора, так и для неявно начинаемых транзакций. *ROLLBACK* взаимно исключает *COMMIT*.

Многие пользователи считают операторы *INSERT*, *UPDATE* и *DELETE* «транзакциями». Однако транзакции могут состоять из большого числа различных операторов. Список этих операторов зависит от платформы, но обычно включает все команды, меняющие данные или объекты базы данных и журналируемые средствами СУБД. В соответствии со стандартом ANSI любой оператор SQL может быть откачен с помощью *ROLLBACK*.

Советы и хитрости

Важно помнить, что в определенных ситуациях некоторые СУБД используют автоматические неявные транзакции, в то время как другие требуют явных транзакций. Поэтому при переносе кода с одной платформы на другую следует использовать стандартный, предопределенный способ работы с транзакциями. Мы рекомендуем всегда явно использовать операторы *START TRAN* или *SET TRAN* для начала транзакций и *COMMIT* или *ROLLBACK* для их завершения.

MySQL

MySQL поддерживает простой и понятный механизм транзакций, а также ключевое слово *CHAIN*:

```
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE] TO [SAVEPOINT] точка_сохранения
```

Опциональное ключевое слово *RELEASE* позволяет автоматически закрыть пользовательское подключение при завершении текущей транзакции.

При создании таблиц в MySQL имейте в виду, что для того, чтобы использовать *ROLLBACK* с этой таблицей, она должна быть транзакционно-защищенной. (Транзакционно-защищенными являются таблицы InnoDB и NDB Cluster. За дополнительной информацией обращайтесь к описанию оператора *CREATE TABLE*.) MySQL позволяет выполнять операторы *COMMIT* и *ROLLBACK* при использовании транзакционно-незащищенных таблиц, но в таком случае эти операторы просто игнорируются и выполняются автоматические фиксации после каждого оператора. Например, если вы вставите данные в транзакционно-незащищенную таблицу, а затем выполните *ROLLBACK*, то вставленные данные не удалятся.

MySQL по умолчанию работает в режиме *AUTOCOMMIT*, поэтому любые изменения данных автоматически записываются на диск. Вы можете выключить режим *AUTOCOMMIT*, выполнив команду *SET AUTOCOMMIT=0*. Вы также можете также контролировать режим автоматической фиксации транзакций на уровне отдельных операторов, используя *BEGIN* или *BEGIN WORK*:

```
BEGIN;
SELECT @A:=SUM(salary) FROM employee WHERE job_type=1;
BEGIN WORK;
UPDATE jobs SET summmary=@A WHERE job_type=1;
COMMIT;
```

MySQL автоматически выполняет *COMMIT* при завершении любого из следующих операторов: *ALTER TABLE*, *BEGIN*, *CREATE INDEX*, *DROP DATABASE*, *DROP TABLE*, *RENAME TABLE* и *TRUNCATE*.

MySQL поддерживает откат до точек сохранения, начиная с версии 4.0.14.

Oracle

Oracle поддерживает оператор *ROLLBACK* по стандарту ANSI с небольшим расширением в виде фразы *FORCE*:

```
ROLLBACK [WORK] {[TO [SAVEPOINT] точка_сохранения] |
  [FORCE 'идентификатор_транзакции']};
```

ROLLBACK отменяет все модификации данных, выполненные в текущей транзакции (или с момента указанной точки сохранения). При этом также освобождаются все удерживаемые блокировки, удаляются все точки сохранения, отменяются все изменения и транзакция завершается.

ROLLBACK...TO SAVEPOINT откатывает только часть транзакции после указанной точки сохранения, удаляет все последующие точки сохранения, освобождает все блокировки, захваченные после указанной точки сохранения. Обратитесь к разделу с описанием оператора *SAVEPOINT* далее в этой же главе за дополнительной информацией.

Реализация оператора в Oracle соответствует стандарту ANSI, за исключением опции *FORCE*. *ROLLBACK FORCE* используется для отката сомнительных распределенных транзакций. Для выполнения *ROLLBACK FORCE* требуется привилегия *FORCE TRANSACTION*. *FORCE* нельзя использовать совместно с *TO SAVEPOINT*. *ROLLBACK FORCE* откатывает не текущую транзакцию, а транзакцию, локальный или глобальный идентификатор которой указан. (Эти транзакции перечислены в системном представлении **DBA_2PC_PENDING**.)

Например, вы могли бы откатить текущую транзакцию до точки сохранения **salary_adjustment**. Следующие две команды эквивалентны:

```
ROLLBACK WORK TO SAVEPOINT salary_adjustment;
ROLLBACK TO salary_adjustment;
```

В следующем примере откатывается сомнительная распределенная транзакция:

```
ROLLBACK FORCE '45.52.67'
```

PostgreSQL

PostgreSQL поддерживает базовый вариант оператора *ROLLBACK* и точки сохранения:

```
ROLLBACK { [WORK] | [TRANSACTION] | PREPARED }
[TO [SAVEPOINT] точка_сохранения ]
```

где:

WORK | TRANSACTION

Необязательные слова.

PREPARED

Откатывает транзакцию, подготовленную для двухфазной фиксации. Только суперпользователь или владелец транзакции может откатить ее. Для подготовки транзакции к двухфазной фиксации используйте не входящий в SQL3 оператор *PREPARE TRANSACTION*, а для фиксации подготовленной транзакции используйте *COMMIT PREPARED*.

TO [SAVEPOINT] точка_сохранения

Откатывает все команды, выполненные после указанной точки сохранения. Точка сохранения остается активной и может быть использована повторно.

ROLLBACK отменяет все изменения данных, выполненные в текущей транзакции. Если на текущий момент транзакция не начата, то оператор вернет ошибку. Например, для отката всех сделанных изменений выполните:

```
ROLLBACK;
```

Будьте внимательны с курсорами при откате к точкам сохранения. Например, курсоры, открытые после точки сохранения, до которой производится откат, закрываются. Если в процессе извлечения строк курсора оператором *FETCH* была создана точка сохранения, то при откате к этой точке позиция курсора остается без изменений. Курсор остается в закрытом состоянии, даже если вы откатываетесь до момента, предшествующего закрытию курсора. Как правило, смешивание курсоров и контрольных точек – не самая лучшая идея.

Помните, что только *RELEASE SAVEPOINT* полностью удаляет точку сохранения. В противном случае точка сохранения остается активной и готовой к использованию.

PostgreSQL поддерживает оператор *ABORT*, являющийся синонимом *ROLLBACK*. Можно использовать *ABORT [WORK]* или *ABORT [TRANSACTION]*.

SQL Server

SQL Server поддерживает ключевые слова *WORK* и *TRAN*. Отличие между ними состоит в том, что *ROLLBACK WORK* не позволяет выполнять откат указанной транзакции или до указанной точки сохранения:

```
ROLLBACK { [WORK] | [TRANSACTION] }
  {имя_транзакции | точка_сохранения};
```

Если *ROLLBACK* выполняется просто с ключевыми словами *WORK* или *TRAN*, то откатываются все текущие открытые транзакции. *ROLLBACK* освобождает все блокировки, хотя при откате до точки сохранения блокировки не освобождаются.

SQL Server позволяет указать имя транзакции или точку сохранения. Они могут быть указаны либо в виде литералов, либо через переменные.

SQL Server не позволяет откатываться к точкам сохранения при двухфазных фиксациях (т. е. при распределенных транзакциях).

Если оператор *ROLLBACK TRANSACTION* выполняется из триггера, то он отменяет все изменения данных, включая выполненные триггером до момента выполнения отката. Вложенные триггеры, следующие за *ROLLBACK*, не выполняются, однако последующие операторы триггера оператором *ROLLBACK* не затрагиваются. *ROLLBACK* действует аналогично *COMMIT* в плане вложенности, сбрасывая в ноль переменную *@@TRANSCOUNT*. (Обратитесь к описанию оператора *COMMIT* за информацией о работе с транзакциями в вложенных триггерах в SQL Server.)

Вот пример использования операторов *COMMIT* и *ROLLBACK* в Transact-SQL. В этом примере в таблицу вставляется несколько записей. Если при вставке происходит ошибка, то транзакция откатывается, а при успешной вставке транзакция фиксируется:

```
BEGIN TRAN - Инициализируем транзакцию
-- Сама транзакция
INSERT INTO sales
VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832')
-- Обработка ошибок
IF @@ERROR <> 0
```

```
BEGIN
  -- Делаем запись об ошибке в журнал событий
  -- и переходим к концу
  RAISERROR 50000 'Insert of sales record failed'
  ROLLBACK WORK
  GOTO end_of_batch
END
-- Фиксируем транзакцию, если не возникло ошибок
COMMIT TRAN
-- Метка для использования оператором GOTO
end_of_batch:
GO
SAVEPOINT sales1
```

См. также

COMMIT
RELEASE SAVEPOINT
SAVEPOINT

SAVEPOINT

Оператор *SAVEPOINT* используется для разделения транзакции на логические части с помощью точек сохранения. В одной транзакции может быть создано несколько точек сохранения. С помощью оператора *ROLLBACK* можно частично откатывать транзакцию до определенной точки сохранения.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

```
SAVEPOINT имя_точки_сохранения
```

Ключевые слова

SAVEPOINT имя_точки_сохранения

Создает в транзакции точку сохранения с указанным именем.

В некоторых платформах допускается создание в одной транзакции нескольких точек сохранения с одинаковыми именами, но это не рекомендуется стандартом ANSI. SQL2003 поддерживает оператор *RELEASE SAVEPOINT* для удаления созданных контрольных точек, описанный в соответствующем разделе.

Общие правила

Точки сохранения создаются и существуют в рамках транзакции, и их имена не должны повторяться в пределах транзакции. Всегда давайте точкам сохранения

понятные имена, потому что вы будете использовать их позднее в ваших программах. Также внимательно используйте операторы *BEGIN* и *COMMIT*: если вы по ошибке напишите *BEGIN* слишком рано или *COMMIT* слишком поздно, то порядок записи транзакций на диск может оказаться неверным.

Советы и хитрости

Как правило, повторяющиеся имена точек сохранения не приводят к ошибкам, но новая точка сохранения делает бесполезной предыдущую одноименную точку сохранения. Поэтому будьте аккуратны при именовании точек сохранения.

При инициализации транзакций для обеспечения целостности используются различные ресурсы (а конкретно – блокировки). Старайтесь делать так, чтобы ваши транзакции завершались как можно быстрее, чтобы удерживаемые блокировки освобождались и могли быть использованы другими пользователями.

В следующем примере выполняется несколько модификаций данных, а затем выполняется откат к точке сохранения:

```
INSERT INTO sales
VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832');
SAVEPOINT after_insert;
UPDATE sales SET terms = 'Net 90'
WHERE sales_id = '7896';
SAVEPOINT after_update;
DELETE sales;
ROLLBACK TO after_insert;
```

MySQL

MySQL полностью поддерживает стандарт ANSI.

Oracle

Oracle полностью поддерживает стандарт ANSI.

PostgreSQL

PostgreSQL полностью поддерживает стандарт ANSI.

SQL Server

SQL Server не поддерживает оператор *SAVEPOINT*. Вместо него используется оператор *SAVE*

```
SAVE TRAN[SACTION] имя_точки_сохранения;
```

Вместо указания имени точки сохранения литералом вы можете использовать переменные. При этом переменные должны иметь тип *CHAR*, *VARCHAR*, *NCHAR* или *NVARCHAR*.

SQL Server позволяет в рамках одной транзакции создать несколько точек сохранения с разными именами. Может показаться, что SQL Server полностью поддерживает вложенные точки сохранения, однако это не так. Каждый раз при выполнении фиксации или отката до точки сохранения выполняется фиксация или откат до последней созданной точки сохранения.

При выполнении *ROLLBACK TRAN* SQL Server откатывает транзакцию до указанной точки сохранения, а затем продолжает выполнение со следующего оператора. Транзакция в конечном итоге должна завершаться оператором *COMMIT* или *ROLLBACK*.

См. также

COMMIT
RELEASE SAVEPOINT
ROLLBACK

SELECT

Оператор *SELECT* извлекает строки, столбцы и производные значения из одной или нескольких таблиц базы данных.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

Полный синтаксис оператора *SELECT* очень сложный и мощный, но его можно разбить на следующие основные фразы:

```
SELECT [{ALL | DISTINCT}] извлекаемый_элемент
  [AS псевдоним][, ...]
FROM [ONLY | OUTER]
{таблица [[AS] псевдоним] | представление
  [[AS] псевдоним]}[, ...]
[ [тип_соединения] JOIN условие_соединения ]
[WHERE условие_поиска] [ {AND | OR | NOT} условие_поиска [...] ]
[GROUP BY выражение_группировки { столбцы_группировки |
  ROLLUP столбцы_группировки  |
  CUBE столбцы_группировки |
  GROUPING SETS   (список_наборов_группировок) |
  ( ) | набор_группировок, список_наборов_группировок }
[HAVING условие_поиска]]
[ORDER BY {выражение_сортировки [ASC | DESC]}[, ...]]
```

Ключевые слова

Каждое из ключевых слов, рассмотренное ниже (за исключением *SELECT* *извлекаемый_элемент*), более детально описывается в разделе «Общие правила»:

[{ALL|DISTINCT}] извлекаемый_элемент

Извлекает значения, составляющие результирующее множество. Каждый извлекаемый элемент может быть литералом, агрегатной или скалярной функцией, математическим выражением, параметром или переменной, под-

запросом, но чаще всего просто столбцом таблицы или представления. Отдельные элементы должны быть разделены запятыми. Если столбцы принадлежат не текущему пользователю, то названия этих столбцов[1] должны быть уточнены названием схемы или владельца. Если таблицей владеет другой пользователь, то имя этого пользователя должно быть включено в ссылку на столбец. Например, если пользователю *jake* требуются данные из схемы **katie**, то запрос может выглядеть так:

```
SELECT emp_id
FROM katie.employee;
```

Вы можете использовать звездочку (*), чтобы извлечь все столбцы из всех таблиц и представлений, перечисленных во *FROM*. Рекомендуется использовать (*) только в запросах, обращающихся к одной таблице.

ALL (используется по умолчанию) возвращает все записи, удовлетворяющие условию поиска. *DISTINCT* удаляет повторяющиеся записи, оставляя по одной записи из каждой группы дубликатов.

AS псевдоним

Заменяет имя столбца (в *SELECT*) или имя таблицы (во *FROM*) на более короткое имя. Эта фраза очень полезна для замены длинных и сложных имен на короткие и легко запоминающиеся. Особенно это важно для столбцов, содержащих вычисляемые выражения, и избавляет вас от необходимости использовать столбцы с названиями вроде **ORA000189x7/0.02**. Также это часто необходимо в запросах, использующих одну и туже таблицу несколько раз, либо содержащих коррелированные подзапросы. Если в *SELECT* или *FROM* указывает несколько элементов, то запятые должны быть указаны после *AS псевдоним*. Также следует использовать единожды объявленные псевдонимы везде в запросе.

FROM имя_таблицы

Указывает все таблицы и представления, из которых запрос извлекает данные. Имена таблиц и представлений должны быть разделены запятыми. Во *FROM* можно указать псевдонимы с помощью *AS*. Использование коротких псевдонимов вместо длинных имен таблиц и представлений упрощает кодирование. (Хотя псевдонимы могут противоречить тщательно разработанным администраторами правилам именования, это не проблема, так как псевдонимы существуют только в рамках запроса, в которых они объявлены. Обратитесь к разделу «Общие правила» за дополнительной информацией по псевдонимам.) Фраза *FROM* может содержать подзапросы (подзапросы описываются в отдельном разделе в этой главе).

ONLY

Указывает, что запрос возвращает строки только из указанных таблиц и представлений, но не из дочерних объектов. При использовании *ONLY* заключите имена таблиц и представлений в скобки. *ONLY* игнорируется, если таблица или представление не имеет наследников.

---

[1] Вероятно, авторы в данном случае имеют в виду таблицы. Это можно видеть из примера, приводимого ниже. – *Прим. науч. ред.*

OUTER

Указывает, что запрос возвращает как данные из указанных таблиц и представлений, так и данные из дочерних таблиц и представлений. Столбцы дочерних таблиц и представлений добавляются в результирующий набор справа в порядке иерархии объектов. В расширенных иерархиях подтаблицы с общими родителями добавляются в порядке создания их типов. При использовании *OUTER* заключите имена таблиц и представлений в скобки. *OUTER* игнорируется, если таблица или представление не имеет наследников.

JOIN условие_соединения

Соединяет строки таблицы, указанной во *FROM*, со строками другой таблицы на основании связи между двумя таблицами по наборам общих значений. Эти значения обычно хранятся в столбцах с одинаковыми именами и типами данных. Такие столбцы называются *ключом соединения*. Часто, но не всегда ключ соединения является первичным ключом одной таблицы и внешним ключом другой таблицы. Если данные в двух таблицах соответствуют, то таблицы можно соединить. (Обратите внимание, что соединение может быть выполнено при помощи фразы *WHERE*. Этот способ называется тэта-соединением.)

условие_соединения чаще всего имеет следующий вид:

```
JOIN таблица2 ON таблица1.столбец1 <оператор_сравнения>
  таблица2.столбец1
JOIN таблица3 ON таблица1.столбецA <оператор_сравнения>
  таблица3.столбецA
[...]
```

Если в качестве оператора сравнения используется =, то соединение называется *эквисоединением*. Однако можно использовать любой оператор: <, >, <=, >= и даже <>.

Используйте оператор *AND*, чтобы выполнить соединение по нескольким условиям. Также вы можете указывать альтернативные условия соединения при помощи *OR*.

Если тип соединения не указан явно, то используется внутреннее соединение. Помните, что существует много разных типов соединений, каждый со своими правилами и особенностями (они рассматриваются в разделе «Общие правила»). Также имейте в виду, что есть альтернативный вариант описания условий соединений с помощью фразы *USING*:

USING (столбец[, ...])

Является альтернативой соединения с помощью *ON*. Вместо описания условий соединения вы можете перечислить столбцы, существующие в обеих таблицах. Соединение выполняется по равенству значений в этих столбцах. Следующие два запроса имеют одинаковый результат:

```
SELECT emp_id
FROM employee
LEFT JOIN sales USING (emp_id, region_id);

SELECT emp_id
FROM employee AS e
```

```
LEFT JOIN sales AS s
  ON e.emp_id = s.emp_id
  AND e.region_id = s.region_id;
```

WHERE условие_поиска

Отфильтровывает ненужные данные, оставляя только строки, удовлетворяющие указанному условию. Неправильно написанная фраза *WHERE* может похоронить производительность нормального в остальном оператора *SELECT*, поэтому изучение всех нюансов *WHERE* является задачей первостепенной важности. Условие поиска имеет следующий вид:

```
WHERE [схема.[таблица.]]столбец оператор значение
```

Во фразах *WHERE* обычно оцениваются значения в столбцах таблицы. Значения оцениваются при помощи различных операторов (рассматриваются в главе 2). Например, вы можете проверять столбец на равенство (=) определенному значению, на превышение (>) определенного значения либо на попадание в определенный диапазон (*BETWEEN*).

Фраза *WHERE* может содержать несколько условий, соединенных операторами *AND* и *OR*. Для определения порядка вычисления условий вы можете использовать скобки. Фраза *WHERE* также может содержать подзапросы (обратитесь к описанию *WHERE* в отдельном разделе в этой главе).

GROUP BY выражение_группировки

Используется в запросах, содержащих такие функции, как *AVG*, *COUNT*, *COUNT DISTINCT*, *MAX*, *MIN* и *SUM* для группировки всего результирующего множества на категории на основании заданного выражения. Выражение группировки используется следующим образом:

```
[GROUP BY выражение_группировки
```

и имеет следующий синтаксис:

```
{ (столбец_группировки [, ...]) |
  ROLLUP (столбец_группировки[, ...]) |
  CUBE (столбец_группировки [, ...]) |
  GROUPING SETS (список_наборов_группировок) |
  ( ) | набор_группировок, список_наборов_группировок }
```

Дополнительная информация и примеры использования *CUBE*, *ROLLUP* и *GROUPING SETS* приводятся в разделе «Общие правила».

HAVING условие_поиска

Накладывает дополнительное условие на группы, являющиеся результатом работы *GROUP BY*. *HAVING* не влияет на данные, используемые для расчета агрегатов. *HAVING* может содержать подзапросы.

ORDER BY выражение_сортировки [ASC|DESC]

Упорядочивает результирующее множество по возрастанию (*ASC*) или убыванию (*DESC*), используя значения указанного выражения. Выражение состоит из разделенного запятыми списка столбцов, по которым вы хотите выполнить сортировку.

Общие правила

Каждая фраза оператора *SELECT* имеет определенное назначение. Поэтому можно отдельно обсуждать фразу *FROM*, отдельно *WHERE*, отдельно *GROUP BY* и т. д. Вы можете получить больше информации и увидеть больше примеров операторов *SELECT*, просмотрев отдельные описания каждой фразы оператора. Однако не в каждом запросе необходимы все фразы. Как минимум в запросе должны быть использованы *SELECT* и *FROM*. Так как оператор *SELECT* очень важен и имеет много опций, то мы решили разбить раздел «Общие правила» на следующие подразделы:

- Псевдонимы и соединения в *WHERE*
- Фраза *JOIN*
- Фраза *WHERE*
- Фраза *GROUP BY*
- Фраза *HAVING*
- Фраза *ORDER BY*

Псевдонимы и соединения в WHERE

Иногда имена столбцов необходимо уточнять именами базы данных, схемы и таблицы, особенно если столбцы с одинаковыми именами есть в нескольких используемых таблицах. Например, в Oracle в схеме **scott** столбец **job_id** есть как в таблице **jobs**, так и в **employee**. В следующем запросе две эти таблицы соединяются при помощи фразы *WHERE*:

```
SELECT   scott.employee.emp_id,
         scott.employee.fname,
         scott.employee.lname,
         jobs.job_desc
FROM     scott.employee,
         jobs
WHERE    scott.employee.job_id = jobs.job_id
ORDER BY scott.employee.fname,
         scott.employee.lname
```

Вы можете использовать псевдонимы, чтоб сделать запрос более простым и понятным:

```
SELECT   e.emp_id,
         e.fname,
         e.lname,
         j.job_desc
FROM     scott.employee AS e,
         jobs AS j
WHERE    e.job_id = j.job_id
ORDER BY e.fname,
         e.lname
```

Эти два запроса иллюстрируют также следующие важные правила, касающиеся *WHERE*:

1. Разделяйте отдельные элементы в *SELECT* и таблицы во *FROM* запятыми.
2. Используйте *AS* для объявления псевдонимов.
3. Объявленные псевдонимы используйте во всех частях запроса.

Как правило, следует использовать для соединений фразу *JOIN*, а не описывать условие соединения в *WHERE*. Это делает ваш код более простым и понятным, позволяя легко отличить условие соединения от условия поиска. Кроме того, это избавляет от неочевидных конструкций, используемых на некоторых платформах для записи с помощью *WHERE* внешних соединений.

Фраза JOIN

Для написания соединения из предыдущего примера в стиле ANSI нужно указать после имени первой таблицы ключевое слово *JOIN*, затем имя второй таблицы, затем ключевое слово *ON* и, наконец, то же условие соединения, что использовалось в *WHERE*. Вот как выглядит тот же запрос с соединением в стиле ANSI:

```
SELECT   e.emp_id, e.fname, e.lname, j.job_desc
FROM     scott.employee AS e
JOIN     jobs AS j ON e.job_id = j.job_id
ORDER BY e.fname, e.lname;
```

В качестве альтернативного варианта вы могли бы использовать фразу *USING*. Вместо описания условий соединения вы можете перечислить столбцы, существующие в обеих таблицах. Соединение выполняется по равенству значений в этих столбцах. В следующем примере два запроса (один с *ON*, другой с *USING*) получают одинаковый результат:

```
SELECT emp_id
FROM employee
LEFT JOIN sales USING (emp_id, region_id);
```

```
SELECT emp_id
FROM employee AS e
LEFT JOIN sales AS s
  ON e.emp_id = s.emp_id
  AND e.region_id = s.region_id;
```

Вы можете использовать следующие типы ANSI соединений:

CROSS JOIN

Возвращает кросс-соединение двух таблиц. К каждой записи из первой таблицы присоединяются все записи из второй таблицы, что может привести к огромному результирующему множеству. (Конечно, результат будет небольшим, если в каждой таблице по 4 строки, но представьте себе что будет, если в каждой таблице будет по 4 миллиона строк!) Это соединение выполняется также при отсутствии условия соединения и известно как картезианское (или декартово) произведение. Мы не рекомендуем использовать кросс-соединения.

INNER JOIN

Отбрасывает неприсоединенные строки из каждой таблицы. Этот тип соединения используется по умолчанию.

LEFT [OUTER] JOIN

Указывает, что возвращаются все записи из левой таблицы. Если для записи из левой таблицы нет соответствующей записи в правой таблице, то запись возвращается, но в столбцах правой таблицы будут значения NULL. Профессионалы рекомендуют все внешние соединения записывать как левые, а не смешивать использование левых и правых соединений.

RIGHT [OUTER] JOIN

Возвращает все записи из правой таблицы вне зависимости от наличия соответствующих записей в левой таблице. В этом случае столбцы, соответствующие левой таблице, могут иметь пустые значения.

FULL [OUTER] JOIN

Возвращает все данные из обеих таблиц вне зависимости от того, была ли определенная запись соединена с записью из противоположной таблицы. Если для какой-либо строки нет соответствия в соединяемой таблице, то в результирующем наборе будут значения NULL.

Не все типы соединений поддерживаются всеми платформами. Поэтому за детальной информацией о поддержке соединений определенной платформой обращайтесь к секции, описывающей конкретную СУБД.

Фраза WHERE

Плохо написанная фраза *WHERE* может похоронить прекрасный в остальном оператор *SELECT*, поэтому стоит внимательно изучить все нюансы фразы *WHERE*. Вот пример типичного подзапроса с составным условием в *WHERE*:

```
SELECT a.au_lname,
       a.au_fname,
       t2.title,
       t2.pubdate
FROM  authors a
JOIN  titleauthor t1 ON a.au_id = t1.au_id
JOIN  titles t2 ON t1.title_id = t2.title_id
WHERE (t2.type = 'business' OR t2.type = 'popular_comp')
  AND t2.advance > 5500
ORDER BY t2.title
```

Обратите внимание, что в этом запросе скобки влияют на порядок вычисления условия поиска. Вы можете использовать скобки для изменения приоритета условий поиска так же, как это делается в алгебраических выражениях.

На некоторых платформах используемая по умолчанию схема упорядочения (то есть порядок сортировки) может влиять на то, как WHERE фильтрует результат запроса. Например, SQL Server сортирует данные в алфавитном порядке без учета регистра, поэтому значения «smith», «SMITH» и «Smith» считаются одинаковыми. А вот Oracle учитывает регистр данных, поэтому значения «smith», «SMITH» и «Smith» считаются различными.

Фраза *WHERE* предлагает намного больше возможностей, чем показано в предыдущем примере. В следующем списке приведены основные возможности фразы *WHERE*:

NOT

Инвертирует значение условия *WHERE*. В запросе вы можете написать *WHERE NOT LIKE*... или *WHERE NOT IN*... .

Операторы сравнения

Вы можете сравнивать наборы значений, используя операторы <, >, <>, >=, <=, и =. Например:

```
WHERE emp_id = '54123'
```

Условия IS NULL и IS NOT NULL

Позволяют искать пустые или непустые значения, используя формат *WHERE выражение IS [NOT] NULL*.

AND

Позволяет объединять несколько условий и отбирать только те строки, которые удовлетворяют всем условиям. Например:

```
WHERE job_id = '12' AND job_status = 'active'
```

OR

Позволяет объединять несколько условий и отбирать только те строки, которые удовлетворяют хотя бы одному условию. Например:

```
WHERE job_id = '13' OR job_status = 'active'
```

LIKE

Позволяет реализовывать поиск по шаблонам, указываемым в кавычках. Групповые символы, поддерживаемые в шаблонах, зависят от платформы и описываются в разделах, посвященным этим платформам. Все платформы поддерживают групповой символ %. Например, чтобы найти все телефоны, код города которых начинается на 415, можно использовать следующее условие:

```
WHERE phone LIKE '415%'
```

EXISTS

Используется совместно с подзапросами для определения того, что подзапрос возвращает данные. Обычно это работает намного быстрее, чем подзапросы с условием *WHERE IN*. Например, следующий запрос находит всех авторов, являющихся сотрудниками:

```
SELECT au_lname FROM authors WHERE EXISTS
  (SELECT 'X' FROM employees last_name=au_lname)
```

BETWEEN

Проверяет попадание значения в диапазон, задаваемый двумя другими значениями (границы диапазона включаются). Например:

```
WHERE ytd_sales BETWEEN 4000 AND 9000.
```

IN

Проверяет вхождение значения в определенный список. Список может задаваться как литералом (например, *WHERE state IN ('or', 'il', 'tn', 'ak')*), так и подзапросом:

```
WHERE state IN (SELECT state_abbr FROM territories).
```

SOME|ANY

Работает так же, как оператор *EXISTS*, но использует немного другой синтаксис. Следующий запрос возвращает авторов, являющихся также сотрудниками:

```
SELECT au_lname FROM authors WHERE au_lname =
   SOME(SELECT last_name FROM employees)
```

ALL

Проверяет, что все строки, возвращаемые подзапросом, удовлетворяют определенному условию. Если запрос не возвращает ни одной строки, то результат *ALL* будет TRUE. Например:

```
WHERE city =
  ALL (SELECT city FROM employees WHERE emp_id = 54123)
```

Фраза GROUP BY

Фраза *GROUP BY* (как и *HAVING*) используется в запросах, содержащих агрегатные функции.

GROUP BY позволяет получать агрегированные значения для одной или нескольких строк, получаемых при разбиения всего множества строк на группы по значениям указанных *столбцов группировки*. Например, следующим запросом мы подсчитываем число сотрудников, нанятых за каждый год с 1999 по 2004:

```
SELECT hire_year, COUNT(emp_id) AS nbr_emps
 FROM employee
WHERE status = 'ACTIVE'
  AND hire_year BETWEEN 1999 AND 2004
GROUP BY hire_year;
```

Результат:

```
hire_year nbr_emps
--------- --------
1999      27
2000      17
2001      13
2002      19
2003      20
2004      32
```

Агрегатные функции используются в запросах для получения суммарных значений. Основные агрегатные функции следующие:

AVG

Возвращает среднее по всем непустым (не NULL) значениям указанного столбца.

AVG DISTINCT

Возвращает среднее по всем уникальным непустым значениям указанного столбца.

COUNT

Возвращает число непустых значений в столбце.

COUNT DISTINCT

Возвращает число уникальных непустых значений в столбце.

COUNT(*)

Считает число строк в таблице.

MAX

Возвращает максимальное непустое значение в столбце.

MIN

Возвращает минимальное непустое значение в столбце.

SUM

Возвращает сумму всех непустых значений в столбце.

SUM DISTINCT

Возвращает сумму всех уникальных непустых значений в столбце.

Некоторые запросы, использующие агрегаты, возвращают одиночное значение. Такие запросы называются скалярными агрегатами. В скалярных агрегатах не требуется фраза *GROUP BY*. Например:

```
-- Запрос
SELECT AVG(price)
FROM titles
-- Результат
14.77
```

Запросы, возвращающие и агрегированные значения, и обычные столбцы, называются векторными агрегатами. Векторные агрегаты используют *GROUP BY* и возвращают одну или несколько строк. Есть несколько правил, касающихся использования *GROUP BY*:

- Фраза *GROUP BY* должна находиться в правильном месте – после *WHERE*, но до *ORDER BY*.
- Все столбцы, которые присутствуют во фразе *SELECT*, но по которым не выполняется агрегация, должны быть указаны в *GROUP BY*.
- Не используйте в *GROUP BY* псевдонимы столбцов (хотя псевдонимы таблиц допустимы).

Предположим, вы хотите получить общую сумму по нескольким покупкам. Таблица **Order_Details** выглядит следующим образом:

```
OrderID   ProductID   UnitPrice  Quantity
--------- ---------- ---------- --------
10248     11         14.0000    12
10248     42         9.8000     10
10248     72         34.8000    5
```

```
10249     14         18.6000    9
10249     51         42.4000    40
10250     41         7.7000     10
10250     51         42.4000    35
10250     65         16.8000    15
...
```

Следующий запрос получает требуемый результат:

```
SELECT OrderID, SUM(UnitPrice * Quantity) AS 'Order Amt'
FROM order_details
WHERE orderid IN (10248, 10249, 10250)
GROUP BY orderid
```

Результат:

```
OrderID  Order Amt
-------- ----------------
10248    440.0000
10249    1863.4000
10250    1813.0000
```

Мы можем детализировать агрегаты, добавив еще один столбец группировки. Следующий запрос возвращает среднюю цену продуктов в разбивке по имени и размеру:

```
SELECT name, size, AVG(unit_price) AS 'avg'
FROM product
GROUP BY name, size
```

Результат:

```
Name                Size   avg
------------------- ------ -----------------------
Flux Capacitor      small  900
P32 Space Modulator small  1400
Transmorgrifier     medium 1400
Acme Rocket         large  600
Land Speeder        large  6500
```

Во фразе *GROUP BY* поддерживаются несколько важных подфраз:

GROUP BY [{ROLLUP | CUBE}] ([столбец_группировки[, ...]])[, список_наборов_группировок]

Группирует значения результирующего набора по одному или нескольким столбцам. (Фраза *GROUP BY* без *CUBE* и *ROLLUP* является простейшим и наиболее часто используемым вариантом.)

ROLLUP

Добавляет в результирующее множество общий итог и промежуточные итоги для каждой комбинации столбцов в виде иерархии. *ROLLUP* добавляет по одной дополнительной строке на каждую группу. В результирующем множестве для столбца, по которому произведена группировка, проставляется NULL как индикатор итогового значения.

CUBE

Формирует итоги для каждой комбинации столбцов группировки. По сути, *CUBE* позволяет быстро получить многомерный набор данных из обычной таблицы без особого программирования. Как и *ROLLUP*, *CUBE* формирует итоги для всех столбцов группировки, но делает это для всех комбинаций столбцов.

GROUP BY GROUPING SETS [{ROLLUP|CUBE}] ([столбец_группировки[, ...]]) [, список_наборов_группировок]

Позволяет в одном запросе формировать агрегаты по нескольким наборам столбцов группировки. Это особенно полезно в случаях, когда вы хотите получить только часть агрегированных результатов. Фраза *GROUPING SETS* позволяет выбрать определенные комбинации столбцов группировок, в то время как *CUBE* использует все комбинации столбцов (получается 2^n групп, где n – количество столбцов в CUBE), а *ROLLUP* – иерархическое подмножество всех комбинаций. Как видно из описания синтаксиса, *GROUPING SETS* можно использовать совместно с *CUBE* и *ROLLUP*.

В табл. 3.4 приводятся примеры фраз *GROUP BY* и соответствующие им наборы столбцов группировок.

Таблица 3.4. Варианты синтаксиса GROUP BY

Синтаксис *GROUP BY*	**Наборы столбцов**
GROUP BY (col_A, col_B, col_C)	(col_A, col_B, col_C)
GROUP BY ROLLUP (col_A, col_B, col_C)	(col_A, col_B, col_C) (col_A, col_B) (col_A) ( )
GROUP BY CUBE (col_A, col_B, col_C)	(col_A, col_B, col_C) (col_A, col_B) (col_A) (col_B, col_C) (col_B) (col_A, col_C) (col_C) ( )
GROUP BY GROUPING SETS ((col_A, col_B), (col_A, col_C), (col_C))	Подзапрос: SELECT * FROM stores WHERE stor_id IN (SELECT stor_id FROM sales WHERE ord_date > '01-JAN-2004')

Каждый вариант фразы *GROUP BY* возвращает различный набор агрегированных значений, а также, в случае *ROLLUP* или *CUBE*, различный набор промежуточных итогов. Суть *ROLLUP*, *CUBE* и *GROUPING SETS* более понятна при рассмотрении на примерах. В следующем запросе мы суммируем число заказов по годам и по кварталам:

```
SELECT order_year AS year, order_quarter AS quarter,
  COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY ROLLUP (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```

Результат:

```
year quarter orders
---- ------- ------
NULL NULL    648 -- общий итог
2003 NULL    380 -- итог за 2003 год
2003 1       87
2003 2       77
2003 3       91
2003 4       125
2004 NULL    268 -- итог за 2004 год
2004 1       139
2004 2       119
2004 3       10
```

Добавление столбцов группировок увеличивает детализацию (и количество подитогов) в результирующем множестве. Изменим предыдущий запрос, добавив группировку по региону. (Так как при этом увеличивается число возвращаемых строк, то мы покажем данные только за первый и второй квартал.)

```
SELECT order_year AS year, order_quarter AS quarter, region,
  COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
  AND order_quarter IN (1,2)
  AND region IN ('USA', 'CANADA')
GROUP BY ROLLUP (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```

Результат:

```
year quarter region orders
---- ------- ------ ------
NULL NULL    NULL   183 -- общий итог
2003 NULL    NULL   68  -- итог за 2003
2003 1       NULL   36  -- итог по всем регионам за 1 кв.2003
2003 1       CANADA 3
2003 1       USA    33
2003 2       NULL   32  -- итог по всем регионам за 2 кв.2003
2003 2       CANADA 3
2003 2       USA    29
2004 NULL    NULL   115 -- итог за 2004
2004 1       NULL   57  -- итог по всем регионам за 1 кв.2004
2004 1       CANADA 11
2004 1       USA    46
2004 2       NULL   58  -- итог по всем регионам за 2 кв.2004
2004 2       CANADA 4
2004 2       USA    54
```

Фраза *GROUP BY CUBE* используется для многомерного анализа агрегированных данных. Как и *GROUP BY ROLLUP*, она создает дополнительные итоги, но итоги создаются для каждой комбинации столбцов группировки. (Как вы увидите, это увеличивает число строк в результирующем наборе.)

В следующем примере мы суммируем количество заказов по годам и по кварталам:

```
SELECT order_year AS year, order_quarter AS quarter,
COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY CUBE (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```

Результат:

```
year quarter orders
---- ------- ------
NULL NULL    648 -- общий итог
NULL 1       226 -- итог за первый квартал по обоим годам
NULL 2       196 -- итог за второй квартал по обоим годам
NULL 3       101 -- итог за третий квартал по обоим годам
NULL 4       125 -- итог за четвертый квартал по обоим годам
2003 NULL    380 -- общий итог за 2003 год
2003 1       87
2003 2       77
2003 3       91
2003 4       125
2004 NULL    268 -- общий итог за 2004 год
2004 1       139
2004 2       119
2004 3       10
```

Фраза *GROUP BY GROUPING SETS* позволяет агрегировать более чем по одному набору. Для каждого набора группировки запрос возвращает итог со значением NULL в соответствующем столбце группировки. В то время как *CUBE* и *ROLLUP* создают предопределенные итоги, фраза *GROUPING SETS* позволяет выбрать, какие итоги вы хотите добавить в запрос. Фраза *GROUPING SETS* не создает общий итог.

Используем пример, аналогичный приведенным в описаниях *CUBE* и *ROLLUP*, но на этот раз посчитаем итоги по годам и кварталам, а затем отдельно по годам:

```
SELECT order_year AS year, order_quarter AS quarter,
  COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY GROUPING SETS ( (order_year, order_quarter),
  (order_year) )
ORDER BY order_year, order_quarter;
```

Результат:

```
year quarter orders
---- ------- ------
```

```
2003 NULL    380 -- общий итог за 2003 год
2003 1       87
2003 2       77
2003 3       91
2003 4       125
2004 NULL    268 -- общий итог за 2004 год
2004 1       139
2004 2       119
2004 3       10
```

Запрос с *GROUPING SETS* можно также представить как объединение с помощью *UNION ALL* нескольких запросов с фразами *GROUP BY*, содержащими отдельные наборы группировок. Вы также можете добавить в *GROUPING SETS* дополнительные итоги с помощью *CUBE* и *ROLLUP* в зависимости от ваших требований.

Наборы группировок в *GROUPING SETS* можно конкатенировать, чтобы генерировать большое число комбинаций группировок. Конкатенированные наборы группировок формируют произведение группировок из каждого набора группировок, указанного в списке *GROUPING SETS*. Конкатенированные наборы группировок совместимы с *CUBE* и *ROLLUP* и создают большое число группировок даже при небольшом числе исходных наборов группировок. Это демонстрируется в табл. 3.5.

Таблица 3.5. Варианты синтаксиса GROUP BY

Синтаксис *GROUP BY*	**Наборы группировок**
GROUP BY (col_A, col_B, col_C)	(col_A, col_B, col_C)
GROUP BY GROUPING SETS (col_A, col_B) (col_Y, col_Z)	(col_A, col_Y) (col_A, col_Z) (col_B, col_Y) (col_B, col_Z)

Вы можете представить, каким большим будет результирующее множество, если в *GROUPING SETS* конкатенируются большие наборы группировок. Однако возвращаемый результат может быть очень ценным, и его трудно получить другим способом.

Фраза HAVING

Фраза *HAVING* накладывает условие отбора на результат работы *GROUP BY*. В остальном *HAVING* очень похожа на *WHERE*, и в ней можно использовать все те же функции и операторы, что и в *WHERE*. Например, мы можем использовать следующий запрос для отбора тех должностей, на которых работают более чем по три человека:

```
-- Запрос
SELECT    j.job_desc "Job Description",
          COUNT(e.job_id) "Nbr in Job"
FROM      employee e
JOIN      jobs j ON e.job_id = j.job_id
GROUP BY  j.job_desc
```

```
HAVING    COUNT(e.job_id) > 3
-- Результат
Job Description            Nbr in Job
-------------------------- -----------
Acquisitions Manager       4
Managing Editor            4
Marketing Manager          4
Operations Manager         4
Productions Manager        4
Public Relations Manager   4
Publisher                  7
```

Заметьте, что по стандарту ANSI фраза *GROUP BY* не является обязательной при использовании *HAVING*. Например, следующий запрос к таблице **employee** корректен, так как *GROUP BY* в нем подразумевается:

```
SELECT COUNT(dept_nbr)
FROM employee
HAVING COUNT(dept_nbr) > 30;
```

Хотя такое использование *HAVING* корректно, оно используется достаточно редко.

Фраза ORDER BY

Результирующее множество может быть отсортировано с помощью фразы *ORDER BY* в соответствии с определенной в базе данных схемой упорядочения. Каждый столбец может использоваться для сортировки по возрастанию (*ASC*) или убыванию (*DESC*). (По умолчанию выполняется сортировка по возрастанию.) Если *ORDER BY* не используется, то большинство платформ возвращают данные либо в порядке их физического хранения, либо в порядке, определяемом используемым в запросе индексом. Однако при отсутствии *ORDER BY* какой-либо определенный порядок строк не гарантируется. Вот пример использования *ORDER BY* в запросе для SQL Server:

```
SELECT   e.emp_id "Emp ID",
         e.fname "First",
         e.lname "Last",
         j.job_desc "Job Desc"
FROM     employee e,
         jobs j
WHERE    e.job_id = j.job_id
  AND    j.job_desc = 'Acquisitions Manager'
ORDER BY e.fname DESC,
         e.lname ASC
```

Результат:

```
Emp ID    First    Last        Job Desc
--------- -------- ----------- --------------------
MIR38834F Margaret Rance       Acquisitions Manager
MAS70474F Margaret Smith       Acquisitions Manager
KJJ92907F Karla    Jablonski   Acquisitions Manager
GHT50241M Gary     Thomas      Acquisitions Manager
```

После того как таблицы соединены и отобраны только запрошенные строки, данные сортируются по имени в убывающем порядке. Если имена каких-либо авторов совпадают, то эти строки упорядочиваются по фамилии в порядке возрастания.

Вы можете использовать в *ORDER BY* столбцы, не извлекаемые запросом. Например, вы можете запрашивать из таблицы идентификаторы сотрудников, но в алфавитном порядке по имени и фамилии сотрудника.

Советы и хитрости

Если вы объявили во фразе *FROM* псевдоним для таблицы или представления, то используйте этот псевдоним при всех ссылках на таблицу или представление в запросе (например, во фразе *WHERE*). Не смешивайте обращение к таблице по имени и с помощью псевдонима. Этому есть две причины. Во-первых, это делает код менее согласованным и усложняет его поддержку. Во-вторых, на некоторых платформах запросы, содержащие смешанные ссылки на таблицы, возвращают ошибки. (Обратитесь к разделу, описывающему подзапросы, за информацией об использовании псевдонимов с подзапросами.)

MySQL, PostgreSQL и SQL Server поддерживают определенный тип запросов, не требующий фразы *FROM*. Используйте такие запросы осторожно, так как по стандарту ANSI фраза *FROM* является обязательной. Запросы без *FROM* при миграции может потребоваться приводить к стандарту ANSI либо к формату, работающему на целевой платформе. Некоторые платформы не поддерживают соединения в стиле ANSI. За информацией о степени поддержки стандарта ANSI для различных частей оператора *SELECT* обращайтесь к документации по платформам.

MySQL

Реализация оператора *SLECT* в MySQL включает частичную поддержку *JOIN*, фразы *INTO*, *LIMIT* и *PROCEDURE*. MySQL до версии 4.0 не поддерживает подзапросы. Синтаксис следующий:

```
SELECT [DISTINCT | DISTINCTROW | ALL]
  [STRAIGHT_JOIN] [ {SQL_SMALL_RESULT | SQL_BIG_RESULT} ]
  [SQL_BUFFER_RESULT]
  [ {SQL_CACHE | SQL_NO_CACHE} ] [SQL_CALC_FOUND_ROWS]
[HIGH_PRIORITY] извлекаемый_элемент AS псевдоним[, ...]
[INTO {OUTFILE | DUMPFILE | переменная[, ...]}
  'файл' опции]
[FROM имя_таблицы AS псевдоним[, ...]
  [ { USE INDEX (индекс1[, ...]) |
  IGNORE INDEX (индекс1[, ...]) } ]
[тип_соединения] [JOIN таблица2] [ON условие_соединения]
[WHERE условие_поиска]
[GROUP BY {целое_число | имя_столбца | формула}
[ASC | DESC][, ...] [WITH ROLLUP]]
[HAVING условие_поиска]
[ORDER BY { целое_число | имя_столбца | формула } [ASC | DESC][, ...]]
```

```
[LIMIT { [смещение,] число_строк] | число_строк
  OFFSET смещение }]
[PROCEDURE процедура (параметр[, ...])]
[{FOR UPDATE | LOCK IN SHARE MODE}];
```

где:

STRAIGHT_JOIN

Указывает оптимизатору, что таблицы нужно соединять в том порядке, в котором они указаны во FROM.

SQL_SMALL_RESULT | SQL_BIG_RESULT

Подсказывает оптимизатору, будет ли результатом работы *GROUP BY* или *DISTINCT* большое (*SQL_BIG_RESULT*) или маленькое (*SQL_SMALL_RESULT*) множество строк. Для обработки *GROUP BY* и *DISTINCT* в MySQL строятся временные таблицы, и эти параметры помогают определить, можно ли хранить временную таблицу в памяти или нужно записать ее на диск.

SQL_BUFFER_RESULT

Предписывает сохранять результирующий набор во временную таблицу, что позволяет раньше освобождать блокировки и быстрее возвращать данные клиенту.

SQL_CACHE | SQL_NO_CACHE

Контролирует кэширование результата запроса. *SQL_CACHE* сохраняет результат запроса в кэше, если запрос является кэшируемым и параметр *query_cache_type* имеет значение *2* или *DEMAND*. *SQL_NO_CACHE* не сохраняет результат запроса в кэш. Для запросов с *UNION*, подзапросами и представлениями везде используется *SQL_NO_CACHE*, если эта опция используется в первом запросе. А *SQL_CACHE* применяется, только если используется после первого запроса.

SQL_CALC_FOUND_ROWS

Подсчитывает число строк в результирующем наборе (вне зависимости от использования *LIMIT*), которое можно получить с помощью *SELECT FOUND_ROWS( )*.

HIGH_PRIORITY

Устанавливает для запроса более высокий приоритет, нежели приоритет операторов, модифицирующих таблицу. Эту опцию нужно использовать только для отдельных запросов, требующих максимальной скорости работы.

извлекаемый элемент

Извлекает указанные столбцы или выражения. Столбцы могут быть указаны в формате *[база_данных.][таблица.]столбец*. Если имя базы данных или таблицы не указано, то подразумеваются текущая база данных и таблица.

FROM...USE INDEX | IGNORE INDEX

Указывает таблицу, из которой извлекаются записи. Таблица может быть указана в формате *[база_данных.][таблица]*. Если во *FROM* указано несколько таблиц, то будет выполнено соединение. Опции *FORCE INDEX*, *USE INDEX (индекс1, ...)* и *IGNORE INDEX (индекс1, ...)* позволяют, соответственно, ис-

пользовать указанные индексы либо не использовать указанные индексы для выполнения запроса.

INTO {OUTFILE | DUMPFILE | переменная[, ...]} 'файл'

Опция *OUTFILE* записывает результат запроса в файл с указанным именем. Файл с указанным именем не должен существовать. Опция *DUMPFILE* записывает в файл данные в одну строку без разделителей столбцов и переносов строк. Эта опция используется в основном для значений *BLOB*. Специфические правила использования этой фразы приводятся далее. Фраза *INTO* позволяет сохранить значения в одну или несколько переменных (по одной для каждого возвращаемого столбца). При сохранении значений в переменные имя файла не указывается.

LIMIT {[смещение,] число_строк | число_строк OFFSET смещение}

Ограничивает число строк, возвращаемое запросом, а также указывает число строк, которое нужно пропустить, перед тем как начать возвращать строки клиенту. Если указано только одно число, то это общее число возвращаемых строк, и строки возвращаются, начиная с самой первой. Альтернативный синтаксис *число_строк OFFSET смещение* поддерживается для совместимости с PostgreSQL.

PROCEDURE процедура (параметр[, ...])

Указывает процедуру, обрабатывающую данные результирующего множества. Обычно это внешняя процедура, написанная на C++, а не внутренняя хранимая процедура.

FOR UPDATE | LOCK IN SHARE MODE

FOR UPDATE накладывает на записи эксклюзивную блокировку, чтобы эти записи можно было модифицировать (работает только для таблиц InnoDB и BDB), а *LOCK IN SHARE* накладывает разделяемую блокировку, что позволяет нескольким сессиям читать данные, но запрещает кому-либо их модифицировать.

При использовании фразы *INTO* помните о следующих правилах. Во-первых, файл не должен существовать, так как функциональность перезаписи не поддерживается. Во-вторых, каждый созданный таким образом файл доступен любому пользователю, подключившемуся к серверу. (Созданный командой *SELECT ... INTO OUTFILE* файл можно затем загрузить командой *LOAD DATA INFILE*.)

Вы можете использовать следующие дополнительные опции для управления форматом файла:

- *ESCAPED BY*
- *FIELDS TERMINATED BY*
- *LINES TERMINATED BY*
- *OPTIONALLY ENCLOSED BY*

Следующий пример иллюстрирует применение этих опций, сохраняя результат запроса в файл со значениями, разделенными запятыми:

```
SELECT job_id, emp_id, lname+fname
INTO OUTFILE "/tmp/employees.text"
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
```

```
LINES TERMINATED BY "\n"
FROM employee;
```

MySQL поддерживает оператор *SELECT* без фразы *FROM* для выполнения арифметических вычислений. Например, следующие запросы являются корректными:

```
SELECT 2 + 2;
SELECT 565 - 200;
SELECT (365 * 2) * 52;
```

Для совместимости с Oracle в MySQL поддерживается псевдотаблица **dual**:

```
SELECT 565 - 200 FROM dual;
```

MySQL очень гибок в поддержке соединений. Вы можете использовать несколько вариантов синтаксиса: например, вы можете явно объявить соединения во фразе *JOIN*, но условие соединения указать в *WHERE*. На других платформах вам нужно придерживаться одного варианта соединения, и нельзя смешивать разные варианты синтаксиса в одном запросе. Однако мы считаем смешивание различных вариантов синтаксиса плохим подходом, поэтому в наших примерах пишем соединения в стиле ANSI.

MySQL поддерживает следующие варианты синтаксиса *JOIN*:

```
FROM таблица1, таблица2
FROM таблица1 {
  [CROSS] JOIN таблица2 |
  STRAIGHT_JOIN таблица2 |
  INNER JOIN таблица2 [ {ON условие_соединения|
    USING (список_столбцов)} ] |
  LEFT [OUTER] JOIN таблица2 {ON условие_соединения |
    USING (список_столбцов)} |
  NATURAL [LEFT [OUTER]] JOIN таблица2 |
  RIGHT [OUTER] JOIN таблица2 [ {ON условие_соединения |
    USING (список_столбцов)} ]|
  NATURAL [RIGHT [OUTER]] JOIN таблица2 }
```

где:

[CROSS] JOIN таблица2

Извлекает все записи из обеих таблиц. Обратитесь к разделу с описанием фразы JOIN за информацией о различных типах соединений.

STRAIGHT_JOIN таблица2

Извлекает записи из обеих таблиц, как это делается и при обычных соединениях, но с одной особенностью. Обычно оптимизатор запросов выбирает определенный порядок соединения таблиц, но иногда выбирается не самый оптимальный вариант. Ключевое слово *STRAIGHT_JOIN* требует от оптимизатора соединять таблицы в том порядке, в котором они перечислены в запросе.

INNER JOIN

Получает только те записи из таблиц, для которых есть соответствующие записи в противоположных таблицах. Обратите внимание, что синтаксис *FROM таблица1, таблица2* является тем же внутренним соединением. При использо-

вании синтаксиса *FROM таблица1, таблица2* не забудьте в *WHERE* указать условие соединения.

NATURAL

Является короткой записью соединения, эквивалентного использованию *USING* для всех столбцов, имеющих одинаковые имена в обеих таблицах. (Пример с *USING* приводится далее.)

LEFT [OUTER] JOIN

Извлекает все записи из левой таблицы, а также присоединенные записи из правой таблицы. Если для определенной записи из левой таблицы не найдено записи в правой таблице, то в соответствующих столбцах будет значение NULL. С помощью *LEFT OUTER JOIN* извлекаются все записи из левой таблицы вне зависимости от того, есть ли для них соответствующие записи в правой таблице.

RIGHT [OUTER] JOIN

Извлекает все записи из правой таблицы, а также присоединенные записи из левой таблицы. *RIGHT JOIN* – это то же самое левое соединение, записанное в противоположном порядке. Рекомендуется не смешивать в запросе левые и правые соединения, так как это усложняет чтение и понимание запросов, а оформлять все внешние соединения либо как левое, либо как правое.

В MySQL есть интересная альтернатива оператору *SELECT* – оператор *HANDLER*. *HANDLER* выглядит и используется так же, как и *SELECT*, но возвращает данные значительно быстрее, так как работает в обход движка запросов MySQL. Но так как *HANDLER* не является стандартным оператором SQL, то дополнительную информацию о нем ищите в документации.

Oracle

В Oracle есть большое число расширений оператора *SELECT* по сравнению с ANSI. Например, так как в Oracle поддерживаются вложенные (nested) и секционированные таблицы (смотрите описание оператора *CREATE TABLE*), то оператор *SELECT* поддерживает запросы к следующим типам структур:

```
[WITH имя_запроса AS (подзапрос)[, ...]]
SELECT { {[ALL | DISTINCT]} | [UNIQUE] }
  [подсказки_оптимизатору] извлекаемый_элемент
  [AS псевдоним][, ...]
[INTO {переменная[, ...] | запись}]
FROM {[ONLY] {[схема.][таблица | представление |
  материализованное_представление]}[@связь_базы_данных]
    [AS [OF] {SCN | TIMESTAMP} выражение] |
  подзапрос [WITH {READ ONLY | CHECK OPTION
    [CONSTRAINT имя_ограничения]}] |
  [[VERSIONS BETWEEN {SCN | TIMESTAMP}
    {выражение | MINVALUE} AND
    {выражение | MAXVALUE}] AS OF {SCN | TIMESTAMP}
    выражение] |
  TABLE (столбец_вложенной_таблицы) [(+)]
  {[PARTITION (имя_секции) | SUBPARTITION (имя_подсекции)]}
  [SAMPLE [BLOCK] [процент_выборки] [SEED
```

```
      (начальное_значение]]} [AS псевдоним]
    [, ...]
  [ [тип_соединения] JOIN условие_соединения
    [PARTITION BY выражение[, ...]] ]
  [WHERE условие_поиска
    [ {AND | OR} условие_поиска[, ...] ]
  [[START WITH значение] CONNECT BY [PRIOR] условие]]
  [GROUP BY выражение_группировки [HAVING условие_поиска]]
  [MODEL модель]
  [ORDER [SIBLINGS] BY значение_сортировки {[ASC | DESC]}
    {[NULLS FIRST | NULLS LAST]}]
  [FOR UPDATE [OF [схема.][таблица.]столбец][, ...]
    {[NOWAIT | WAIT (целое_число)]}]
```

Фразы оператора соответствуют стандарту ANSI, если об отличиях не сказано дополнительно. То же самое касается и отдельных фраз оператора. Например, в Oracle фраза *GROUP BY* почти соответствует стандарту, включая такие элементы, как *CUBE*, *ROLLUP*, *GROUPING SETS*, конкатенированные наборы группировок и *HAVING*.

Параметры следующие:

WITH имя_подзапроса AS (подзапрос)[, ...]

Объявляет именованный подзапрос. Oracle позволяет объявить именованный подзапрос, а затем ссылаться на него в запросе. (В целях оптимизации этот подзапрос может обрабатываться как вложенный табличный подзапрос или как временная таблица.)

ALL | DISTINCT | UNIQUE

ALL и *DISTINCT* работают так, как этого требует стандарт ANSI (смотрите описание синтаксиса ANSI SQL, приведенное ранее). *UNIQUE* является синонимом *DISTINCT*. *DISTINCT* нельзя использовать со столбцами типа *LOB*.

подсказка_оптимизатору

Дает указание оптимизатору запросов на выполнение определенных шагов. Например, с помощью подсказок вы можете использовать в запросе индекс, который по умолчанию не был бы использован. За детальной информацией по подсказкам оптимизатору обращайтесь к документации.

извлекаемый_элемент

Может быть выражением или столбцом именованного запроса, таблицы, представления или материализованного представления в формате *[схема.][таблица.]столбец*. Если вы не указали схему, то подразумевается текущая схема. Oracle позволяет использовать именованные подзапросы (объявляемые с помощью *WITH*) так же, как и обычные вложенные табличные подзапросы. Этот подход называется разложением на подзапросы (subquery factoring). Вы также можете использовать обычные подзапросы и звездочку (*) для получения всех столбцов.

INTO {переменная[, ...] | запись}

Сохраняет значения результирующего множества в переменные или записи PL/SQL.

FROM [ONLY]

Указывает таблицу, представление, материализованное представление, секцию или подзапрос, из которых извлекаются строки. Ключевое слово *ONLY* используется только для представлений, входящих в иерархии, и позволяет выбрать записи только из указанного представления, но не его дочерних представлений.

AS [OF] {SCN | TIMESTAMP} выражение

Позволяет выполнять ретроспективные запросы с указанием системного номера изменения (*SCN*) или значения типа *TIMESTAMP*. Запрос извлекает записи в том виде, в котором они были на указанный момент времени или к указанному системному изменению. (Эта функциональность также может быть использована на уровне сессии с помощью пакета *DBMS_FLASHBACK*.) Значение *SCN* должно быть целым числом, а *TIMESTAMP* должно быть временной меткой. Ретроспективные запросы нельзя использовать для присоединенных серверов.

подзапрос [WITH {READ ONLY | CHECK OPTION [CONSTRAINT имя_ограничения]}]

Рассматривается отдельно, так как Oracle позволяет дополнительно управлять использованием подзапроса. *WITH READONLY* указывает, что целевой подзапрос нельзя обновлять. *WITH CHECK OPTION* указывает, что при обновлениях подзапроса строки должны оставаться в результирующем множестве подзапроса. *WITH CONSTRAINT* создает для таблицы ограничение *CHECK OPTION* с указанным именем. Обратите внимание, что *WITH CHECK OPTION* и *WITH CONSTRAINT* обычно используются в операторах *INSERT...SELECT*.

[[VERSIONS BETWEEN {SCN | TIMESTAMP} {выражение | MINVALUE} AND{ выражение | MAXVALUE}] AS OF {SCN | TIMESTAMP} выражение]

Позволяет использовать ретроспективные запросы для получения истории изменений, выполненных в таблице, представлении или материализованном представлении. Псевдостолбец *VERSIONS_XID* показывает идентификатор транзакции, выполнившей изменение. Для ретроспективных запросов требуется указывать либо системный номер изменения (*SCN*), либо временную метку (*TIMESTAMP*). (Ретроспективные запросы на уровне сессии можно выполнять с помощью пакета *DBMS_FLASHBACK*.)

Опциональная подфраза *VERSIONS BETWEEN* используется для получения нескольких версий данных в интервале, задаваемом значениями *SCN* или временными метками либо ключевыми словами *MINVALUE* и *MAXVALUE*. Без этой фразы возвращается только одна предыдущая версия данных. (В Oracle поддерживаются псевдостолбцы с дополнительной информацией о версиях.)

Фраза *AS OF*, описанная ранее в этом списке, определяет *SCN* или момент времени, на который выполняется запрос с фразой *VERSIONS*.

Вы не можете использовать ретроспективные запросы с фразой *VERSIONS* для временных таблиц, внешних таблиц, кластерных таблиц и представлений.

TABLE

Используется при обращениях к вложенным таблицам в иерархии.

PARTITION

Ограничивает запрос определенной секцией таблицы. То есть строки извлекаются только из указанной секции, но не всей таблицы.

SUBPARTITION

Ограничивает запрос определенной подсекцией таблицы. Это уменьшает ввод/вывод, так как строки извлекаются только из указанной секции, но не всей таблицы.

SAMPLE [BLOCK] [процент_выборки] [SEED (начальное_значение)]

Позволяет выбрать не все строки таблицы, а лишь определенный процент строк, либо блоков таблицы. *BLOCK* указывает, что используется случайная выборка блоков. Процент выборки может иметь любое значение в диапазоне от 0.000001 до 99. Опциональная фраза *SEED* позволяет сделать результат более стабильным и повторимым. Если вы будете использовать одинаковое начальное значение (может быть от 0 до 4294967295), то Oracle будет по возможности возвращать одинаковый результат. Без использования *SEED* результат выборки будет варьироваться от запроса к запросу. *SAMPLE* можно использовать только в запросах, обращающихся к одной таблице.

JOIN

Соединяет результирующие наборы из двух таблиц в одном запросе. Детально описывается далее.

PARTITION BY выражение[, ...]

Используется для выполнения специального типа запроса с так называемым *секционированным внешним соединением*, расширяющим обычный синтаксис соединений выполнением внешнего соединения к секции из одного или нескольких столбцов, указанных в выражении. Этот тип запросов используется для получения разреженных данных по определенным измерениям, которые при обычных запросах исключались бы из выборки. Примеры приводятся в следующем разделе «Секционированные внешние соединения». Фраза *PARTITION BY* может быть использована в любой части внешнего соединения, что приводит к объединению внешних соединений с каждой секцией секционированной выборки и таблицы, находящейся с другой стороны условия соединения. (Если эта фраза опущена, то Oracle обрабатывает весь результирующий набор как одну секцию.) *PARTITION BY* нельзя использовать совместно с *FULL OUTER JOIN*.

WHERE... [[START WITH значение] CONNECT BY [PRIOR] условие]

Отфильтровывает строки в результирующем множестве. Oracle позволяет с помощью фразы *START WITH* использовать хранящуюся в таблице информацию об иерархиях. *START WITH* идентифицирует строки, являющиеся корневыми. *CONNECT BY* определяет взаимосвязь между записями-родителями и потомками. Ключевое слово *PRIOR* позволяет ссылаться на родительскую запись.

В иерархических запросах можно использовать псевдостолбец *LEVEL* для определения корневых записей (1), детей (2), внуков (3) и т. д. Также можно использовать псевдостолбцы *CONNECT_BY_ISCYCLE* и *CONNECT_BY_ISLEAF*. В иерархических запросах недопустимы фразы *ORDER BY* и *GROUP BY*.

Вы можете указать правило сортировки записей, имеющих одного и того же родителя, с помощью *ORDER SIBLINGS BY*.

ORDER [SIBLINGS] BY выражение_сортировки {NULLS FIRST | NULLS LAST}

Сортирует результат запроса по указанному выражению. Выражение может состоять из имен столбцов, псевдонимов, порядковых номеров столбцов. Фраза *ORDER SIBLINGS BY* используется в иерархических запросах с *CONNECT BY*. *ORDER SIBLINGS BY* сохраняет порядок, заданный иерархией, и выполняет сортировку только тех записей, которые имеют одинаковых родителей (то есть находятся на одном уровне иерархии). Опции *NULLS FIRST* и *NULLS LAST* указывают, что значения NULL должны при сортировке идти первыми или последними, соответственно.

FOR UPDATE [OF [схема.]... {[NOWAIT | WAIT (целое_число)]}

Блокирует строки результата, так что они не могут быть заблокированы или изменены другими пользователями, пока вы не завершите транзакцию. Фразу *FOR UPDATE* нельзя использовать в подзапросах, в запросах с *DISTINCT* и *GROUP BY*, а также в запросах с агрегатными функциями и функциями работы с множествами. В иерархических запросах дочерние записи не блокируются при блокировании родительских записей. *OF* блокирует только строки указанных таблиц или представлений, без *OF* блокируются строки всех таблиц и представлений, указанных во *FROM*. Используемые в *OF* столбцы не существенны, но там следует указывать имена столбцов, а не псевдонимы. Если при выполнении запроса обнаруживается, что записи уже заблокированы другой сессией, то *NOWAIT* сразу же завершает запрос, а *WAIT* ждет освобождения блокировки либо до бесконечности, либо не более заданного числа секунд. Если не указано ни *WAIT*, ни *NOWAIT*, то по умолчанию Oracle будет ждать, пока блокировка не освободится.

В отличие от других платформ, Oracle не допускает операторы *SELECT* без фразы *FROM*. Следующий запрос в Oracle некорректен:

```
SELECT 2 + 2;
```

В качестве обходного решения в Oracle предлагается таблица **dual**. Используйте эту таблицу всегда, когда вам не нужно извлекать данные из обычных таблиц, а вы просто хотите выполнить вычисление. Следующие запросы корректны:

```
SELECT 2 + 2
FROM dual;
SELECT (((52-4) * 5) * 8)
FROM dual;
```

Реализация оператора *SELECT* в Oracle позволяет легко извлекать данные из таблиц. Например, в Oracle можно использовать именованные подзапросы. Именованный подзапрос – это просто псевдоним для запроса, с помощью которого упрощается написание сложных запросов с множеством подзапросов. Например:

```
WITH pub_costs AS
  (SELECT pub_id, SUM(job_lvl) dept_total
   FROM employees e
   GROUP BY pub_id),
avg_costs AS
```

```
   (SELECT SUM(dept_total)/COUNT(*) avg
    FROM employee)
SELECT * FROM pub_costs
WHERE dept_total > (SELECT avg FROM avg_cost)
ORDER BY department_name;
```

В этом примере объявляются два подзапроса – **pub_costs** и **avg_costs** – которые используются затем в основном запросе. Именованные подзапросы фактически эквивалентны обычным подзапросам, но их не требуется писать полностью при каждом использовании.

Oracle позволяет с помощью фразы *PARTITION* извлекать строки только из указанной секции секционированной таблицы либо получать статистическую выборку (размер выборки задается в процентах от общего количества строк или блоков) с помощью *SAMPLE*:

```
SELECT *
FROM sales PARTITION (sales_2004_q3) sales
WHERE sales.qty > 1000;
SELECT *
FROM sales SAMPLE (12);
```

Ретроспективные запросы Oracle – это функциональность, позволяющая получить данные такими, каким они были на определенный момент времени. Например, вы можете получить зарплату каждого сотрудника во вчерашнем варианте, до того, как в базе данных были выполнены значительные изменения:

```
SELECT job_lvl, lname, fname
FROM employee
  AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY);
```

Другим интересным расширением Oracle являются иерархические запросы. Иерархические запросы используются со специально спроектированными таблицами и возвращают результат в порядке, определяемом иерархией. Например, следующий запрос возвращает имена сотрудников и их позицию в иерархии (столбец **org_chart**), а также идентификаторы сотрудника и его менеджера, и описание должности:

```
SELECT LPAD(' ',2*(LEVEL-1)) || lname AS org_chart,
  emp_id, mgr_id, job_id
FROM employee
START WITH job_id = 'Chief Executive Officer'
CONNECT BY PRIOR emp_id = mgr_id;

ORG_CHART      EMP_ID MGR_ID JOB_ID
-------------- ------ ------ ------------------------
Cramer         101    100    Chief Executive Officer
  Devon        108    101    Business Operations Mgr
    Thomas     109    108    Acquisitions Manager
    Koskitalo  110    108    Productions Manager
    Tonini     111    108    Operations Manager
  Whalen       200    101    Admin Assistant
  Chang        203    101    Chief Financial Officer
    Gietz      206    203    Comptroller
Buchanan       102    101    VP Sales
   Callahan    103    102    Marketing Manager
```

В этом примере *CONNECT BY* указывает, что значение **mgr_id** записи-потомка должно совпадать со значением **emp_id** в родительской записи, а *START WITH* определяет строки, с которой разворачивается иерархия.

Oracle поддерживает следующие варианты синтаксиса *JOIN* (дополнительная информация приводится в описании подфразы *JOIN*):

```
FROM таблица1 {
  CROSS JOIN таблица2 |
  INNER JOIN таблица2 [ {ON условие_соединения|
  USING (список_столбцов)} ] |
LEFT [OUTER] JOIN таблица2 [ {ON условие_соединения
  | USING (список_столбцов)} ] |
NATURAL [LEFT [OUTER]] JOIN таблица2 |
RIGHT [OUTER] JOIN таблица2 [ {ON условие_соединения
  | USING (список_столбцов)} ]|
NATURAL [RIGHT [OUTER]] JOIN таблица2
FULL [OUTER] JOIN таблица2 }
```

[CROSS] JOIN

Получает все записи из обеих таблиц. Синтаксически это то же самое, что указание *FROM таблица1, таблица2* без условия соединения в *WHERE*.

INNER JOIN

Получает только те записи из таблиц, для которых есть соответствующие записи в противоположных таблицах. Обратите внимание, что синтаксис *FROM таблица1, таблица2* с условием соединения в *WHERE* является тем же внутренним соединением.

NATURAL

Является короткой записью соединения, эквивалентного использованию *USING* со всеми столбцами, имеющими одинаковые названия в обеих таблицах. (Будьте осторожны, так как столбцы могут иметь одинаковые названия, но быть разного типа или хранить разные данные.) Столбцы типа *LOB* в натуральных соединениях использовать нельзя. Использование столбца типа *LOB* или столбца-коллекции вызовет ошибку.

LEFT [OUTER] JOIN

Извлекает все записи из левой таблицы, а также присоединенные записи из правой таблицы. Если для определенной записи из левой таблицы не найдено записи в правой таблице, то в соответствующих столбцах будет значение NULL. С помощью *LEFT OUTER JOIN* извлекаются все записи из левой таблицы вне зависимости от того, есть ли для них соответствующие записи в правой таблице. Например:

```
SELECT j.job_id, e.lname
FROM jobs j
LEFT OUTER JOIN employee e ON j.job_id = e.job_id
ORDER BY d.job_id
```

RIGHT [OUTER] JOIN

Извлекает все записи из правой таблицы вне зависимости от того, есть ли для определенной записи соответствующая запись в левой таблице. Правое соеди-

нение похоже на левое, за исключением того, что необязательной таблицей является левая.

FULL [OUTER] JOIN

Возвращает все данные из обеих таблиц вне зависимости от того, была ли определенная запись соединена с записью из противоположной таблицы. Если для какой-либо строки нет соответствия в соединяемой таблице, то в результирующем наборе будут значения NULL.

ON условие_соединения

Определяет условие соединения двух или более таблиц. В условии необходимо указать пары столбцов из таблиц, которые должны иметь одинаковые значения. Несколько условий объединяются с помощью *AND*.

USING (список_столбцов)

Является альтернативой соединения с помощью *ON*. Вместо описания условий соединения вы можете перечислить столбцы, существующие в обеих таблицах. Соединение выполняется по равенству значений в этих столбцах. Столбцы должны иметь одинаковые названия в обеих таблицах и не быть уточнены именами или псевдонимами таблиц. В *USING* нельзя указывать столбцы типа *LOB*. Следующие два запроса (один использует *USING*, другой – явное условие соединения) имеют одинаковый результат:

```
SELECT column1
FROM foo
LEFT JOIN poo USING (column1, column2);
SELECT column1
FROM foo
LEFT JOIN poo ON foo.column1 = poo.column1
AND foo.column2 = poo.column2;
```

Обратите внимание, что в старых версиях Oracle при описании условий внешних соединений в *WHERE* использовался маркер (+). Соединения в стиле ANSI начали поддерживаться только с версии Oracle9i Release1. Поэтому вам может встречаться действующий код, использующий старый синтаксис. Например, следующий запрос семантически эквивалентен предыдущему примеру левого внешнего соединения:

```
SELECT j.job_id, e.lname
FROM jobs j, employee e
WHERE j.job_id = e.job_id (+)
ORDER BY d.job_id
```

Старый вариант синтаксиса более сложен для понимания и чаще приводит к ошибкам. Мы настоятельно рекомендуем использовать стандартный синтаксис ANSI.

Секционированные внешние соединения

Секционированные внешние соединения позволяют легко запрашивать разреженные данные, в то время как сделать это другим образом не так просто. (Стандарт ANSI описывает секционированные внешние соединения, но Oracle является первой платформой, в которой появилась их поддержка.) Например, в таблице **product** есть информация о всех продуктах, которые мы производим, а в таб-

лице **manufacturing** хранится информация о производстве. Так как мы не каждый день производим все товары, то соединение этих двух таблиц может быть разреженным по времени:

```
SELECT manufacturing.time_id AS time, product_name AS name,
    quantity AS qty
FROM product
PARTITION BY (product_name)
RIGHT OUTER JOIN times ON (manufacturing.time_id =
    product.time_id)
WHERE manufacturing.time_id
  BETWEEN TO_DATE('01/10/05', 'DD/MM/YY')
    AND TO_DATE('06/10/05', 'DD/MM/YY')
ORDER BY 2, 1;
```

Результат запроса следующий:

```
time      name           qty
--------- ----------     ----------
01-OCT-05 flux capacitor  10
02-OCT-05 flux capacitor
03-OCT-05 flux capacitor
04-OCT-05 flux capacitor
05-OCT-05 flux capacitor
06-OCT-05 flux capacitor  10
06-OCT-05 flux capacitor  8
01-OCT-05 transmorgrifier 10
01-OCT-05 transmorgrifier 15
02-OCT-05 transmorgrifier
03-OCT-05 transmorgrifier
04-OCT-05 transmorgrifier 10
04-OCT-05 transmorgrifier 11
05-OCT-05 transmorgrifier
06-OCT-05 transmorgrifier
```

Этот запрос и его результат показывают, как секционированные внешние соединения позволяют легко получить результат в тех случаях, когда сделать это другим образом проблематично из-за разреженных данных.

Ретроспективные запросы

В Oracle 9i и более поздних версиях поддерживаются ретроспективные запросы – запросы, позволяющие получить предыдущие значения данных. В следующем примере мы выполним для таблицы обычный запрос, затем обновим таблицу оператором *UPDATE*, а затем получим предыдущую версию данных ретроспективным запросом. Итак, сначала выполним обычный запрос:

```
SELECT salary FROM employees
WHERE last_name = 'McCreary';
```

Результат:

```
SALARY
----------
3800
```

Теперь выполним изменения в таблице и опять запросим текущее значение:

```
UPDATE employees SET salary = 4000
WHERE last_name = 'McCreary ';
SELECT salary FROM employees
WHERE last_name = 'McCreary ';
```

Результат:

```
SALARY
----------
4000
```

Наконец, выполним ретроспективный запрос, чтобы получить значение, которое было в прошлом:

```
SELECT salary FROM employees
AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY)
WHERE last_name = 'McCreary';
```

Результат:

```
SALARY
----------
3800
```

Для более скрупулезного изучения мы можем запросить все значения данных, которые существовали в указанный период, предположим, за последние два дня:

```
SELECT salary FROM employees
VERSIONS BETWEEN TIMESTAMP
SYSTIMESTAMP - INTERVAL '1' MINUTE AND
SYSTIMESTAMP - INTERVAL '2' DAY
WHERE last_name = 'McCreary';
```

Результат:

```
SALARY
----------
4000
3800
```

Фраза MODEL

В Oracle 10g появилась мощная опция оператора *SELECT* – фраза *MODEL*, позволяющая выполнять с набором данных манипуляции, похожие на те, что обычно делают в электронных таблицах. Фактически эта фраза призвана снять необходимость в извлечении данных в электронные таблицы (например, в MS Excel) для ручной обработки. Эта фраза создает многомерный массив, в котором ячейки адресуются значениям измерений. Например, вы могли бы создать массив по продукту и времени, адресуя значения массива по комбинации значений этих двух измерений. Вы можете создавать правила, аналогичные формулам Excel, выполняемым для изменения текущих значений, создания новых значений или даже строк в вашей модели.

Фраза *MODEL* должна находиться после *GROUP BY* и *HAVING*, но перед *ORDER BY*. Предыдущее описание синтаксиса оператора *SELECT* для Oracle показывает только положение фразы в операторе, а детали приводятся далее:

```
MODEL
  [{IGNORE | KEEP} NAV]
  [UNIQUE {DIMENSION | SINGLE REFERENCE}]
  [ RETURN {UPDATED | ALL} ]
  [ REFERENCE имя_справочной_модели ON (подзапрос)
    [PARTITION BY (столбец [AS псевдоним][, ...])]
    DIMENSION BY (столбец [AS псевдоним][, ...])
    MEASURES (столбец [AS псевдоним][, ...])
  [{IGNORE | KEEP} NAV]
  [UNIQUE {DIMENSION | SINGLE REFERENCE}] ]
[MAIN имя_модели]
  [PARTITION BY (столбец [AS псевдоним][, ...])]
  DIMENSION BY (столбец [AS псевдоним][, ...])
  MEASURES (столбец [AS псевдоним][, ...])
  [{IGNORE | KEEP} NAV]
  [UNIQUE {DIMENSION | SINGLE REFERENCE}]]
правила
  [RULES [UPSERT [ALL] | UPDATE]
  [{AUTOMATIC | SEQUENTIAL} ORDER]]
  [ITERATE (целое_число) [UNTIL (условие)]]
  ( [ {UPSERT [ALL] | UPDATE } ]
  мера [...]
  [FOR { измерение | ( измерение[, ...] ) }
  { [IN ({подзапрос | литерал[, ...]})] |
    [LIKE шаблон] FROM начало TO конец
      {INCREMENT | DECREMENT} шаг }[, ...]
  [ORDER [SIBLINGS] BY (столбец_сортировки [ASC | DESC]
    [NULLS FIRST | NULLS_LAST][, ...])]]
  = выражение[, ...] )
```

Параметры фразы *MODEL* следующие:

{IGNORE|KEEP} NAV

Указывает режим работы с неопределенными и отсутствующими значениями (*NAV*). Либо они выводятся как NULL (опция *KEEP*), либо заменяются на значения по умолчанию (опция *IGNORE*): *0* для числовых типов, *01-Января-2000* для временных типов, пустую строку для символьных типов и NULL для всех остальных типов.

UNIQUE {DIMENSION|SINGLE REFERENCE}

Указывает область, для которой проверяется, что ссылки на ячейки соответствуют одиночным значениям. *DIMENSION* требует, чтобы любая ссылка на ячейку (неважно, в левой или в правой части правила) соответствовала одиночному значению. *SINGLE REFERENCE* проверяет только ссылки на ячейки в правых частях правил.

RETURN {UPDATED|ALL} ROWS

Указывает, возвращаются ли из модели все строки (*ALL*) или только обновленные (*UPDATED*).

имя_справочной_модели

Определяет модель, в которой вы не можете выполнять вычисления, но можете использовать ее значения в основном запросе.

имя_справочной_модели *ON* (*подзапрос*)

Указывает имя и источник строк для справочной модели.

псевдоним

Указывает псевдоним для секции.

DIMENSION BY (*столбец* [,*столбец*...])

Указывает измерения модели. Значения этих столбцов используются для индексации значений, являющихся ячейками многомерного пространства.

MEASURES (*столбец* [,*столбец*...])

Указывает значения, связанные с каждой уникальной комбинацией измерений (то есть ячейки модели).

PARTITION BY [( ] *столбец* [,*столбец*...][ )]

Разбивает модель на независимые секции по указанным столбцам. Вы можете не разбивать на секции справочные модели.

имя_главной_модели

Указывает модель, в которой выполняется работа. Строки из соответствующего оператора *SELECT* наполняют модель, затем применяются правила и возвращается результат.

MAIN *имя_модели*

Начинает описание главной модели и присваивает ей имя.

RULES [*UPSERT*|*UPDATE*]

Указывает, могут ли правила создавать новые ячейки и обновлять существующие (*UPSERT*) либо только обновлять существующие ячейки (*UPDATE*). Если вы хотите создавать в модели новые строки, то используйте *UPSERT*. (*UPSERT* является значением по умолчанию.) Вы также можете задавать этот параметр на уровне правил (смотрите описание синтаксиса правил).

{*AUTOMATIC*|*SEQUENTIAL*} *ORDER*

AUTOMATIC указывает, что оптимизатор сам определяет порядок выполнения правил. При *SEQUENTAL* (значение по умолчанию) правила выполняются в порядке их следования.

ITERATE (*целое_число*)

Указывает, сколько раз должен быть выполнен весь набор правил. По умолчанию правила выполняются только один раз.

UNTIL (условие)

Указывает условие, при выполнении которого прекращаются итерации выполнения правил. Тем не менее, вы все равно должны указать максимальное число итераций, чтобы защититься от бесконечного цикла.

мера[...]

Ссылка на одну из мер, перечисленных во фразе *MEASURE*. При ссылке на меру необходимо использовать квадратные скобки как часть синтаксиса. Вы должны указать значения всех измерений (либо явно указав значение, либо через подзапрос), чтобы получить соответствующее значение меры.

FOR ...

Цикл *FOR* для итераций по одному или нескольким измерениям. Цикл *FOR* похож на подзапрос, в котором каждая строка результата соответствует определенной комбинации значений измерений.

значения_измерений

Список значений, полученных из столбцов или выражений, идентифицирующих уникальную ячейку модели.

IN ({подзапрос| литерал[, литерал...]})

Источником значений для цикла *FOR* может быть подзапрос или список литералов.

LIKE шаблон

Позволяет вставлять значения измерений в шаблон. Используйте знак % для указания места, в которое вставляется значение. Например, используйте *FOR x LIKE 'A%B' FROM 1 TO 3 INCREMENT 1* для генерации значений *'A1B'*, *'A2B'*, *'A3B'*.

FROM начало TO конец {INCREMENT | DECREMENT} шаг

Указывает начальное и конечное значение для цикла *FOR*, а также шаг цикла.

ORDER BY (столбец_сортировки[, ...])

Определяет порядок выполнения правила по отношению к ячейкам, указанным в левой части правила. Если вы не используете эту фразу, то порядок применения правила к ячейкам не определен.

Следующие функции предназначены специально для использования во фразе *MODEL*:

CV( ) или CV(столбец_измерения)

Возвращает текущее значение столбца измерения. Может использоваться только в правой части выражения правила. При использовании записи *CV( )* столбец измерения определяется автоматически в соответствии с позицией вызова функции в списке значений измерений.

PRESENTNNV(мера [измерение, измерение...], значение_не_null, значение_null)

В зависимости от того, равнялось ли значение указанной меры NULL на момент начала обработки модели, возвращает соответствующее значение *значение_не_null* или *значение_null*. Эту функцию можно использовать только в правой части выражения правила.

PRESENTV(мера [измерение, измерение...], существовало, не_существовало)

В зависимости от того, существовала ли указанная мера на момент начала обработки модели, возвращает соответствующее значение. Эту функцию можно использовать только в правой части выражения правила. Имейте в виду, что существование меры – это вопрос, совершенно отличный от того, имеет ли мера значение NULL.

ITERATION_NUMBER

Возвращает 0 для первой итерации правил, 1 – для второй и т. д. Эта функция используется, если вы хотите, чтобы правила зависели от номера итерации.

Следующий пример демонстрирует, как фраза *MODEL* дает возможность на базе обычного оператора *SELECT* строить многомерный массив и вычислять новые значения, задавая формулы на уровне строк или массива. Вычисленные значения возвращаются как часть результирующего множества запроса:

```
SELECT SUBSTR(region,1,20) country,
  SUBSTR(product,1,15) product, year, sales
FROM sales_view
WHERE region IN ('USA','UK')
MODEL RETURN UPDATED ROWS
    PARTITION BY (region)
    DIMENSION BY (product, year)
    MEASURES (sale sales)
    RULES (
      sales['Bounce',2006] = sales['Bounce',2005]
        + sales['Bounce',2004],
      sales['Y Box', 2006] = sales['Y Box', 2005],
      sales['2_Products',2006] = sales['Bounce',2006]
        + sales['Y Box',2006] )
ORDER BY region, product, year;
```

В этом примере запрос к материализованному представления **sales_view** возвращает сумму по продажам за несколько лет в регионах *'USA'* и *'UK'*. Фраза *MODEL* используется между *WHERE* и *ORDER BY*. Так как в **sales_view** есть данные за 2004 и 2005 годы, то мы можем указать правила для прогнозирования продаж в 2006 году.

Подфраза *RETURN UPDATED ROWS* ограничивает возвращаемый результат только строками, обновленными или созданными в модели. Затем определяется логическая структура данных с использованием элементов материализованного представления и подфраз *PARTITION BY*, *DIMENSION BY* и *MEASURES*. Подфраза *RULES* затем ссылается на отдельные ячейки, определяемые значениями измерений, так же, как это делается в макроязыках электронных таблиц, с определенными ссылками и использованием диапазонов значений.

Oracle (как и SQL Server, но немного в другом формате) поддерживает так называемые запросы с поворотом (pivot query), не описанные ни в ANSI, ни в ISO. Запрос с поворотом позволяет «повернуть данные на бок», что позволяет извлечь из этих данных новую полезную информацию. В Oracle сначала нужно создать «повернутую» таблицу (pivot table или сводную таблицу). Используя запросы с поворотом, вы можете сделать так, что данные столбца **order_type** станут заголовками столбцов:

```
CREATE TABLE pivot_table AS
SELECT * FROM (SELECT year, order_type, amt FROM sales)
PIVOT SUM(amt) FOR order_type IN ('retail', 'web');

SELECT * FROM pivot_table ORDER BY YEAR;
```

Результат следующий::

```
YEAR RETAIL      WEB
---- ----------- ------
2004 7014.54
2005 9745.12
```

```
2006 16717.88    10056.6
2007 28833.34    39334.9
2008 66165.77    127109.4
```

PostgreSQL

PostgreSQL поддерживает стандартный вариант оператора *SELECT* с *JOIN* и подзапросами. PostgreSQL также позволяет создавать новые временные и постоянные таблицы с помощью синтаксиса *SELECT...INTO*:

```
SELECT [ALL | DISTINCT [ON (извлекаемый_элемент[, ...])]]
  [AS псевдоним [(список_псевдонимов)]][, ...]
[INTO [[TEMP]ORARY] [TABLE] имя_новой_таблицы]
[FROM [ONLY] таблица1[.*] [AS псевдоним][, ...] ]
[ [тип_соединения] JOIN table2 {[ON условие_соединения] |
  [USING (список_столбцов)]}
[WHERE условие_поиска]
[GROUP BY выражение_группировки]
[HAVING условие_поиска]
[ORDER BY выражение_сортировки
  [{ASC | DESC | USING оператор][, ...] }]
[FOR UPDATE [OF столбец[, ...]]
[LIMIT {число_записей | ALL} ][OFFSET [число_записей]] ]
```

где:

ALL | DISTINCT [ON (извлекаемый_элемент[, ...]]

ALL и *DISTINCT* работают так, как это описано в стандарте: *ALL* (используется по умолчанию) возвращает все строки, включая повторяющиеся, а *DISTINCT* удаляет повторяющиеся строки. *DISTINCT ON* удаляет повторяющиеся строки, имеющие одинаковые значения не во всех столбцах, а только в указанных (пример будет дальше).

извлекаемый_элемент

Может быть любым стандартным элементом, описанным в ANSI. Кроме обычной звездочки (*) вы можете использовать формат *таблица*.* для получения всех столбцов определенной таблицы.

AS псевдоним [(список_псевдонимов)]

Объявляет псевдоним или список псевдонимов для одного или нескольких столбцов (или таблиц во фразе *FROM*). *AS* требуется для псевдонимов элементов *SELECT*, но во *FROM* слово *AS* необязательно. (На некоторых платформах слово *AS* полностью необязательно.)

INTO [[TEMP]ORARY] [TABLE] имя_новой_таблицы

Создает новую временную или постоянную таблицу. Для создания временных таблиц, удаляемых при закрытии сессии, можно использовать как *TEMP*, так и *TEMPORARY*. В противном случае создаются постоянные таблицы. Постоянные таблицы должны иметь новые, уникальные имена. Временная таблица может иметь такое же имя, как существующая постоянная таблица. В этом случае по этому имени в текущей сессии будет использоваться временная таблица, а во всех остальных сессиях – постоянная таблица.

FROM [ONLY] таблица1[, ...]

Указывает одну или несколько таблиц, из которых извлекают данные. (Не забудьте указать условии тэта-соединения в *WHERE*, чтобы не получить картезианское произведение таблиц.) PostgreSQL поддерживает наследование таблиц. *ONLY* указывает, что запрос извлекает данные только из основной таблицы, но не из наследников. (Вы можете отключить наследование глобально с помощью команды *SET SQL_Inheritance TO OFF*.) PostgreSQL также поддерживает вложенные табличные подзапросы (подзапросы описываются в отдельном разделе в этой главе). Фраза *FROM* не используется в простых вычислениях:

```
SELECT 8 * 40;
```

PostgreSQL также неявно включит фразу *FROM* в оператор *SELECT*, содержащий столбцы с уточненными именами. Например, следующий вариант запроса допустим (хотя и не рекомендуется):

```
SELECT sales.stor_id WHERE sales.stor_id = '6380';
```

GROUP BY выражение_группировки

В качестве выражения группировки можно использовать имя столбца либо его номер по порядку в списке элементов *SELECT*. Пример, иллюстрирующий это, приводится далее при описании *ORDER BY*.

ORDER BY выражение_сортировки

В качестве выражения сортировки можно использовать имя столбца, его псевдоним, номер столбца по порядку в списке *SELECT*. Например, следующие два запроса функционально эквивалентны:

```
SELECT stor_id, ord_date, qty AS quantity
FROM sales
ORDER BY stor_id, ord_date DESC, qty ASC;
SELECT stor_id, ord_date, qty
FROM sales
ORDER BY 1, 2 DESC, quantity;
```

В операторах *SELECT*, использующих одну таблицу, можно сортировать данные по столбцам, не извлекаемым запросом. Например:

```
SELECT *
FROM sales
ORDER BY stor_id, qty;
```

ASC и *DESC* поддерживаются по стандарту. По умолчанию используется *ASC*. PostgreSQL при сортировке считает NULL наибольшим значением. Поэтому при сортировке по возрастанию NULL идут в конце, а при сортировке по убыванию – в начале.

FOR UPDATE [OF столбец[, ...]] LIMIT {число |ALL} [OFFSET [число]]

LIMIT ограничивает число строк, возвращаемых запросом. Опциональный параметр *OFFSET* позволяет пропустить определенное число записей перед тем, как начать возвращать их клиенту. При использовании *LIMIT* необходимо использовать *ORDER BY*, иначе результат будет непредсказуем. (В Post-

greSQL версии 7.0 и старше *LIMIT/OFFSET* используется как подсказка оптимизатору и влияет на выбираемый план выполнения запроса.)

PostgreSQL поддерживает удобное расширение фразы *DISTINCT* – *DISTINCT ON*. Эта фраза позволяет указать столбцы, которые будут использоваться при определении повторяющихся записей. В этом случае также необходимо использовать *ORDER BY*, так как из группы повторяющихся записей будет оставляться только одна, являющаяся первой в указанном порядке сортировки. Без *ORDER BY* результат будет непредсказуемым. Например:

```
SELECT DISTINCT ON (stor_id), ord_date, qty
FROM sales
ORDER BY stor_id, ord_date DESC;
```

Этот запрос возвращает наиболее свежую запись о продаже по каждому магазину, так как используется сортировка по дате заказа. Без фразы *ORDER BY* нельзя было бы точно сказать, какая запись из группы повторяющихся записей будет оставлена.

PostgreSQL поддерживает следующие типы синтаксиса *JOIN*:

```
FROM таблица1[, ...] {
CROSS JOIN таблица2 |
[INNER] JOIN таблица2 [ {ON условие_соединения |
USING (список_столбцов)} ] |
LEFT [OUTER] JOIN таблица2 [ {ON условие_соединения |
USING (список_столбцов)} ] |
NATURAL [LEFT [OUTER]] JOIN таблица2 |
RIGHT [OUTER] JOIN таблица2 [ {ON условие_соединения |
USING (список_столбцов)} ]|
NATURAL [RIGHT [OUTER]] JOIN таблица2
FULL [OUTER] JOIN таблица2 }
```

где:

[CROSS] JOIN

Извлекает все записи из обеих таблиц. Синтаксически это то же самое, что и *FROM таблица1, таблица2*. Результатом является картезианское произведение (обычно очень плохая вещь!).

INNER JOIN

Извлекает только те записи из обеих таблиц, которые имеют соответствующие значения в обеих таблицах. Обратите внимание, что синтаксис *FROM таблица1, таблица2* семантически эквивалентен внутреннему соединению с неуказанным условием соединения (то есть картезианскому произведению). При использовании синтаксиса *FROM таблица1, таблица2* не забудьте указать условие соединения в *WHERE*. При неуказанном типе соединения подразумевается внутреннее соединения.

NATURAL

Является короткой записью, эквивалентной соединению с фразой *USING*, в которой перечислены все столбцы, имеющие одинаковые названия в обеих таблицах. (Будьте осторожны, так как может оказаться, что столбцы имеют одинаковые названия, но разный тип данных или хранят разные значения.)

LEFT [OUTER] JOIN

Извлекает все записи из левой таблицы, а также присоединенные записи из правой таблицы. Если для определенной записи из левой таблицы не найдено записи в правой таблице, то в соответствующих столбцах будет значение NULL. С помощью *LEFT OUTER JOIN* извлекаются все записи из левой таблицы вне зависимости от того, есть ли для них соответствующие записи в правой таблице. Например:

```
SELECT j.job_id, e.lname
FROM jobs j
LEFT OUTER JOIN employee e ON j.job_id = e.job_id
ORDER BY d.job_id
```

RIGHT [OUTER] JOIN

Извлекает все записи из правой таблицы вне зависимости от того, были ли к ним присоединены соответствующие записи из левой таблицы. Это аналог *LEFT JOIN*, выполняемый в противоположном направлении.

FULL [OUTER] JOIN

Получает все записи, как при внутреннем соединении, а также все записи из обеих таблиц, к которым не были присоединены записи из противоположных таблиц.

ON условие_соединения

Определяет условие соединения двух или более таблиц. В условии следует указать пары столбцов из таблиц, которые должны иметь одинаковые значения. Несколько условий объединяются с помощью *AND*.

SQL Server

SQL Server поддерживает основную функциональность оператора *SELECT* согласно стандарту ANSI, включая различные типы соединений. Также поддерживаются некоторые расширения *SELECT*: подсказки оптимизатору, фразы *INTO* и *TOP*, расширения *GROUP BY*, *COMPUTE* и *WITH OPTIONS*:

```
[WITH табличное_выражение[, ...]]
SELECT {[ALL | DISTINCT] | [TOP число [PERCENT]
  [WITH TIES]]} извлекаемый_элемент [AS псевдоним]
[INTO имя_новой_таблицы]
[FROM {[функция_наборов_строк | таблица1[, ...]]}
  [AS псевдоним]]
[ [тип_соединения] JOIN таблица2[ON условие_соединения] ]
[WHERE условие_поиска]
[GROUP BY [GROUPING SETS]
  {столбец_группировки[, ...] | ALL}]
[WITH { CUBE | ROLLUP }]
[HAVING условие_поиска]
[ORDER BY выражение_сортировки [ ASC | DESC ]]
[COMPUTE {агрегация (выражение)}[, ...]
  [BY выражение[, ...]]]
[FOR {BROWSE | XML {RAW | AUTO | EXPLICIT}
  [, XMLDATA][, ELEMENTS][, BINARY base64]}]
[OPTION (подсказка_оптимизатору[, ...])]
```

где:

WITH табличное_выражение

Объявляет временный именованный набор строк, построенный из результата оператора *SELECT*.

TOP число [PERCENT] [WITH TIES]

Указывает, что запрос должен вернуть только указанное число строк (или процент от общего числа строк, если используется *PERCENT*). Фраза *WITH TIES* используется только с *ORDER BY*. *WITH TIES* указывает, что помимо строк, указанных в *TOP*, возвращаются дополнительно стоки из основной выборки, имеющие те же значения в столбцах, перечисленных в *ORDER BY*, что и последняя строка из *TOP*.

INTO имя_новой_таблицы

Создает из результата запроса новую таблицу. Вы можете с помощью этой опции создавать как временные, так и постоянные таблицы. (Ознакомьтесь с правилами, касающимися создания постоянных и временных таблиц в SQL Server, приведенными в этом разделе.) Выполнение оператора *SELECT...INTO* не журналируется, поэтому оператор выполняется быстро, но на него не влияют *COMMIT* и *ROLLBACK*.

FROM {[функция_наборов_строк | таблица1[, ...]]}

Поддерживает стандартную функциональность, описанную в ANSI, включая вложенные табличные подзапросы. SQL Server также поддерживает так называемые *функции наборов строк*. Эти функции позволяют извлекать данные из нестандартных источников, таких как XML потоки, файловые структуры полнотекстового поиска (в которых SQL Server хранит, например, документы MS Word и презентации MS PowerPoint) и внешних источников (таких как файлы MS Excel).

(За полным перечнем поддерживаемых функций наборов строк обращайтесь к документации.) В числе множества других поддерживаются следующие функции:

CONTAINSTABLE

Возвращает таблицу, построенную на базе другой таблицы, содержащей как минимум один полнотекстовый индекс по столбцу типа *TEXT* или *NTEXT*. Могут быть использованы различные алгоритмы поиска: точное сравнение, нечеткий поиск, взвешенное совпадение (weighted-match), совпадение с заданным расстоянием между словами (proximity-match). Построенная таблица может быть использована как любая другая обычная таблица.

FREETEXTTABLE

Работает аналогично *CONTAINSTABLE*, за исключением того, что записи строятся на базе поиска произвольной строки. Функция *FREETEXTTABLE* очень полезна для нерегламентированных запросов к полнотекстовым таблицам, но менее точна, чем *CONTAINSTABLE*.

OPENDATASOURCE

Позволяет получать доступ к внешним данным (например, таблицам MS Excel или Sybase Adaptive Server) через OLE DB без создания присоединен-

ного сервера. Используется только для нечастого нерегламентированного использования. Если вам требуется часто запрашивать данные из внешнего источника, то используйте присоединенный сервер.

OPENQUERY

Выполняет сквозной запрос к присоединенному серверу. Это эффективный способ выполнения вложенных табличных подзапросов к источникам, являющимся внешними по отношению к SQL Server.

OPENROWSET

Выполняет сквозной запрос к внешнему источнику. Эта функция аналогична *OPENDATASOURCE* за исключением того, что *OPENDATASOURCE* лишь открывает источник, но не выполняет *SELECT*. *OPENROWSET* предназначен только для эпизодического, нерегламентированного использования.

OPENXML

Преобразует строку XML в табличный формат, пригодный для запросов.

[GROUP BY [GROUPING SETS] {столбец_группировки[, ...] | ALL}] [WITH {CUBE | ROLLUP}]

SQL Server поддерживает стандарт ANSI с некоторыми отличиями. Первое заметное отличие заключается в синтаксисе: стандарт ANSI предписывает использовать синтаксис *GROUP BY [{CUBE | ROLLUP}] (столбец_группировки[, ...])*, а в SQL Server используется *GROUP BY [ALL] (столбец_группировки) [WITH {CUBE | ROLLUP}]*. Ключевое слово *ALL* нельзя использовать совместно *GROUPING SETS*, *CUBE* и *ROLLUP*. Для определения наборов группировок используйте *GROUPING SETS*. Обратите внимание, что наборы группировок не могут быть вложенными. (SQL Server поддерживает также фразу *GROUP BY ( )*, добавляющую в результирующий набор общий итог.)

GROUP BY ALL возвращает категории группировок даже в том случае, если соответствующая агрегация равна NULL (обычно SQL Server не возвращает категории с агрегациями, равными NULL). Использовать эту фразу следует только с *WHERE*. *WITH {CUBE | ROLLUP}* добавляет в результирующий набор агрегаты по категориям. Проще говоря, *ROLLUP* добавляет промежуточные итоги по категориям, а *CUBE* создает итоги по всем сочетаниям категорий.

Функция *GROUPING* помогает отличить обычные значения NULL от значений NULL, создаваемых операциями *ROLLUP* и *CUBE*.

В этом примере выполняется группировка по **royalty**, а агрегируется значение **advance**. Функция *GROUPING* применяется к полю **royalty**:

```
USE pubs
SELECT royalty, SUM(advance) 'total advance',
GROUPING(royalty) 'grp'
FROM titles
GROUP BY royalty WITH ROLLUP
```

Этот запрос возвращает две строки с NULL в столбце **royalty**. Первое значение NULL соответствует группе строк с NULL в этом столбце. Второе значение

NULL соответствует итогу, добавленному операцией *ROLLUP*. В итоговой строке выводится сумма столбца **advance** для всех значений **royalty**, и в столбце **grp** выводится значение 1.

Вот результат:

```
royalty   total advance  grp
--------- -------------- ---
NULL      NULL           0
10        57000.0000     0
12        2275.0000      0
14        4000.0000      0
16        7000.0000      0
24        25125.0000     0
NULL      95400.0000     1
```

ORDER BY

Работает в соответствии с требованиями стандарта ANSI. Однако следует помнить, что в SQL Server различные схемы упорядочения могут сильно влиять на получаемый результат. Например, при использовании определенных схем упорядочения значения «SMITH» и «smith» могут считаться различными. Нельзя использовать в *ORDER BY* столбцы типов *TEXT*, *NTEXT* и *IMAGE*.

COMPUTE {агрегация(выражение)}[, ...] [BY выражение[, ...]]

Создает дополнительные агрегация (обычно итоги), выводимые в конце результирующего набора. *BY* добавляет в данные промежуточные итоги. В одном запросе одновременно можно использовать *COMPUTE* и *COMPUTE BY*. *COMPUTE BY* следует использовать только в паре с *ORDER BY*, хотя выражение в *ORDER BY* может быть частью выражения сортировки. В качестве агрегации можно использовать любую из следующих функций: *AVG*, *COUNT*, *MAX*, *MIN*, *STDEV*, *STDEVP*, *VAR*, *VARP* и *SUM*. Примеры приводятся далее в этом разделе. *COMPUTE* ни в каком виде не работает с *DISTINCT* и столбцами типов *TEXT*, *NTEXT* и *IMAGE*.

FOR {BROWSE | XML {RAW | AUTO | EXPLICIT}[, XMLDATA][, ELEMENTS][,BINARY BASE64]}

FOR BROWSE используется при необходимости обновления данных, полученных через курсоры, работающие в режиме DB-Library. (DB-Library – это исходный метод доступа к SQL Server, вытесненный позднее почти из всех приложений интерфейсом OLEDB.) *FOR BROWSE* можно использовать только с таблицами, имеющими уникальный индекс и поле типа *TIMESTAMP*. *FOR BROWSE* нельзя использовать в операторах с *UNION* и с включенным параметром *HOLDLOCK*.

FOR XML

Используется клиентом SQL Server для получения результирующего множества в формате XML. Вы можете определить формат XML документа с помощью опций *RAW*, *AUTO* или *EXPLICIT*. *RAW* преобразует каждую строку результата в XML элемент общего вида с тэгом <row/>. *AUTO* преобразует результирующий набор в простое XML-дерево. Наконец, *EXPLICIT* преобразует результирующий набор в XML-дерево заданного формата. Однако при этом запрос должен быть написан так, что требуемая ин-

формация по вложенности указывается явно. Вы можете использовать несколько дополнительных параметров:

XMLDATA

Добавляет к XML документу схему. *ELEMENTS* возвращает столбцы как дочерние элементы, а не атрибуты XML.

BINARY BASE64

Возвращает двоичные данные в кодировке base-64. (Двоичный формат используется по умолчанию в режиме *AUTO*, но должен быть выбран явно в режимах *RAW* и *EXPLICIT*.) *FOR XML* не может использоваться ни в каких подзапросах, в сочетании с фразами *COMPUTE* и *BROWSE*, в определения представлений, в результирующих наборах пользовательских функций и в курсорах. Агрегации и *GROUP BY* исключают *FOR XML AUTO*.

OPTION (подсказка_оптимизатору [,...])

Заменяет шаги плана выполнения запроса на указанные. Так как оптимизатор обычно выбирает наилучший план выполнения запроса, то мы очень не рекомендуем прибегать к подсказкам оптимизатору. За дополнительной информацией по использованию подсказок оптимизатору обращайтесь к документации.

Вот пример использования в SQL Server функциональности *SELECT...INTO*. В этом примере с помощью *SELECT...INTO* создается новая таблица **non_mgr_employees**, содержащая идентификатор, фамилию, имя и описание должности (из таблицы **jobs**) для каждого сотрудника (из таблицы **employees**), не являющегося менеджером:

```
-- Запрос
SELECT e.emp_id, e.fname, e.lname,
SUBSTRING(j.job_desc,1,30) AS job_desc
INTO non_mgr_employee
FROM employee e
JOIN jobs AS j ON e.job_id = j.job_id
WHERE j.job_desc NOT LIKE '%MANAG%'
ORDER BY 2,3,1
```

Созданная таблица **non_mgr_emploees** может быть использована в запросах так же, как и любая другая таблица.

Так как оператор *SELECT...INTO* не журналируется и его результат не восстанавливается после сбоев, то использовать его можно только при разработке, но не в системах, находящихся в промышленной эксплуатации.

Многочисленные расширения по сравнению с ANSI оператора *SELECT* в SQL Server затрагивают и *GROUP BY*. Например, фраза *GROUP BY ALL* используется для включения в результат агрегаций, имеющих значение NULL. Следующие два запроса отличаются только ключевым словом *ALL*, но имеют весьма разный результат:

```
-- Обычная фраза GROUP BY
SELECT type, AVG(price) AS price
```

```
FROM titles
WHERE royalty <= 10
GROUP BY type
ORDER BY type
-- Результат
type         price
------------ -------
business     17.3100
mod_cook     19.9900
popular_comp 20.0000
psychology   13.5040
trad_cook    17.9700

-- Фраза GROUP BY ALL
SELECT type, AVG(price) AS price
FROM titles
WHERE royalty = 10
GROUP BY ALL type
ORDER BY type
-- Результат
type         price
------------ -------
business     17.3100
mod_cook     19.9900
popular_comp 20.0000
psychology   13.5040
trad_cook    17.9700
UNDECIDED    NULL
```

COMPUTE имеет несколько вариантов использования, сильно влияющих на результат запроса. В следующем примере суммируются цены на книги в разбивке по типу книги и с сортировкой по типу книги и по цене:

```
-- Запрос
SELECT type, price
FROM titles
WHERE type IN ('business','psychology')
  AND price > 10
ORDER BY type, price
COMPUTE SUM(price) BY type
-- Результат
type         price
------------ ---------------------
business     11.9500
business     19.9900
business     19.9900
             sum
             =====================
             51.9300
type         price
------------ ---------------------
psychology   10.9500
psychology   19.9900
psychology   21.5900
```

```
             sum
             ====================
             52.5300
```

COMPUTE без *BY* используется совершенно иным образом. В следующем примере подсчитывается общий итог по всем ценам и всем авансам для книг с ценой больше $16:

```
-- Запрос
SELECT type, price, advance
FROM titles
WHERE price > $16
COMPUTE SUM(price), SUM(advance)
-- Результат
Type         price                advance
------------ -------------------- ---------------------
business     19.9900              5000.0000
business     19.9900              5000.0000
mod_cook     19.9900              .0000
popular_comp 22.9500              7000.0000
popular_comp 20.0000              8000.0000
psychology   21.5900              7000.0000
psychology   19.9900              2000.0000
trad_cook    20.9500              7000.0000
             sum
             ====================
             165.4500
                                  sum
                                  ====================
                                  41000.0000
```

Вы можете использовать в одном запросе и *COMPUTE*, и *COMPUTE BY*. (Для краткости мы не приводим результат, а только запрос.) В следующем примере суммируются цены и авансы для книг по бизнесу и психологии с ценами больше $16:

```
SELECT type, price, advance
FROM titles
WHERE price > $16
  AND type IN ('business','psychology')
ORDER BY type, price
COMPUTE SUM(price), SUM(advance) BY type
COMPUTE SUM(price), SUM(advance)
```

Помните, что при использовании *COMPUTE BY* необходимо также указывать *ORDER BY*. (*ORDER BY* не требуется при обычном *COMPUTE*, без *BY*.) Существует множество вариантов использования этих фраз в одном запросе: несколько *COMPUTE* и *COMPUTE BY*, *GROUP BY* с *COMPUTE BY* и даже *COMPUTE* с *ORDER BY*. Иногда бывает действительно интересно поэкспериментировать с разными вариантами запросов. Конечно, это не так интересно, как в парке развлечений, но что вы хотели? Все-таки это книга по программированию!

SQL Server поддерживает фразу *FOR XML*, конвертирующую стандартное результирующее множество запроса в XML. Эта функциональность очень полезна для веб-приложений, работающих с базами данных. Вы можете использовать

FOR XML прямо в запросе к таблице либо в хранимой процедуре. Например, мы можем преобразовать результат одного из предыдущих запросов в XML:

```
SELECT type, price, advance
FROM titles
WHERE price > $16
  AND type IN ('business','psychology')
ORDER BY type, price
FOR XML AUTO
```

Результат не очень красив, но очень полезен:

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
----------------------------------------------------------
<titles type="business" price="19.9900" advance="5000.0000"/>
<titles type="business" price="19.9900"
advance="5000.0000"/>
<titles type="psychology" price="19.9900" advance="2000.0000"/>
<titles type="psychology" price="21.5900"
advance="7000.000
```

Если вам требуется схема XML либо требуется использовать тэги для каждого элемента, то вы можете добавить в *FOR XML* ключевые слова *XMLDATA* и *ELEMENTS*. Запрос будет выглядеть следующим образом:

```
SELECT type, price, advance
FROM titles
WHERE price > $16
  AND type IN ('business','psychology')
ORDER BY type, price
FOR XML AUTO, XMLDATA, ELEMENTS
```

В SQL Server есть еще несколько улучшений для работы с XML. Например, функция *OPENXML* может быть использована для вставки XML документа в таблицу SQL Server. Также поддерживаются хранимые процедуры, используемые для преобразования и манипуляции XML-документами.

SQL Server (как и Oracle, но немного в другом формате) поддерживает так называемые запросы с поворотом (pivot query), не описанные ни в ANSI, ни в ISO. Запрос с поворотом позволяет «повернуть данные на бок», что дает возможность извлечь из этих данных новую полезную информацию. Предположим, у нас есть запрос, возвращающий результат из двух столбцов и четырех строк:

```
SELECT days_to_make, AVG(manufacturing_cost) AS Avg_Cost
FROM manufacturing.products
GROUP BY days_to_make;
```

Результат запроса:

```
days_to_make Avg_Cost
0            5
1            225
2            350
4            950
```

Запрос с поворотом вращает данные таким образом, что значения столбца **days_to_make** становятся названиями столбцов и запрос возвращает одну строку:

```
SELECT 'Avg_Cost' As Cost_by_Days, [0], [1], [2], [3], [4]
FROM (SELECT days_to_make, manufacturing_cost
      FROM manufacturing.products)
AS source
PIVOT
  (AVG(manufacturing_cost)
   FOR days_to_make IN ([0], [1], [2], [3], [4]) )
AS pivottable;
```

Результат запроса:

```
Cost_by_Days     0    1    2    3    4
Avg_Cost         5    225  350  NULL 950
```

См. также

JOIN

WHERE

SET

Оператор *SET* используется для присвоения значений пользовательским и системным переменным.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Не поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
SET переменная = значение
```

Ключевые слова

переменная

Указывает системную или пользовательскую переменную.

значение

Указывает текстовое или числовое значение, устанавливаемое для пользовательской или системной переменной.

Общие правила

Значения переменных устанавливаются на время сессии. Тип присваиваемого значения должен совпадать с типом переменной. Например, вы не можете присвоить текстовое значение числовой переменной. Команды, используемые для создания переменных, зависят от платформы: в Oracle и SQL Server переменные объявляются оператором *DECLARE*, но в других платформах могут использоваться другие способы.

Присваиваемое переменной значение не обязательно должно быть литералом. Это может быть и динамически получаемое из подзапроса значение. В следующем примере мы присваиваем переменной **emp_id_var** максимальное значение идентификатора сотрудника:

```
DECLARE emp_id_var CHAR(5)
SET emp_id_var = (SELECT MAX(emp_id)
  FROM employees WHERE type = 'F')
```

В этом примере значение 'F' поля type используется для отбора только работающих на полную ставку сотрудников.

Советы и хитрости

Оператор *SET* хорошо переносится между платформами. Только Oracle использует другой способ присвоения значений переменным. В следующем примере для SQL Server мы объявляем переменную **emp_id_var** и присваиваем ей значение:

```
DECLARE emp_id_var CHAR(5)
SET emp_id_var = '67888'
```

То же действие на Oracle выполняется следующим образом:

```
DECLARE emp_id_var CHAR(5);
emp_id_var := '67888';
```

Дополнительная информация об этих отличиях приводится в следующих разделах.

MySQL

Ключевое слово *SET* имеет несколько значений. Во-первых, *SET* – это тип данных, хранящий несколько значений, разделенных запятыми. Во-вторых, *SET* используется для присваивания значения пользовательской переменной. В этой главе описывается второй вариант, его синтаксис следующий:

```
SET @переменная = значение[, ...]
```

Одним оператором вы можете установить значения нескольких переменных, перечислив их через запятую:

```
SET @new_var.order_qty = 125, @new_var.discount = 4;
```

MySQL позволяет использовать для присвоения значений переменным оператор *SELECT* таким же образом, как это описано в стандарте ANSI для оператора *SET*. Однако этот способ имеет определенный недостаток. Проблема заключается в том, что значения внутри оператора *SELECT* не присваиваются немедленно. Рассмотрим следующий пример:

```
SELECT (@new_var := row_id) AS a,
  (@new_var + 3) AS b
FROM table_name;
```

Переменная *@new_var* не будет иметь значение *row_id+3*, а сохранит значение, установленное в первой части оператора. По этой причине рекомендуется устанавливать значения переменных по одной.

Oracle

Оператор *SET* как способ присвоения значений переменных в Oracle не поддерживается. Для присвоения используется оператор *:=*, имеющий следующий синтаксис:

```
переменная := значение
```

PostgreSQL

В PostgreSQL оператор *SET* используется для присвоения переменных среды выполнения:

```
SET [SESSION | LOCAL] переменная { TO | = }
  { значение | DEFAULT }
```

Для переменной можно указать определенное значение, задаваемое литералом, либо установить значение по умолчанию с помощью ключевого слова *DEFAULT*. Ключевое слово *SESSION* указывает, что команда влияет только на текущую сессию (это поведению по умолчанию, если не указано ни *SESSION*, ни *LOCAL*). Ключевое слово *LOCAL* указывает, что команда влияет только на текущую транзакцию. После окончания транзакции устанавливается значение по умолчанию, определенное на уровне сессии.

Некоторые клиентские настройки также могут быть сконфигурированы при помощи оператора *SET* (хотя для этих же целей можно использовать *SET_CONFIG*). PostgreSQL поддерживает следующие дополнительные настройки:

[CLIENT_ENCODING] NAMES

Определяет используемую клиентом многобайтную кодировку для серверов PostgreSQL, поддерживающих многобайтные кодировки.

DATESTYLE

Устанавливает стиль отображения времени и даты. Поддерживаемые стили включают:

ISO

Отображает дату и время в формате *YYYY-MM-DD HH:MM:SS* (формат по умолчанию по стандарту ISO 8601).

SQL

Отображает дату и время в формате Oracle/Ingres, а не в формате, определяемом ANSI SQL.

Postgresql

Отображает дату и время в формате PostgreSQL (это больше не является значением по умолчанию).

German

Отображает дату и время в формате *DD.MM.YYYY*.

Вы можете детализировать стили SQL и PostgreSQL, используя ключевые слова European, US и NonEuropean, отображающие данные в формате *DD/MM/YYYY*, *MM/DD/YYYY* и *MM/DD/YYYY*, соответственно (например, *SET DATESTYLE = SQL, European;*).

SEED

Инициализирует встроенный генератор псевдослучайных чисел. Значение может быть любым числом от 0 до 1, умноженным на $2^{31}-1$. Для этих же целей можно воспользоваться функцией *setseed*, например:

```
SELECT setseed(значение);
```

SERVER_ENCODING

Активирует поддержку многобайтных кодировок на серверах, имеющих такую возможность.

TIME ZONE {часовой пояс | LOCAL | DEFAULT}

Позволяет выбрать указанный часовой пояс либо (если указано *LOCAL* или *DEFAULT*) использовать часовой пояс операционной системы. Полный список допустимых часовых поясов ищите в документации.

SQL Server

SQL Server поддерживает оператор *SET* для присвоения значений переменным, объявленным оператором *DECLARE*, а также курсорным переменным. (*SET* используется и для других целей, таких как включение и отключение настроек сессии, например *SET NOCOUNT ON*.) Синтаксис оператора следующий:

```
SET { { @переменная = значение }
  | { @курсорная_переменная =
        { @курсорная_переменная | имя_курсора
  | { CURSOR [ FORWARD_ONLY | SCROLL ]
  [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
  [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
  [ TYPE_WARNING ]
FOR оператор_select
  [ FOR { READ ONLY | UPDATE [ OF столбец[, ...] ] }
  ] } } } }
```

Оператор не поддерживает ключевое слово *DEFAULT*, но в остальном синтаксис ANSI поддерживается.

SET CONNECTION

Оператор *SET CONNECTION* позволяет пользователю переключаться между несколькими открытыми соединениями с одной или несколькими базам данных.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Не поддерживается
PostgreSQL	Не поддерживается
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

```
SET CONNECTION {DEFAULT | имя_соединения}
```

Ключевые слова

имя_соединения

Указывает соединение текущей сессии. Если указано соединение, отличное от используемого в текущий момент, то текущим соединением становится указанное в операторе.

DEFAULT

Переключается на соединение, используемое по умолчанию. Эта опция позволяет использовать соединение по умолчанию, даже если его имя неизвестно.

Описание

Этот оператор не закрывает соединение. Он переключается с текущего соединения на указанное соединение (делая его текущим) либо с текущего соединения на соединение по умолчанию. При переключении между соединениями предыдущее соединение переходит в спящее состояние (без фиксации или отката транзакции), а новое соединение становится активным.

Общие правила

Оператор *SET CONNECTION* не создает соединения, он просто переключает контекст соединения. Для создания новых соединений используйте оператор *CONNECT*, а для закрытия соединений – *DISCONNECT*.

Советы и хитрости

Оператор *SET CONNECTION* используется нечасто, так как пользователи обычно подключаются программно через ODBC, JDBC или иным способом. Однако на тех платформах, где *SET CONNECTION* поддерживается, он может быть очень удобен для частого изменения настроек соединения без закрытия текущих соединений.

MySQL

Не поддерживается.

Oracle

Не поддерживается.

PostgreSQL

Не поддерживается.

SQL Server

SQL Server поддерживает оператор *SET CONNECTION*, но только во встроенном SQL, но не в инструменте выполнения запросов SQL Query Analyzer. Хотя оператор и поддерживается в соответствии со стандартом SQL2003, в различных программах (например, на C++) он используется нечасто. Большинство предпочитает использовать оператор *USE*. Синтаксис *SET CONNECTION* следующий:

```
SET CONNECTION имя_соединения
```

Ключевое слово *DEFAULT* не поддерживается, в остальном оператор соответствует стандарту ANSI. Имя соединения должно соответствовать соединению, от-

крытому ранее с помощью оператора *CONNECT*, и может задаваться как литералом, так и через переменную.

Вот пример программы на T-SQL для SQL Server, демонстрирующей использование операторов *CONNECT*, *DISCONNECT* и *SET CONNECTION*:

```
EXEC SQL CONNECT TO chicago.pubs AS chicago1 USER sa;
EXEC SQL CONNECT TO new_york.pubs AS new_york1 USER read-only;
-- Открываются соединения с серверами "chicago" и "new_york"
EXEC SQL SET CONNECTION chicago1;
EXEC SQL SELECT name FROM employee INTO :name;
-- Соединение chicago1 делается активным и выполняется
-- работа с данными
EXEC SQL SET CONNECTION new_york1;
EXEC SQL SELECT name FROM employee INTO :name;
-- Соединение new_york1 делается активным и выполняется
-- работа с данными
EXEC SQL DISCONNECT ALL;
-- Закрываются все соединения. Вы могли бы вместо этого использовать
-- два оператора DISCONNECT: по одному для каждого соединения.
```

См. также

CONNECT
DISCONNECT

SET CONSTRAINT

Оператор *SET CONSTRAINT* устанавливает в текущей транзакции момент проверки ограничений целостности с откладываемой проверкой: ограничения целостности проверяются либо после каждого оператора DML либо при фиксации транзакции. Если в текущей сессии транзакция не начата, то оператор действует на следующую транзакцию.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Не поддерживается

Синтаксис SQL2003

```
SET CONSTRAINT {имя_ограничения[, ...] | ALL}
  {DEFERRED | IMMEDIATE}
```

Ключевые слова

имя_ограничения [, ...] | ALL

Указывает одно или несколько ограничений, для которых устанавливается режим проверки. Ключевое слово *ALL* устанавливает режим проверки для всех ограничений с возможностью отложенной проверки.

DEFERRED

Проверяет выполнение ограничений целостности в момент фиксации транзакции, а не после каждого оператора DML.

IMMEDIATE

Проверяет выполнение ограничений целостности после каждого оператора DML, а не при фиксации транзакции.

Общие правила

SET CONSTRAINT определяет режим работы откладываемых ограничений целостности в текущей транзакции. Если в текущей сессии транзакция не начата, то оператор действует на следующую транзакцию.

В следующем примере для всех ограничений целостности устанавливается режим проверки после каждого оператора DML:

```
SET CONSTRAINT ALL IMMEDIATE;
```

В следующем примере для двух ограничений устанавливается отложенный режим проверки в конце транзакции:

```
SET CONSTRAINT scott.hr_job_title, scott.emp_bonus DEFERRED;
```

Советы и хитрости

В момент создания ограничения вы определяете, допускает ли ограничение отложенную проверку (*DEFERRABLE*) или не допускает (*NOT DEFERRABLE*). Оператор *SET CONSTRAINT*, выполненный для ограничения типа *NOT DEFERRABLE*, завершится с ошибкой.

MySQL

Не поддерживается.

Oracle

Oracle полностью поддерживает оператор, как того требует стандарт ANSI, а также позволяет использовать *SET CONSTRAINTS* как синоним *SET CONSTRAINT*.

PostgreSQL

PostgreSQL поддерживает оператор в соответствии со стандартом ANSI. Однако в PostgreSQL можно откладывать проверку только ограничений ссылочной целостности, а ограничения *CHECK* и *UNIQUE* всегда имеют тип *IMMEDIATE*.

SQL Server

Не поддерживается.

SET PATH

Оператор *SET PATH* меняет значение настройки *CURRENT PATH* на одну или несколько схем.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Не поддерживается
PostgreSQL	Не поддерживается
SQL Server	Не поддерживается

Синтаксис SQL2003

```
SET PATH имя_схемы[, ...]
```

Ключевые слова

имя_схемы [, ...]

Определяет одну или несколько схем в качестве текущего пути.

Общие правила

SET PATH определяет одну или несколько схем, используемых для неуточненных имен функций, процедур и методов.

В следующем примере в качестве текущего пути (т. е. текущей схемы) используется **scott**:

```
SET PATH scott;
```

Если в текущей схеме будет вызываться процедура без указания схемы, то будет подразумеваться, что процедура находится в схеме **scott**.

Советы и хитрости

При указании нескольких схем все они должны находиться в текущей базе данных. (Нельзя использовать схемы удаленного сервера.)

SET PATH не задает схему для неуточненных имен таблиц и представлений, а только для подпрограмм (процедур, функций и методов).

SET PATH не поддерживается ни одной платформой из числа рассматриваемых в этой книге.

См. также

SET SCHEMA

SET ROLE

Оператор *SET ROLE* включает и выключает определенные роли в текущей сессии.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Не поддерживается

Синтаксис SQL2003

```
SET ROLE {NONE | имя_роли}
```

Ключевые слова

NONE

Присваивает текущей сессии роль *CURRENT ROLE*.

имя_роли

Связывает с текущей сессией набор привилегий указанной роли.

Общие правила

Если пользователь с помощью оператора *CONNECT* открыл сессию, то *SET ROLE* позволяет связать с текущей сессией набор привилегий определенной роли. Оператор *SET ROLE* можно выполнять только вне транзакций.

Имя роли, указываемое в *SET ROLE*, должно соответствовать существующей в базе данных роли. Задавать имя роли можно как литералом, так и через переменную.

Советы и хитрости

Сессии создаются оператором *CONNECT*, а роли – оператором *CREATE ROLE*.

Большинство платформ предлагает те или иные способы установки или изменения роли, используемой пользовательской сессией. Стандарт ANSI предписывает использовать для этих целей оператор *SET ROLE*, но он не очень широко поддерживается различными платформами. В следующих примерах рассматриваются аналогичные команды для каждой платформы, дополнительную информацию ищите в документации.

MySQL

Оператор *SET ROLE* не поддерживается. Управлять настройками подключения в MySQL можно с помощью секции [client] конфигурационного файла *.my.cnf*, находящегося в домашнем каталоге. Например:

```
[client]
host=server_name
user=user_name
password=client_password
```

Для быстрого переключения ролей вы можете изменять свойства *host*, *user* и *password*, присваивая новые значения переменным *MYSQL_HOST*, *USER* (только под Windows) и *MYSQL_PWD* (хотя использование *MYSQL_PWD* небезопасно, так как значение может быть просмотрено другими пользователями).

Oracle

Когда пользователь создает сессию, Oracle явно присваивает этому пользователю роли. Роли, из-под которых работает сессия, можно поменять оператором *SET ROLE*, при условии, что пользователь является членом включаемой роли. В файле параметров *INIT.ORA* имеется настройка *MAX_ENABLED_ROLES*, ограничивающая максимальное число ролей, которые могут быть включены одновременно. Синтаксис оператора *SET ROLE* в Oracle следующий:

```
SET ROLE { имя_роли [IDENTIFIED BY пароль][, ...]
| [ALL [EXCEPT имя_роли [, ...]]
| NONE };
```

Параметры оператора *SET ROLE* следующие:

имя_роли

Указывает одну или несколько ролей, членом которых является пользователь. Любые роли, не указанные в этом операторе, будут недоступны пользователю в текущей сессии. Вы можете активировать несколько ролей, перечислив их имена через запятую.

IDENTIFIED BY пароль

Эту фразу необходимо использовать при включении ролей, защищенных паролем.

ALL

Включает все роли, членом которых (напрямую или через другие роли) является пользователь. Эту фразу нельзя использовать одновременно с *IDENTIFIED BY*.

EXCEPT

Определяет список ролей, исключаемых из команды *SET ROLE ALL*.

NONE

Отключает все роли, в том числе и роли, используемые по умолчанию.

Роли, защищенные паролями, можно активировать только при помощи оператора *SET роль IDENTIFIED BY пароль*. Например, следующий оператор включает роли *read_only* и *updater*, защищенные, соответственно, паролями *editor* и *red_marker*:

```
SET ROLE
read_only IDENTIFIED BY editor,
updater IDENTIFIED BY red_marker;
```

Для включения всех ролей, кроме *read_write*, мы можем выполнить следующий оператор:

```
SET ROLE ALL EXCEPT read_write;
```

PostgreSQL

PostgreSQL поддерживает синтаксис ANSI с дополнениями.

```
SET [ SESSION | LOCAL ] ROLE rolename| NONE
```

Эта команда устанавливает идентификатор текущей сессии в значение *rolename*. Все последующие SQL команды будут выполняться от имени указанной.

SESSION | LOCAL

Фраза SESSION позволяет распространить настройки к текущей сессии (работает по умолчанию). Фраза LOCAL применяет изменения, производимые командой, только для текущей транзакции. После окончания текущей транзакции возвращаются настройки, действующие для сессии. Фразу SET LOCAL

нет смысла применять вовне блока BEGIN, т. к. транзакция сразу же завершается.

RESET ROLE

восстанавливает набор ролей пользователя в значения по умолчанию

SQL Server

Не поддерживается.

См. также

CONNECT
CREATE ROLE
DISCONNECT
SET SESSION AUTHORIZATION

SET SCHEMA

Оператор *SET SCHEMA* меняет значения параметра *CURRENT SCHEMA* на схему, указанную пользователем.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Не поддерживается
PostgreSQL	Не поддерживается
SQL Server	Не поддерживается

Синтаксис SQL2003

```
SET SCHEMA имя_схемы[, ...]
```

Ключевые слова

имя_схемы [, ...]

Указывает одну или несколько схем, используемых в качестве текущих.

Общие правила

Оператор *SET SCHEMA* указывает схему, используемую по умолчанию для неуточненных имен объектов, таких как таблицы и представления.

В следующем примере в качестве текущей схемы выбирается схема **scott**:

```
SET SCHEMA scott;
```

Теперь, если в текущей сессии используется объект с неуточненной (неуказанной) схемой, то будет использоваться схема **scott**.

Советы и хитрости

Оператор *SET SCHEMA* не позволяет в качестве текущей схемы установить схему из удаленного сервера.

SET SCHEMA не задает схему для неуточненных имен подпрограмм, таких как процедуры, методы и функции, а только для таких объектов, как таблицы и представления.

Этот оператор не поддерживается ни в одной платформе из числа рассматриваемых в этой книге.

См. также

SET PATH

SET SESSION AUTHORIZATION

Оператор *SET SESSION AUTHORIZATION* определяет идентификатор пользователя в текущей сессии.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Не поддерживается
PostgreSQL	Поддерживается
SQL Server	Не поддерживается

Синтаксис SQL2003

```
SET SESSION AUTHORIZATION имя_пользователя
```

Ключевые слова

имя_пользователя

Делает указанного пользователя текущим пользователем сессии. Имя пользователя можно задавать литералом, через параметр или переменную.

Общие правила

Этот оператор позволяет переключаться между пользователями и работать с их привилегиями.

Советы и хитрости

В некоторых платформах вы можете использовать специальные ключевые слова, такие как *SESSION USER* или *CURRENT USER*. Обычно они делают одно и то же: возвращают имя пользователя текущей активной сессии. Однако иногда они могут возвращать и различные значения, например при использовании функции *SETUID* или других похожих механизмов.

Для выполнения *SET SESSION AUTHORIZATON* требуются права суперпользователя, но обратное переключение на исходного пользователя вы сможете сделать, даже если текущий пользователь не имеет привилегий на выполнение *SET SESSION AUTHORIZATION*.

С помощью следующего запроса вы могли бы получить результат функций *SESSION_USER* и *CURRENT_USER*:

```
SELECT SESSION_USER, CURRENT_USER;
```

Выполнять *SET SESSION AUTHORIZATION* следует до начала транзакции, чтобы все действия следующей транзакции выполнялись от имени нового пользователя. Этот оператор должен быть единственным в текущей транзакции.

MySQL

Не поддерживается. Для использования привилегий другого пользователя необходимо отключиться от MySQL, а затем подключиться заново.

Oracle

Не поддерживается. Для достижения аналогичного эффекта необходимо отключиться от сервера, а затем заново подключиться с помощью оператора *CONNECT*.

PostgreSQL

PostgreSQL поддерживает оператор в соответствии со стандартом ANSI. Единственным небольшим отличием является то, что по стандарту этот оператор нельзя выполнять в рамках транзакции, а в PostgreSQL это не имеет значения.

SQL Server

Не поддерживается. Для достижения аналогичного эффекта необходимо отключиться от сервера, а затем заново подключиться с помощью оператора *CONNECT*.

См. также

CONNECT
GRANT
SESSION_USER

SET TIME ZONE

С помощью оператора *SET TIME ZONE* можно поменять часовой пояс текущей сессии, если требуется, чтобы он отличался от используемого по умолчанию.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с ограничениями
SQL Server	Не поддерживается

Синтаксис SQL2003

```
SET TIME ZONE {LOCAL | INTERVAL {+ | -}'00:00'
  [HOUR TO MINUTE]}
```

Ключевые слова

LOCAL

Устанавливает для текущей сессии часовой пояс, используемый сервером.[1]

INTERVAL

Указывает смещение часового пояса от универсального координированного времени (Coordinated Universal Time, UTC) в терминах часов и минут. Смещение может быть как в сторону увеличения (+), так и уменьшения (–).

HOUR TO MINUTE

Указывает тип данных значения *TIME ZONE*.

Общие правила

Этот относительно простой оператор устанавливает в сессии пользователя либо часовой пояс сервера (*LOCAL*), либо часовой пояс с заданным смещением относительно универсального координированного времени (известного также как среднее время по Гринвичу, GMT). Например, *INTERVAL 2* соответствует часовому поясу с временем на 2 часа больше, чем UTC, а *INTERVAL -6* соответствует часовому поясу с временем на 6 часов меньше, чем UTC.

Советы и хитрости

Как и большинство операторов *SET*, этот оператор может быть выполнен только вне рамок транзакции. Другими словами, не следует окружать *SET TIME ZONE* операторами *START/BEGIN TRAN* и *COMMIT*.

MySQL

Не поддерживается.

Oracle

Начиная с Oracle 9*i*, вы можете использовать для выбора часового пояса оператор *ALTER SESSION* следующего вида:

```
ALTER SESSION
  SET TIME_ZONE = {'[+ | -] чч:мм'
    | LOCAL
    | DBTIMEZONE
    | 'регион'}
```

Для установки в сессии часового пояса по умолчанию используйте *LOCAL*, а для использования часового пояса базы данных используйте *DBTIMEZONE*. Вы можете указывать название региона часового пояса, например ‘EST’ или ‘PST’, либо указать смещение часового пояса в часах и минутах относительно UTC. Например, значение ‘-5:00’ означает, что время отстает от UTC на 5 часов (т. е., если время UTC 10:00, то в указанном часовом поясе оно составляет 5:00).

---

[1] В оригинале стандарта SQL2003 параметр LOCAL сбрасывает настройки временной зоны текущего сеанса в значения по умолчанию. См. описание LOCAL для Oracle – там это описано правильно. – *Прим. науч. ред.*

Для получения списка названий часовых поясов используйте следующий запрос:

```
SELECT tzname FROM v$timezone_names;
```

Оба следующих оператора устанавливают часовой пояс восточного стандартного времени (Eastern Standard Time). В первом случае это делается указанием смещения относительно UTC, во втором случае – указанием имени часового пояса:

```
ALTER SESSION SET TIME_ZONE = '-5:00';
ALTER SESSION SET TIME_ZONE = 'EST';
```

Поддержка часовых поясов в Oracle достаточно сложна. Книга Стивена Фейерштейна (Steven Feuerstein) Oracle PL/SQL Programming содержит хорошее описание поддержки часовых поясов в главе, посвященной типам данных даты и времени.

PostgreSQL

PostgreSQL позволяет установить для сессии часовой пояс, используемый сервером по умолчанию, с помощью *LOCAL* или *DEFAULT*:

```
SET TIME ZONE {'часовой_пояс' | LOCAL | DEFAULT };
```

Существуют некоторые отклонения от стандарта ANSI:

'часовой_пояс'

Указывает название часового пояса. Набор допустимых значений зависит от операционной системы. Например, каталог /usr/share/zoneinfo содержит список часовых поясов на Linux-серверах.

LOCAL | DEFAULT

Устанавливает часовой пояс, используемый по умолчанию в операционной системе сервера.[1]

Например, 'PST8PDT' соответствует часовому поясу Калифорнии на Linux-системах, а 'Europe/Rome' соответствует часовому поясу Италии как на Linux, так и на других системах. Если вы указываете некорректный часовой пояс, то используемым часовым поясом становится UTC.

В следующем примере в качестве часового пояса выбирается стандартное тихоокеанское время:

```
SET TIME ZONE 'PST8PDT';
```

А теперь возвращается часовой пояс, используемый по умолчанию:

```
SET TIME ZONE LOCAL;
```

SQL Server

Не поддерживается.

---

[1] *http://www.postgresql.org/docs/8.4/interactive/sql-set.html – Прим. науч. ред.*

SET TRANSACTION

Оператор *SET TRANSACTION* определяет для транзакции такие параметры, как возможность модификации данных, уровень изоляции и некоторые другие.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с ограничениями
PostgreSQL	Поддерживается
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
SET [LOCAL] TRANSACTION [READ ONLY | READ WRITE]
  [ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
    REPEATABLE READ | SERIALIZABLE}]
  [DIAGNOSTIC SIZE целое_число]
```

Ключевые слова

LOCAL

Меняет настройки транзакции текущей сессии только на локальном сервере. Если ключевое слово *LOCAL* не указано, то меняются настройки для следующей транзакции, даже если она выполняется на удаленном сервере.

READ ONLY

Устанавливает для следующей транзакции режим «только чтение». После окончания транзакции восстанавливается настройка по умолчанию.

READ WRITE

Позволяет следующей транзакции и чтение и запись данных.

ISOLATION LEVEL

Устанавливает уровень изоляции следующей транзакции в сессии.

READ COMMITTED

Позволяет транзакции читать строки, записанные другими транзакциями, только после их фиксации.

READ UNCOMMITTED

Позволяет транзакции читать строки, записанные другими транзакциями, но еще не зафиксированные.

REPEATABLE READ

Всем сессиям доступны данные, зафиксированные до начала первых транзакций в сессиях. Другие открытые сессии могут читать и изменять только данные, зафиксированные в текущих сессиях. Следовательно, поздние транзакции могут добавить строки, которые видны транзакциям в более ранних сессиях, но для получения этих записей необходимо повторно выполнить запрос.

SERIALIZABLE

Все сессии могут видеть записи, зафиксированные до момента начала первых транзакций этих сессий. До этого момента сессии могут видеть записи других сессий, но не могут вставлять или обновлять данные до окончания тех транзакций. Это наиболее строгий уровень изоляции и используется в SQL2003 по умолчанию.

DIAGNOSTIC SIZE *целое_число*

Указывает число сообщений об ошибках, фиксируемых при выполнении транзакции. Оператор *GET DIAGNOSTICS* позволяет получить эти сообщения.

Общие правила

Оператор *SET TRANSACTION* устанавливает параметры для следующей транзакции. Поэтому этот оператор должен быть выполнен после окончания одной транзакции, но до начала следующей. (Для начала новой транзакции с одновременным указанием ее характеристик используется оператор *START TRANSACTION*.) Одним оператором можно установить несколько параметров транзакции, но при этом должен быть указан только один метод доступа, один уровень изоляции и один параметр диагностики.

Уровень изоляции транзакции определяет степень изоляции транзакции от действий других параллельно выполняющихся транзакций. Уровень изоляции определяет следующее:

- Будут ли строки, прочитанные и обновленные в текущей сессии, доступны другим одновременно выполняющимся сессиям базы данных.
- Будут ли обновления и записи строк другими транзакциями влиять на текущую транзакцию.

Если вы не знакомы с уровнями изоляции транзакций, то внимательно прочитайте документацию по платформе.

Советы и хитрости

Фраза *ISOLATION LEVEL* определяет параметры поведения и допустимые аномалии, касающиеся одновременного выполнения транзакций. Существуют следующие аномалии:

Грязное чтение

Возникает, когда в транзакции читаются данные, измененные, но не зафиксированные другой транзакцией. В этом случае могут быть внесены изменения в записи, которые не были (а возможно и не будут) зафиксированы.

Неповторяющееся чтение

Возникает, когда в транзакции читается запись, модифицируемая из другой транзакции. Если первая транзакция попытается вновь прочитать запись, то запись в первоначальном виде не будет найдена.

Фантомные записи

Возникают в том случае, когда одна транзакция пытается прочитать строку, которая еще не существует в начале данной транзакции, но вставляется второй транзакцией, прежде чем закончится первая транзакция. Если первая

транзакция снова выполнит поиск этой строки, то обнаружит ее неожиданное появление. Эта новая строка называется *фантомной строкой, или записью*.

Таблица 3.6 показывает влияние уровня изоляции транзакции на возможность возникновения аномалий.

Таблица 3.6. Уровни изоляции и аномалии

Уровень изоляции	Грязное чтение	Неповторяющееся чтение	Фантомные записи
READ UNCOMMITED	Возможно	Возможно	Возможно
READ COMMITED	Невозможно	Возможно	Возможно
REPEATABLE READ	Невозможно	Невозможно	Возможно
SERIALIZABLE	Невозможно	Невозможно	Невозможно

MySQL

MySQL позволяет установить уровень изоляции следующей транзакции, всех транзакций в сессии либо глобально во всех сессиях сервера:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
[READ UNCOMMITTED |READ COMMITTED | REPEATABLE READ
  | SERIALIZABLE]
```

По умолчанию уровень изоляции определяется только для следующей транзакции. Ключевые слова следующие:

GLOBAL

Определяет уровень изоляции транзакций для всех последующих транзакций во всех пользовательских сессиях и системных потоках.

SESSION

Определяет уровень изоляции транзакций для всех последующих транзакций в текущей сессии.

TRANSACTION ISOLATION LEVEL

Устанавливает один из описанных в предыдущем разделе уровней изоляции транзакций. По умолчанию в MySQL используется уровень *REPEATABLE READ*.

Для установки уровня изоляции транзакций для всех сессий (GLOBAL) требуется привилегия SUPER. Также вы можете определить используемый по умолчанию уровень изоляции при запуске MySQL из командной строки ключом *–transaction-isolation=*. Следующий оператор устанавливает для всех транзакций всех сессий уровень изоляции *SERIALIZABLE*:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Oracle

Oracle позволяет выбирать для транзакции режим «только чтение» или «чтение-запись», устанавливать уровень изоляции, а также использовать определенный сегмент отката:

```
SET TRANSACTION { [ READ ONLY | READ WRITE ]
| [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]
| [ USE ROLLBACK SEGMENT имя_сегмента ]
| NAME 'имя_транзакции' };
```

где:

READ ONLY

Устанавливает для следующей транзакции режим «только чтение» и уровень изоляции *SERIALIZABLE*. Эта опция недоступна пользователю SYS. В транзакциях, работающих в режиме «только чтение», можно использовать только операторы *SELECT*, *ALTER SESSION*, *ALTER SYSTEM*, *LOCK TABLE* и *SET ROLE*.

READ WRITE

Разрешает следующей транзакции как читать, так и модифицировать данные (этот режим используется по умолчанию).

READ COMMITTED

Этот уровень изоляции используется в Oracle по умолчанию. Соответствует стандарту ANSI.

SERIALIZABLE

Устанавливает уровень изоляции транзакций *SERIALIZABLE*, соответствующий стандарту ANSI. Требует установки инициализационного параметра *COMPATIBLE* в значение 7.3.0 или выше.

USE ROLLBACK SEGMENT имя_сегмента

Предписывает транзакции, модифицирующей данные, использовать определенный сегмент отката. Так как эта настройка действует только на текущую транзакцию, то *USE ROLLBACK SEGMENT* должен быть первым оператором в транзакции. Эта опция несовместима с опцией *READ ONLY*. Если сегмента отката с указанным именем не существует, то оператор завершится с ошибкой.

NAME

Присваивает текущей транзакции имя, состоящее не более чем из 255 символов. Это опция очень полезна при распределенных транзакциях, так как позволяет легко идентифицировать локальную транзакцию, являющуюся частью распределенной транзакции.

Опция *USE ROLLBACK SEGMENT* может быть очень полезна при настройке производительности, так как позволяет для продолжительных транзакций использовать большие сегменты отката, способные хранить большие объемы данных отката, а для коротких транзакций – маленькие сегменты отката, способные поместиться в кэше.

Оператор *SET TRANSACTION* должен быть первым оператором в любом пакете операторов SQL. Однако в Oracle этот оператор действует аналогично оператору *START TRANSACTION*[1], поэтому вы можете свободно использовать любой из них.

В следующем примере для еженедельного отчета выполняется запрос, защищенный от влияния других процессов, вставляющих и модифицирующих строки:

---

[1] Который в Oracle не поддерживается. – *Прим. науч. ред.*

```
SET TRANSACTION READ ONLY NAME 'chicago';
SELECT prod_id, ord_qty
FROM sales
WHERE stor_id = 5;
```

В следующем примере ночной процесс пакетной обработки выполняет транзакции, способные переполнить любой сегмент отката, кроме специально созданного для этого процесса:

```
SET TRANSACTION USE ROLLBACK SEGMENT huge_tran_01;
```

PostgreSQL

SET TRANSACTION в PostgreSQL влияет только на новую начинаемую транзакцию. Поэтому вам может потребоваться начинать каждую транзакцию этим оператором. Синтаксис следующий:

```
SET TRANSACTION ISOLATION LEVEL
  {READ COMMITTED | SERIALIZABLE};
```

где:

READ COMMITTED

Устанавливает определенный стандартом ANSI уровень изоляции *READ COMMITTED*. (Этот уровень изоляции используется по умолчанию.)

SERIALIZABLE

Устанавливает для транзакции уровень изоляции *SERIALIZABLE*.

По умолчанию в PostgreSQL используется уровень изоляции *READ COMMITTED*. Поменять используемый по умолчанию уровень изоляции можно одним из следующих операторов:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
  { READ COMMITTED | SERIALIZABLE }
SET default_transaction_isolation =
  { 'read committed' | 'serializable' }
```

Конечно, впоследствии вы опять можете изменить уровень изоляции отдельной транзакции с помощью *SET TRANSACTION*.

Например, вы можете установить для следующей транзакции режим изоляции *SERIALIZABLE*:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Также вы можете установить уровень изоляции *SERIALIZABLE* для всех транзакций текущей сессии:

```
SET SESSION CHARACTERISTICS AS TRANSACTION
  ISOLATION LEVEL SERIALIZABLE;
```

SQL Server

SET TRANSACTION в SQL Server устанавливает уровень изоляции транзакций для всей сессии. Все операторы, следующие за *SET TRANSACTION*, выполняются с указанным уровнем изоляции, если только явно не выбран другой уровень. Синтаксис оператора следующий:

```
SET TRANSACTION ISOLATION LEVEL
{ READ COMMITTED
| READ UNCOMMITTED
| REPEATABLE READ
| SERIALIZABLE}
```

где:

READ COMMITTED

Устанавливает уровень изоляции транзакций *READ COMMITTED*. Этот уровень изоляции используется по умолчанию.

READ UNCOMMITTED

Устанавливает уровень изоляции транзакций *READ UNCOMMITTED*. Имеет тот же эффект, что и использование подсказки оптимизатору *NOLOCK*.

REPEATABLE READ

Устанавливает уровень изоляции транзакций *REPEATABLE READ*.

SERIALIZABLE

Устанавливает уровень изоляции транзакций *SERIALIZABLE*. Имеет тот же эффект, что и использование подсказки оптимизатору *HOLDLOCK*.

Например, следующий оператор изменяет уровень изоляции всех операторов *SELECT* с *READ COMMITTED* на *REPEATABLE READ*:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
```

См. также

COMMIT
ROLLBACK

START TRANSACTION

Оператор *START TRANSACTION* позволяет выполнить все функции оператора *SET TRANSACTION* с одновременным началом новой транзакции.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Не поддерживается
PostgreSQL	Не поддерживается, используйте BEGIN TRANSACTION
SQL Server	Не поддерживается, используйте BEGIN TRANSACTION

Синтаксис SQL2003

```
START TRANSACTION [READ ONLY | READ WRITE]
  [ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
    REPEATABLE READ | SERIALIZABLE}]
  [DIAGNOSTIC SIZE целое_число]
```

Ключевые слова

READ ONLY

Устанавливает для следующей транзакции режим «только чтение». После окончания транзакции восстанавливается настройка по умолчанию.

READ WRITE

Позволяет следующей транзакции и чтение и запись данных.

ISOLATION LEVEL

Устанавливает уровень изоляции следующей транзакции в сессии.

READ COMMITTED

Позволяет транзакции читать строки, записанные другими транзакциями только после их фиксации.

READ UNCOMMITTED

Позволяет транзакции читать строки, записанные другими транзакциями, но еще не зафиксированные.

REPEATABLE READ

Всем сессиям доступны данные, зафиксированные до начала первых транзакций в сессиях. Другие открытые сессии могут читать и изменять только данные, зафиксированные в текущих сессиях. Следовательно, поздние транзакции могут добавить строки, которые видны транзакциям в более ранних сессиях, но для получения этих записей необходимо повторно выполнить запрос.

SERIALIZABLE

Все сессии могут видеть записи, зафиксированные до момента начала первых транзакций этих сессий. До этого момента сессии могут видеть записи других сессий, но не могут вставлять или обновлять данные до окончания тех транзакций. Это наиболее строгий уровень изоляции и используется в SQL2003 по умолчанию.

DIAGNOSTIC SIZE *целое_число*

Указывает число сообщений об ошибках, фиксируемых при выполнении транзакции. Оператор *GET DIAGNOSTICS* позволяет получить эти сообщения.

Общие правила

Согласно стандарту ANSI, единственным отличием *START* и *SET* является то, что *SET* выполняется вне текущей транзакции, а *START* начинает новую транзакцию. Таким образом, *SET TRANSACTION* влияет на настройки следующей транзакции, а *START TRANSACTION* – текущей.

Хотя только MySQL поддерживает *START TRANSACTION*, три платформы (MySQL, PostgreSQL, SQL Server) поддерживают аналогичный оператор *BEGIN [TRAN[SACTION]]* и синоним *BEGIN [WORK]*. *BEGIN TRANSACTION* явно начинает новую транзакцию, но не определяет ее уровень изоляции. Относительно *START TRANSACTION* нужно помнить лишь то, что этот оператор определяет режим доступа транзакции к данным, уровень изоляции и глубину диагностики только для текущей транзакции. Если начинается новая транзакция, то либо нужно заново устанавливать все эти свойства, либо использовать значения по умолчанию.

Большинство платформ позволяют неявно управлять транзакциями в режиме так называемой автоматической фиксации. В этом режиме база данных выполняет каждый оператор в отдельной транзакции, автоматически оборачивая его в операторы *BEGIN TRAN* и *COMMIT TRAN*.

Альтернативой автоматической фиксации транзакций является ручное управление транзакциями. В ручном режиме управления транзакциями вы начинаете новую транзакцию оператором *START TRANSACTION*. Новая транзакция также может начинаться при выполнении некоторых операторов, инициирующих транзакции, например *INSERT*, *UPDATE*, *DELETE* или *SELECT*. Транзакция не фиксируется и не откатывается, пока явно не будет выполнен оператор *COMMIT* или *ROLLBACK*.

Oracle не поддерживает явное объявление новых транзакций при помощи оператора *START TRANSACTION*, но он поддерживает фиксацию, откат и точки сохранения. Другие платформы, включая MySQL, PostgreSQL и SQL Server позволяют как явно начать новую транзакцию оператором *START TRANSACTION*, так и явно зафиксировать или откатить транзакцию, или объявить точку сохранения.

Советы и хитрости

Многие из рассматриваемых здесь платформ по умолчанию работают в режиме автоматической фиксации транзакций. Поэтому желательно использовать явные транзакции только в том случае, если все транзакции текущей сессии будут явными. Другими словами, не смешивайте в одной сессии явные и автоматические транзакции. Каждая явно начатая транзакция может быть зафиксирована только оператором *COMMIT*, а в случае возникновения ошибок и необходимости отката нужно использовать *ROLLBACK*.

Внимательно проверяйте, что каждому оператору *START* соответствует парный оператор *COMMIT* или *ROLLBACK*. Пока не будет выполнен один из указанных операторов, СУБД не сможет завершить транзакцию. Отсутствие своевременных фиксаций или откатов транзакций может привести к росту транзакции до огромного размера.

Рекомендуется выполнять *COMMIT* или *ROLLBACK* после одного или нескольких операторов, так как длительные транзакции блокируют ресурсы, не допуская их использования другими пользователями. Большие длительные транзакции могут переполнять сегменты отката и файлы журналов, особенно если эти файлы не очень велики.

MySQL

MySQL по умолчанию работает в режиме автоматической фиксации транзакций, что означает, что все успешные изменения данных записываются на диск сразу же после выполнения, а в случае возникновения ошибки – сразу же откатываются.

MySQL поддерживает оператор *START TRANSACTION* и его синоним *BEGIN*. Вы можете отложить автоматическую фиксацию для одного или нескольких операторов, используя следующий синтаксис:

```
START TRANSACTION [WITH CONSISTENT SNAPSHOT]
BEGIN [WORK]
```

где:

WITH CONSISTENT SNAPSHOT

Начинает согласованное чтение данных при использовании механизмов хранения, поддерживающих согласованные чтения (на текущий момент только InnoDB) при работе на уровнях изоляции *SERIALIZABLE* и *REPEATABLE READ*.

BEGIN [WORK]

Объявляет начало одной или нескольких транзакций. Слово *WORK* необязательно и не несет функциональной нагрузки.

Следующий оператор отключает режим автоматической фиксации транзакций для всех сессий и потоков:

```
SET AUTOCOMMIT=0
```

После отключения режима автоматической фиксации транзакции требуется фиксировать все сделанные изменения с помощью *COMMIT* и откатывать все изменения с помощью *ROLLBACK*. Отключение автоматической фиксации транзакций имеет смысл только для транзакционно-безопасных таблиц, таких как таблицы InnoDB и BDB. Отключение автоматической фиксации транзакций для транзакционно-небезопасных таблиц не имеет никакого эффекта – автоматическая фиксация все равно будет выполняться.

Более ранние версии MySQL использовали журнал обновлений. Однако журнал обновлений не поддерживает транзакции ANSI, кроме как для таблиц InnoDB и NDB Cluster.

При выполнении *COMMIT* изменения транзакции записываются одной операцией в двоичный журнал. Например:

```
BEGIN;
  SELECT @A := SUM(salary)
  FROM employee
  WHERE type=1;
  UPDATE payhistory SET summmary=@A WHERE type=1;
COMMIT;
```

При откате транзакции, использующей транзакционно-незащищенные таблицы, вы получите ошибку *ER_WARNING_NOT_COMPLETE_ROLLBACK*, а транзакционно-защищенные таблицы будут восстановлены как обычно.

Oracle

Не поддерживается. Транзакции в Oracle начинаются неявно. Обратитесь к описанию оператора *SET TRANSACTION* для Oracle за информацией об управлении отдельными транзакциями.

PostgreSQL

PostgreSQL не поддерживает *START TRANSACTION*. Вместо этого используется следующий синтаксис:

```
BEGIN [ WORK | TRANSACTION ]
```

где:

BEGIN

Объявляет начало одной или нескольких транзакций.

WORK

Необязательное слово, не имеющее никакого эффекта.

TRANSACTION

Необязательное слово, не имеющее никакого эффекта.

PostgreSQL по умолчанию работает в режиме автоматической фиксации транзакций, т. е. каждый оператор DML выполняется в отдельной транзакции. Если оператор выполняется без ошибок, то PostgreSQL автоматически выполняет *COMMIT*, а в случае возникновения ошибок – *ROLLBACK*. Оператор *BEGIN* позволяет явно выполнить *COMMIT* или *ROLLBACK* по окончании транзакции, состоящей из нескольких операторов.

Транзакции, управляемые вручную, работают значительно быстрее автоматических транзакций. Для максимальной изоляции транзакций следует использовать уровень *SERIALIZABLE*. PostgreSQL позволяет использовать различные операторы DML (*INSERT*, *UPDATE*, *DELETE*) внутри блока *BEGIN*...*COMMIT*. Однако при выполнении *COMMIT* либо фиксируются все изменения, либо не фиксируются никакие, в зависимости от успешности выполнения *COMMIT*.

BEGIN имеет отдельное значение на платформах, поддерживающих собственные процедурные языки (это Oracle и SQL Server). На этих платформах *BEGIN* (без слова *TRANSACTION*) обозначает начало нового блока. По этой причине рекомендуется использовать ключевое слово *TRANSACTION* для начала транзакций в PostgreSQL. В противном случае вы можете столкнуться с проблемами при попытке миграции кода на Oracle или SQL Server.

Вот пример использования оператора *BEGIN TRANSACTION* в PostgreSQL:

```
BEGIN TRANSACTION;
  INSERT INTO jobs(job_id, job_desc, min_lvl, max_lvl)
  VALUES(15, 'Chief Operating Officer', 185, 135)
COMMIT;
```

SQL Server

SQL Server поддерживает оператор *BEGIN TRANSACTION* вместо предписанного стандартом ANSI оператора *START TRANSACTION*. Также поддерживаются небольшие расширения, упрощающие резервное копирование и восстановление. Синтаксис оператора следующий:

```
BEGIN TRAN[SACTION] [дескриптор_транзакции
  [WITH MARK [ 'дескриптор_журнала' ]]]
```

где:

TRAN[SACTION]

Объявляет начало транзакции. Вместо *TRANSACTION* можно использовать сокращение *TRAN*.

дескриптор_транзакции

Литерал или переменная символьного типа (*CHAR*, *NCHAR*, *VARCHAR* или NVARCHAR) до 32 символов в длину, задающие имя, используемое для идентификации транзакции. При работе с вложенными транзакциями именует только самую внешнюю транзакцию.

WITH MARK дескриптор_журнала

Указывает текстовую метку, записываемую в журнал транзакций. Это позволяет восстанавливать базу данных, находящуюся в режиме полного восстановления, до определенного момента времени, зафиксированного этой меткой. *WITH MARK* необходимо использовать совместно с именованной транзакцией.

При использовании вложенных транзакций имя транзакции может быть указано только в самой внешней паре операторов *BEGIN*...*COMMIT* или *BEGIN*... *ROLLBACK*. В целом мы рекомендуем избегать вложенных транзакций.

Вот пример для SQL Server, в котором несколько операторов *INSERT* выполняются в рамках одной транзакции:

```
BEGIN TRANSACTION
  INSERT INTO sales VALUES('7896','JR3435','Oct 28 2003',25,
    'Net 60','BU7832')
  INSERT INTO sales VALUES('7901','JR3435','Oct 28 2003',17,
    'Net 30','BU7832')
  INSERT INTO sales VALUES('7907','JR3435','Oct 28 2003',6,
    'Net 15','BU7832')
  COMMIT
GO
```

Если по каким-то причинам задерживается выполнение одного из операторов *INSERT*, задерживаются все операторы, так как они выполняются в одной транзакции.

См. также

COMMIT
ROLLBACK

Subquery (подзапросы)

Подзапросы – это вложенные запросы. Подзапросы могут быть использованы в разных частях оператора SQL.

СУБД	Уровень поддержки
MySQL	Поддерживается с ограничениями
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

В SQL существуют следующие типы подзапросов:

Скалярные подзапросы

Подзапросы, возвращающие одиночное значение. Являются наиболее широко поддерживаемым типом подзапросов.

Табличные подзапросы

Подзапросы, возвращающие более одного значения или одной строки значений.[1]

Вложенные табличные подзапросы

Подзапросы, возвращающие более одного столбца и одной строки.[2]

Скалярные и векторные подзапросы на некоторых платформах используются как часть выражения в списке элементов *SELECT*, фразах *WHERE* и *HAVING*. Вложенные табличные подзапросы обычно используются во фразе *FROM* оператора *SELECT*.

Коррелированный подзапрос – это подзапрос, зависящий от значения внешнего запроса. Поэтому коррелированный подзапрос выполняется отдельно для каждой записи внешнего запроса. Если подзапрос вложен на несколько уровней, то он может использовать значения любого внешнего запроса.

В зависимости от места в операторе, в котором используется подзапрос, к подзапросу выдвигаются различные требования. Уровень поддержки подзапросов различными платформами также варьируется: на некоторых платформах подзапросы можно использовать в любых фразах операторов (*SELECT*, *FROM*, *WHERE* и *HAVING*), а на других платформах – только в одной-двух фразах.

Подзапросы обычно ассоциируются с оператором *SELECT*. Но так как подзапросы можно использовать во фразе *WHERE*, то их можно использовать и в любом операторе, поддерживающем *WHERE*, включая *SELECT*, *INSERT...SELECT*, *DELETE* и *UPDATE*.

Синтаксис SQL2003

Скалярные, табличные и вложенные табличные подзапросы представлены в следующем обобщенном виде:

```
SELECT столбец1, столбец2, ... (скалярный_подзапрос)
FROM таблица1, ... (вложенный_табличный_подзапрос)
  AS имя_табличного_подзапроса]
WHERE foo = (скалярный_подзапрос)
   OR foo IN (табличный_подзапрос)
```

Коррелированные подзапросы более сложны, так как они зависят от значений, получаемых внешним запросом. Например:

```
SELECT столбец1
FROM таблица1 AS t1
WHERE foo IN
```

[1] Фактически столбец значений. – *Прим. науч. ред.*

[2] Фактически таблица. – *Прим. науч. ред.*

```
(SELECT значение1
 FROM таблица2 AS t2
 WHERE t2.pk_identifier = t1.fk_identifier)
```

IN использован только для демонстрации. Вместо *IN* можно использовать любой оператор сравнения.

Ключевые слова

скалярный_подзапрос

Включает скалярный подзапрос в список элементов *SELECT* или во фразы *WHERE* и *HAVING*.

вложенный_табличный_подзапрос

Включает вложенный табличный подзапрос во фразу *FROM* в сочетании с *AS*.

табличный_подзапрос

Включает табличный подзапрос во фразу *WHERE* в сочетании с такими операторами, как *IN*, *ANY*, *SOME*, *EXISTS* или *ALL*, для работы с множествами значений. Табличные подзапросы возвращают одну или несколько строк, каждая из которых содержит одиночное значение.

Общие правила

Подзапросы позволяют получить одно или несколько значений и использовать их в операторах *SELECT*, *INSERT*, *UPDATE* или *DELETE* либо внутри другого подзапроса. Подзапросы можно использовать везде, где допустимы выражения. Часто подзапрос можно заменить эквивалентным соединением. В некоторых платформах подзапросы могут работать менее эффективно, чем соединения.

Подзапросы всегда заключаются в скобки.

Подзапросы могут быть использованы во фразе *SELECT*, содержащей как минимум один элемент, во фразе *FROM*, ссылающейся на одну или несколько таблиц или представлений, и во фразах *WHERE* и *HAVING*.

Скалярные подзапросы могут возвращать только одиночные значения. Определенные операторы в *WHERE*, такие как =, <, >, >=, <= и <> (или *!=*), также используются только с одиночными значениями. Если подзапрос возвращает несколько значений, а оператор может работать только с одним, то запрос завершится с ошибкой. С другой стороны, табличные подзапросы, возвращающие несколько значений, можно использовать совместно с выражениями *[NOT] IN*, *ANY*, *ALL*, *SOME* или *[NOT] EXISTS*.

Вложенные табличные подзапросы могут быть использованы только во фразе *FROM* и обязательно должны быть снабжены псевдонимом при помощи *AS*. Результирующее множество вложенного табличного подзапроса, иногда называемого «производным представлением», может быть использовано как обычное представление (смотрите описание оператора *CREATE VIEW*). Внешний запрос может использовать все (или не все) столбцы подзапроса.

Коррелированные подзапросы обычно встречаются во фразах *WHERE* или *HAVING* внешнего запроса (и реже в списке элементов *SELECT*) и связываются с внешним запросом через фразу *WHERE*. (Коррелированные подзапросы можно также использовать в качестве вложенных табличных подзапросов, хотя такая ситуация встречается редко.) В подзапросе необходимо использовать фразу *WHERE*, условие которой включает значения из внешнего подзапроса; это требование иллюстрируется предыдущим примером синтаксиса ANSI.

Также важно с помощью *AS* указать псевдонимы (называемые коррелированными именами) для всех таблиц, используемых как во вложенном коррелированном подзапросе, так и во внешнем запросе. Коррелированные имена позволяют избежать неоднозначности и помогают СУБД быстро разрешить ссылки на используемые таблицы.

Подзапросы, совместимые с ANSI, должны удовлетворять следующим требованиям:

- Подзапросы не могут включать фразу *ORDER BY*.
- Подзапросы не могут быть включены в агрегирующие функции. Например, следующий подзапрос некорректен: *SELECT foo FROM table1 WHERE sales >=AVG(SELECT column1 FROM sales_table...)*. Вы можете обойти это ограничение, выполнив агрегацию в запросе, а не в подзапросе.

Советы и хитрости

На большинстве платформ подзапросы не должны использовать большие объекты (например, *BLOB* и *CLOB* на Oracle и *IMAGE* и *TEXT* на SQL Server), а также типы данных-коллекции (такие как *TABLE* и *CURSOR* на SQL Server).

Все платформы поддерживают подзапросы, но не все платформы поддерживают все виды подзапросов. Таблица 3.7 показывает, какие подзапросы поддерживаются каждой платформой.

В настоящий момент все платформы поддерживают все виды подзапросов.

Таблица 3.7. Поддержка подзапросов платформами

Платформа	MySQL	Oracle	PostgreSQL	SQL Server
Скалярный подзапрос в списке элементов *SELECT*	+	+	–	+
Скалярный подзапрос во фразах *WHERE/HAVING*	+	+	+	+
Векторный подзапрос во фразах *WHERE/HAVING*	+	+	+	+
Вложенный табличный подзапрос во фразе *FROM*	+	+	+	+
Коррелированный подзапрос во фразах *WHERE/HAVING*	+	+	+	+

Область применения подзапросов не ограничена операторами *SELECT*. Они также могут быть использованы в операторах *INSERT*, *UPDATE* и *DELETE*, имеющих фразу *WHERE*. Подзапросы часто используются в следующих целях:

- Для определения строк, вставляемых в целевую таблицу при использовании *INSERT...SELECT*, *CREATE TABLE...SELECT* и *SELECT...INTO*.
- Для определения строк представления или материализованного представления в операторе *CREATE VIEW*.
- Для определения значений строк, обновляемых оператором *UPDATE*.
- Для определения значений, используемых в условиях во фразах *WHERE* и *HAVING* операторов *SELECT*, *UPDATE* и *DELETE*.
- Для построения представлений «на лету» (вложенные табличные подзапросы).

Примеры

В этом разделе приводятся примеры подзапросов, работающие одинаково корректно на MySQL, Oracle, PostgreSQL и SQL Server.

В первом примере показан простой скалярный подзапрос в списке элементов *SELECT*:

```
SELECT job, (SELECT AVG(salary) FROM employee) AS "Avg Sal"
FROM employee
```

Вложенные табличные подзапросы используются так же, как и представления. В следующем примере мы в подзапросе получаем зарплату и уровень образования, а затем во внешнем запросе агрегируем полученные значения:

```
SELECT AVG(edlevel), AVG(salary)
FROM (SELECT edlevel, salary
      FROM employee) AS emprand
GROUP BY edlevel
```

Помните, что на некоторых платформах этот запрос может завершиться с ошибкой, если в нем для подзапроса с помощью *AS* не будет задан псевдоним.

В следующем примере во фразе *WHERE* используется стандартный табличный подзапрос. Мы хотим получить номера всех проектов, выполняемых сотрудниками департамента 'A00':

```
SELECT projno
FROM emp_act
WHERE empno IN
  (SELECT empno
  FROM employee
  WHERE workdept = 'A00')
```

Этот подзапрос выполняется только один раз для всего запроса.

В следующем примере мы хотим для каждого сотрудника получить его имя и уровень старшинства с точки зрения времени работы в компании. Мы делаем это с помощью коррелированного подзапроса:

```
SELECT firstname, lastname,
  (SELECT COUNT(*)
  FROM employee, senior
  WHERE employee.hiredate > senior.hiredate) as senioritype
FROM employee
```

В отличие от предыдущего запроса, в этом случае подзапрос выполняется один раз для каждой строки внешнего запроса. Общее время работы такого запроса может быть очень велико, так как внутренний запрос может потребоваться выполнить много раз.

Коррелированные подзапросы зависят от значений, получаемых во внешнем запросе. Ими может быть непросто овладеть, но они предлагают уникальные возможности. В следующем примере возвращается информация о заказах, в которых количество проданного товара меньше, чем средняя продажа этого товара:

```
SELECT s1.ord_num, s1.title_id, s1.qty
FROM sales AS s1
WHERE s1.qty <
  (SELECT AVG(s2.qty)
  FROM sales AS s2
  WHERE s2.title_id = s1.title_id)
```

В этом примере вы могли бы решить задачу, соединив таблицу саму с собой. Однако бывают ситуации, когда коррелированный подзапрос является единственным простым способом решить задачу.

Следующий пример показывает, как коррелированный подзапрос может быть использован для обновления строк таблицы:

```
UPDATE course SET ends =
  (SELECT min(c.begins) FROM course AS c
  WHERE c.begins BETWEEN course.begins AND course.ends)
WHERE EXISTS
  (SELECT * FROM course AS c
  WHERE c.begins BETWEEN course.begins AND course.ends)
```

Аналогичным образом вы можете использовать подзапрос для определения множества удаляемых строк. В этом примере коррелированный подзапрос используется для удаления строк одной таблицы, имеющих соответствующие строки в другой таблице:

```
DELETE FROM course
WHERE EXISTS
   (SELECT * FROM course AS c
    WHERE course.id > c.id
      AND (course.begins BETWEEN c.begins
      AND c.ends OR course.ends BETWEEN c.begins
         AND c.ends))
```

MySQL

MySQL поддерживет вложенные табличные подзапросы в качестве элементов списка *SELECT* и во фразе *WHERE*.

Oracle

Oracle поддерживает подзапросы, описанные в стандарте ANSI, но называются они иначе. Вложенный табличный подзапрос, используемый во фразе *FROM*, называется *вложенным представлением* (*inline view*). Это выглядит разумно, так как вложенные табличные подзапросы фактически являются представлениями, создаваемыми на лету. Подзапросы, используемые во фразах *WHERE* и *HAVING*, называются *вложенными подзапросами*. Oracle также поддерживает коррелированные подзапросы в списке элементов *SELECT* и во фразах *WHERE* и *HAVING*.

PostgreSQL

PostgreSQL поддерживает подзапросы во фразах *FROM*, *WHERE* и *HAVING*. Однако подзапросы во фразе *HAVING* не могут содержать *ORDER BY*, *FOR UPDATE* и *LIMIT*. На текущий момент PostgreSQL также поддерживает подзапросы в списке элементов *SELECT*.

SQL Server

SQL Server поддерживает подзапросы, описанные в стандарте ANSI. Скалярные подзапросы могут использоваться практически везде, где допустимы обычные выражения. Подзапросы не могут содержать фразы *COMPUTE* и *FOR BROWSE* и могут содержать *ORDER BY*, если также используется *TOP*.

См. также

DELETE
INSERT
SELECT
UPDATE
WHERE

TRUNCATE TABLE

Оператор *TRUNCATE TABLE*, не описанный в стандарте ANSI, безвозвратно, без журналирования удаляет данные из таблицы. Этот оператор быстро удаляет все строки из таблицы, не изменяя структуру самой таблицы и не записывая никаких (или записывая небольшое количество) данных в журналы транзакций. Однако так как *TRUNCATE TABLE* не журналируется, то он не может быть откачен после выполнения.

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

Фактически стандартный синтаксис

Официально оператор *TRUNCATE TABLE* не входит в стандарт ANSI. Однако он широко поддерживается в следующем стандартном виде:

```
TRUNCATE TABLE имя_таблицы
```

Ключевые слова

имя_таблицы

Указывает любую таблицу в текущей базе данных или схеме.

Общие правила

Оператор *TRUNCATE TABLE* действует так же, как и оператор *DELETE* без фразы *WHERE*, – удаляет все строки таблицы. Однако есть существенный нюанс: оператор *TRUNCAET TABLE* не журналируется, поэтому работает быстрее, но не может быть откачен в случае ошибочного запуска, а кроме того, *TRNCATE TABLE* не вызывает срабатывания триггеров, которые запускаются оператором *DELETE*.

Этот оператор следует выполнять вручную. Мы очень не рекомендуем использовать этот оператор в автоматизированных скриптах или промышленно эксплуатирующихся системах, содержащих ценные данные. *TRUNCATE TABLE* не комбинируется с операторами управления транзакциями, такими как *BEGIN TRAN* или *COMMIT*.

Советы и хитрости

Так как оператор *TRUNCATE TABLE* не журналируется, то обычно он используется только при базах данных разработчиков. В промышленных базах данных используйте его очень осторожно.

Мы рекомендуем не использовать *TRUNCATE TABLE* в хранимых процедурах и функциях промышленных приложений, так как на большинстве платформ этот оператор не журналируется и не может быть откачен в случае ошибочного запуска.

Оператор *TRUNCATE TABLE* завершится с ошибкой, если в момент его запуска кто-либо удерживает блокировку на очищаемую таблицу. *TRUNCATE TABLE* не инициирует срабатывание триггеров, но работает корректно при их наличии. Однако оператор не работает при наличии активных внешних ключей.

MySQL

MySQL версий 3.23 и старше поддерживает базовый вариант оператора:

```
TRUNCATE TABLE имя_таблицы
```

Для выполнения *TRUNCATE TABLE* MySQL удаляет таблицу, а затем создает ее заново. Так как MySQL хранит каждую таблицу в файле *<имя_таблицы>.frm*, то этот файл должен присутствовать в каталоге базы данных для корректной работы оператора.

Например, для удаления всех строк из таблицы **publishers** можно выполнить:

```
TRUNCATE TABLE publishers
```

Oracle

Oracle позволяет очищать таблицы и индексные кластеры (но не хеш-кластеры). Используется следующий синтаксис:

```
TRUNCATE { CLUSTER [владелец.]кластер
| TABLE [владелец.]таблица [{PRESERVE | PURGE} MATERIALIZED VIEW LOG] }
[{DROP | REUSE} STORAGE]
```

Oracle был первой платформой, поддерживающей *TRUNCATE TABLE*, остальные платформы начали поддерживать этот оператор позднее. *TRUNCATE TABLE* в Oracle имеет больше возможностей, чем обычно реализуется в других платформах.

CLUSTER | TABLE

Указывает, очищается ли кластер или таблица. Индексный кластер – это физическая структура, в которой связанные записи таблицы хранятся на диске рядом, что уменьшает необходимое для соединений число операций ввода-вывода. Опции *MATERIALIZED VIEW LOG* и *STORAGE* при очистке кластеров не используются.

{PRESERVE | PURGE} MATERIALIZED VIEW LOG

Сохраняет (*PRESERVE*) или очищает (*PURGE*) журнал материализованных представлений при очистке таблицы.

{DROP | REUSE} STORAGE

Позволяет освободить пространство, ранее занимавшееся очищенной таблицей (*DROP*), либо оставить его выделенным таблице, хотя и пустым (*REUSE*).

Например:

```
TRUNCATE TABLE scott.authors
PRESERVE MATERIALIZED VIEW LOG REUSE STORAGE
```

Этот оператор удаляет все записи из таблицы **scott.authors**. При этом сохраняется журнал материализованных представлений, а занятое этой таблицей пространство не освобождается.

PostgreSQL

Реализация оператора в PostgreSQL отличается от общепринятого стандарта тем, что ключевое слово *TABLE* является необязательным:

```
TRUNCATE TABLE имя_таблицы
```

Следующий оператор удаляет все записи из таблицы **authors**:

```
TRUNCATE TABLE authors
```

SQL Server

SQL Server поддерживает оператор в общепринятом стандартном виде.

См. также

DELETE

UNION

Оператор *UNION* объединяет результаты двух запросов в одно множество.

UNION относится к операторам *работы с множествами*. Другие операторы работы с множествами включают *INTERSECT* и *EXCEPT/MINUS*. (*EXCEPT* и *MINUS* функционально эквивалентны, но *EXCEPT* является стандартом по ANSI.) Все эти операторы предназначены для одновременной работы с результирующими множествами нескольких запросов, отсюда и название – операторы работы с множествами.

СУБД	Уровень поддержки
MySQL	Не поддерживается
Oracle	Поддерживается с ограничениями
PostgreSQL	Поддерживается с ограничениями
SQL Server	Поддерживается с ограничениями

Синтаксис SQL2003

На число запросов, которые можно объединять с помощью *UNION*, никаких ограничений не накладывается. Общий синтаксис оператора следующий:

```
<оператор_select1>
UNION [ALL | DISTINCT]
<оператор_select2>
UNION [ALL | DISTINCT]
...
```

Ключевые слова

UNION

Объединяет результирующие наборы операторов в один набор. Повторяющиеся строки по умолчанию исключаются.

ALL | DISTINCT

Позволяет исключить повторяющиеся строки из конечного результата (*DISTINCT*) либо оставить повторяющиеся строки (*ALL*). Столбцы с пустыми значениями считаются одинаковыми. Если не указано ни *ALL*, ни *DISTINCT*, то по умолчанию дубликаты удаляются.

Общие правила

При использовании *UNION* следует помнить только об одном существенном правиле: порядок, число и типы данных во всех объединяемых запросах должны совпадать.

Типы данных не должны быть идентичными, но они должны быть совместимыми. Например, совместимыми являются типы данных *CHAR* и *VARCHAR*. Как правило, в результирующем множестве будет использован самый длинный тип данных. Например, если объединяются три запроса, в которых столбцы имеют

типы *CHAR(5)*, *CHAR(10)* и *CHAR(12)*, то конечный результат будет иметь тип *CHAR(12)*, а данные из более коротких столбцов будут дополнены пробелами.

Советы и хитрости

Хотя согласно стандарту ANSI оператор *INTERSECT* имеет наибольший приоритет среди всех операторов работы с множествами, большинство платформ выполняют все операторы работы с множествами с равным приоритетом. Вы можете явно определять порядок выполнения операторов с помощью скобок. В противном случае с большой вероятностью операторы будут выполняться в порядке следования, сверху вниз.

На некоторых платформах использование *DISTINCT* может приводить к большим затратам на выполнение запроса, так как при этом требуется дополнительный проход по данным для удаления дубликатов. Поэтому если вы уверены, что в данных дубликатов нет (либо их не требуется удалять), то используйте *ALL* для более быстрого выполнения запроса.

В соответствии со стандартом ANSI можно использовать только одну фразу *ORDER BY* для всего оператора. Используйте ее в конце последнего оператора *SELECT*. Для исключения неоднозначности в столбцах и таблицах вы можете снабдить все столбцы соответствующими одинаковыми псевдонимами. Однако для именования столбцов конечного результата используются только псевдонимы из первого запроса в *SELECT...UNION*. Например:

```
SELECT au_lname AS "lastname", au_fname AS "firstname"
FROM authors
UNION
SELECT emp_lname AS "lastname", emp_fname AS "firstname"
FROM employees
ORDER BY lastname, firstname
```

Хотя наборы столбцов запросов могут иметь совместимые типы данных, в некоторых платформах могут быть различия в обработке длин столбцов. Например, если столбец **au_lname** из первого запроса имеет значительно большую длину, чем столбец **emp_name** из второго запроса, то правила выбора длины столбца для конечного результата могут быть различными в разных платформах. Хотя обычно просто выбирается тип с наибольшей длиной (наименее ограниченный).

Каждая СУБД может иметь свои правила именования столбцов в случае, когда столбцы в запросах имеют разные названия. Обычно используются названия столбцов из первого запроса.

MySQL

Не поддерживается.

Oracle

Oracle поддерживает базовую функциональность операторов *UNION* и *UNION ALL* в соответствии со следующим синтаксисом:

```
<оператор_select1>
UNION [ALL]
<оператор_select2>
```

```
UNION [ALL]
...
```

Вместо *UNION DISTINCT* следует использовать функционально эквивалентный *UNION*. Нельзя использовать оператор *UNION* в следующих случаях:

- Если запросы содержат столбцы типов *LONG*, *BLOB*, *CLOB*, *BFILE* и *VARRAY*.
- Если запросы содержат фразы *FOR UPDATE* и выражения *TABLE*.

Если первый запрос содержит выражения в списке столбцов, то для этих выражений при помощи *AS* должны быть заданы псевдонимы. Также только последний запрос может содержать фразу *ORDER BY*. Например, мы можем получить все идентификаторы магазинов, без дубликатов, с помощью следующего запроса:

```
SELECT stor_id FROM stores
UNION
SELECT stor_id FROM sales
```

PostgreSQL

PostgreSQL поддерживает базовую функциональность операторов *UNION* и *UNION ALL* в соответствии со следующим синтаксисом:

```
<оператор_select1>
UNION [ALL]
<оператор_select2>
UNION [ALL]
...
```

PostgreSQL не поддерживает оператор *UNION* и *UNION ALL* для запросов, содержащих фразу *FOR UPDATE*. Вместо *UNION DISTINCT* следует использовать функционально эквивалентный *UNION*.

Самый первый запрос не может содержать фразы *LIMIT* и *ORDER BY*. Последующие запросы могут содержать эти фразы, но такие запросы должны быть заключены в скобки. Иначе самое последнее вхождение *LIMIT* и *ORDER BY* будет применено к конечному результату.

Например, вы могли бы получить список авторов и всех сотрудников, фамилии которых начинаются на ‘P’, следующим запросом:

```
SELECT a.au_lname
FROM authors AS a
WHERE a.au_lname LIKE 'P%'
UNION
SELECT e.lname
FROM employee AS e
WHERE e.lname LIKE 'W%';
```

SQL Server

SQL Server поддерживает базовую функциональность операторов *UNION* и *UNION ALL* в соответствии со следующим синтаксисом:

```
<оператор_select1>
UNION [ALL]
<оператор_select2>
```

```
UNION [ALL]
...
```

Вместо *UNION DISTINCT* следует использовать функционально эквивалентный *UNION*.

Вы можете использовать *SELECT...INTO* вместе с *UNION* и *UNION ALL*, но *INTO* может быть использовано только в самом первом запросе объединения. Специальные ключевые слова, такие как *SELECT TOP* или *GROUP BY...WITH CUBE*, можно использовать во всех запросах объединения при условии, что если они используются в одном запросе, то они должны быть использованы и во всех остальных запросах. Если вы используете *SELECT TOP* или *GROUP BY...WITH CUBE* только в одном запросе, то оператор завершится с ошибкой.

Все объединяемые запросы должны иметь одинаковое число столбцов. Типы соответствующих столбцов не должны быть строго одинаковыми, но должны быть совместимыми. Например, одновременное использование *CHAR* и *VARCHAR* допустимо. SQL Server использует для результирующего набора наибольший из двух типов. Например, если *SELECT...UNION* содержит столбцы типов *CHAR(5)* и *CHAR(10)*, то в результирующем наборе будет использован тип *CHAR(10)*. Числовые значения конвертируются и возвращаются с наиболее точным типом данных.

В следующем примере объединяются результаты двух запросов, использующих *GROUP BY...WITH CUBE*:

```
SELECT ta.au_id, COUNT(ta.au_id)
FROM pubs..titleauthor AS ta
JOIN pubs..authors AS a ON a.au_id = ta.au_id
WHERE ta.au_id >= '722-51-5454'
GROUP BY ta.au_id WITH CUBE
UNION
SELECT ta.au_id, COUNT(ta.au_id)
FROM pubs..titleauthor AS ta
JOIN pubs..authors AS a ON a.au_id = ta.au_id
WHERE ta.au_id < '722-51-5454'
GROUP BY ta.au_id WITH CUBE
```

См. также

EXCEPT
INTERSECT
MINUS
SELECT

UPDATE

Оператор *UPDATE* изменяет существующие в таблице данные. С большой осторожностью используйте оператор *UPDATE* без фразы *WHERE*, так как в этом случае изменяются все строки таблицы.

СУБД	Уровень поддержки
MySQL	Поддерживается с вариациями
Oracle	Поддерживается с вариациями
PostgreSQL	Поддерживается с вариациями
SQL Server	Поддерживается с вариациями

Синтаксис SQL2003

```
UPDATE [ONLY] { имя_таблицы | имя_представления}
SET {{имя_столбца = { ARRAY [элемент_массива[, ...]] |
  DEFAULT | NULL | скалярное_выражение },
    имя_столбца = { ARRAY | DEFAULT |
  NULL | скалярное_выражение }
    [, ...]}
  | ROW = выражение_список}
[ WHERE условие_поиска | WHERE CURRENT OF имя_курсора ]
```

Ключевые слова

ONLY

Используется только с объектными таблицами и представлениями для указания того, что не требуется обновлять подтаблицы целевой таблицы. Попытка использования этой опции для обновления обычных таблиц приведет к ошибке.

имя_таблицы | имя_представления

Указывает целевую таблицу или представление, обновляемые оператором *UPDATE*. Вы должны иметь соответствующие привилегии для обновления таблицы. Обновление представлений имеет свои особенности. В целом рекомендуется выполнять *UPDATE* только для представлений, построенных на базе одной таблицы.

SET

Присваивает определенное значение столбцу или строке.

имя_столбца

Используется совместно с *SET* в формате *SET столбец1 = 'foo'*. Позволяет присвоить столбцу новое значение. Вы можете обновить любое количество столбцов, но нельзя обновлять один столбец несколько раз в рамках одного оператора.

ARRAY [элемент_массива[, ...]]

Записывает в столбец массив значений или пустой массив (при помощи *ARRAY[ ]*). Эта опция не очень широко поддерживается различными платформами.

NULL

Присваивает столбцу значение *NULL*.

DEFAULT

Присваивает столбцу значение, используемое по умолчанию, заданное при помощи *DEFAULT*.

скалярное_выражение

Присваивает столбцу любое одиночное значение, будь то строковый или числовой литерал, скалярная функция или скалярный подзапрос.

ROW

Используется совместно с *SET* в формате *SET ROW = ROW('foo','bar')* вместо присвоения значений отдельным столбцам в виде *SET столбец=значение*. Эта опция не очень широко поддерживается различными платформами.

выражение_список

Устанавливает значение для каждого столбца таблицы.

WHERE условие_поиска

Определяет условие отбора строк, обновляемых оператором. Можно использовать любое допустимое условие *WHERE*. Как правило, это условие проверяется отдельно для каждой строки перед выполнением оператора. Если условие поиска является подзапросом, то подзапрос выполняется для каждой строки, что может занять очень много времени.

WHERE CURRENT OF имя_курсора

Ограничивает действие оператор *UPDATE* текущей записью указанного курсора.

Общие правила

Оператор *UPDATE* используется для изменения значений в одном или нескольких столбцах строк таблицы или представления. Как правило, вы одним оператором будете обновлять одну или несколько строк. Новые значения должны быть скалярами (за исключением выражений-списков и массивов). То есть каждый столбец должен получать одиночное значение, постоянное на момент выполнения транзакции, неважно, является оно литералом или формируется функцией или подзапросом.

Простейший оператор *UPDATE* без фразы *WHERE* выглядит так:

```
UPDATE authors
SET contract = 0
```

В этом примере обнуляются статусы контрактов всех авторов (т. е. контракты с авторами разрываются). Аналогичным образом в *UPDATE* можно использовать арифметические выражения:

```
UPDATE titles
SET price = price * 1.1
```

Этот оператор *UPDATE* увеличивает цену всех книг на 10%.

Вы можете в одном операторе обновлять несколько столбцов. В следующем примере мы приводим имена и фамилии авторов к верхнему регистру:

```
UPDATE authors SET au_lname = UPPER(au_lname),
  au_fname = UPPER(au_fname);
```

Добавление в *UPDATE* фразы *WHERE* позволяет обновлять строки избирательно:

```
UPDATE titles
SET    type = 'pers_comp',
       price = (price * 1.15)
WHERE  type = 'popular_com';
```

Этот запрос вносит два изменения во все записи типа 'popular_com': цена увеличивается на 15 процентов и тип меняется на 'pers_comp'.

Вы также можете использовать функции или подзапросы для формирования значений, которые вы присваиваете в операторе *UPDATE*. В некоторых случаях вам может потребоваться обновить текущую строку открытого курсора. Следующий пример демонстрирует оба подхода:

```
UPDATE titles SET ytd_sales = (SELECT SUM(qty)
                               FROM sales
                               WHERE title_id = 'TC7777')
WHERE CURRENT OF title_cursor;
```

В этом запросе подразумевается, что вы объявили и открыли курсор с названием **title_cursor**, и в нем обрабатываются книги с идентификатором 'TC7777'.

Иногда требуется обновить значения в одной таблице на основании значений в другой таблице. Например, если вам требуется обновить дату публикации для всех книг определенного автора, то вы можете найти автора и получить список его книг через подзапросы:

```
UPDATE titles
SET pubdate = 'Jan 01 2002'
WHERE title_id IN
  (SELECT title_id
   FROM titleauthor
   WHERE au_id IN
     (SELECT au_id
      FROM authors
      WHERE au_lname = 'White'))
```

Советы и хитрости

Оператор *UPDATE* изменяет значения в существующих записях таблицы. Если вы хотите обновлять представление, то такое представление должно содержать все обязательные (*NOT NULL*) столбцы, в противном случае оно не будет обновляемым. Столбцы, имеющие значения по умолчанию, можно опускать.

Фраза *SET ROW* не поддерживается большинством платформ, и следует избегать ее использования. Фраза *WHERE CURRENT OF* поддерживается достаточно широко, но она предназначена для использования с курсорами: ознакомьтесь с предназначением и функциональностью курсоров, прежде чем использовать *WHERE CURRENT OF*.

При интерактивной работе с базой данных рекомендуется перед выполнением *UPDATE* выполнять *SELECT* с аналогичной фразой *WHERE*. Это позволяет просмотреть записи, которые будут обновлены, и убедиться, что вы не измените чего-либо, что не собирались.

MySQL

MySQL поддерживает стандарт ANSI с небольшими вариациями, включающими фразы *LOW PRIORITY*, *IGNORE* и *LIMIT*:

```
UPDATE [LOW PRIORITY] [IGNORE] имя_таблицы
SET имя_столбца = {скалярное_выражение}
  [, ...]
WHERE условие_поиска
[ORDER BY столбец1 [{ASC | DESC}][, ...]]
[LIMIT целое_число]
```

где:

LOW PRIORITY

Предписывает отложить выполнение *UPDATE* до того момента, как все клиенты завершат чтение из таблицы.

IGNORE

Указывает, что можно игнорировать сообщения об ошибках, генерируемые ограничениями *PRIMARY KEY* и *UNIQUE*. Тем не менее обновляются только те строки, при обновлении которых ошибки не возникают.

ORDER BY

Указывает порядок, в котором следует обновлять строки.

LIMIT *целое_число*

Ограничивает *UPDATE* указанным числом обновленных строк.

MySQL поддерживает присваивание значения по умолчанию с помощью *DEFAULT* в операторе *INSERT*, но в *UPDATE* такая возможность на текущий момент отсутствует.

Следующий оператор *UPDATE* увеличивает все цены в таблице **titile** на 1:

```
UPDATE titles SET price = price + 1;
```

Следующий оператор *UPDATE* увеличивает цены только для первых 10 записей в таблице **titles** (по возрастанию поля **title_id**). Цена увеличивается дважды, выражения выполняются слева направо: сперва цена удваивается, затем прибавляется 1:

```
UPDATE titles
SET price = price * 2,
price = price + 1
ORDER BY title_id
LIMIT 10;
```

Oracle

Оператор *UPDATE* в Oracle позволяет обновлять таблицы, представления, материализованные представления и подзапросы:

```
UPDATE [ONLY]
{ [схема.]{представление | материализованное представление |
  таблица}
  [@связь_с_базой_данных] [псевдоним]
  {[PARTITION (секция)] |
```

```
    [SUBPARTITION (подсекция)]}
    | подзапрос [WITH { [READ ONLY] |
      [CHECK OPTION [CONSTRAINT ограничение]] }]
    | [TABLE (выражение_коллекция) [ ( + ) ] ] }
  SET {столбец1[, ...]} = {выражение[, ...] | подзапрос} |
      VALUE [(псевдоним)] = {значение | (подзапрос)},
    {столбец2[, ...]} = {выражение[, ...] | подзапрос } |
      VALUE [(псевдоним)] = {значение | (подзапрос)},
    [, ...]
  {WHERE условие_поиска | CURRENT OF имя_курсора}
  [RETURNING выражение[, ...] INTO переменная[, ...]];
```

Синтаксические элементы оператора *UPDATE* следующие:

ONLY

Обновляет только записи указанного представления, но не представлений-потомков, образующих иерархию.

материализованное_представление

Применяет *UPDATE* к указанному материализованному представлению.

@связь_с_базой_данных

Позволяет применить *UPDATE* к объекту на удаленном сервере с помощью указанной связи с базой данных.

псевдоним

Указывает псевдоним для обновляемой таблицы. Oracle позволяет в операторе *UPDATE* указывать псевдонимы для таблиц, но не с помощью *AS*. Псевдонимы могут быть также использованы во фразе *VALUE*.

PARTITION

Применяет *UPDATE* к указанной секции, а не ко всей таблице. При обновлении секционированной таблицы имя секции указывать необязательно. Но это во многих случаях может уменьшить сложность фразы *WHERE*.

SUBPARTITION

Применяет оператор *UPDATE* к указанной подсекции, а не ко всей таблице.

WITH READ ONLY

Указывает, что подзапрос нельзя обновлять. В некоторых случаях *UPDATE* может модифицировать записи, используемые в подзапросе, а *WITH READ ONLY* предотвращает это.

WITH CHECK OPTION

Запрещает выполнять любые изменения в обновляемой таблице, если измененные данные не будут попадать в результирующее множество подзапроса.

CONSTRAINT

Указывает Oracle на допустимость выполнения только изменений, удовлетворяющих указанному ограничению целостности.

SET VALUE

Позволяет установить значения сразу для целой строки для любого типа *TABLE*. Фраза *SET VALUE* аналогична описанной в стандарте ANSI фразе *SET*

ROW. Вы также можете указать подзапрос, возвращающий все значения, требуемые *SET VALUE*.

RETURNING

Позволяет извлечь значения обновленных строк, в то время как *UPDATE* обычно просто возвращает число обновленных строк. При обновлении одной строки значения можно сохранить в переменные PL/SQL или связанные переменные. При обновлении нескольких строк значения можно сохранить в связанных массивах (bind arrays).

INTO

Указывает переменные или массивы, в которые сохраняются значения, указанные в *RETURNING*.

Существуют следующие правила для оператора *UPDATE* в Oracle:

- Оператор *UPDATE* нельзя выполнять для таблиц или представлений с базовыми таблицами, содержащими предметные индексы в статусе *FAILED* или *IN PROGRESS*. Также *UPDATE* нельзя выполнять для любых таблиц, имеющих индекс, какая-либо секция которого находится в статусе *UNUSABLE*. (Вы можете обойти это ограничение, установив параметр сессии *SKIP_UNUSABLE_INDEXES* в значение *TRUE*.)
- *UPDATE* для представлений может оказаться проблематичным, если запрос представления содержит соединение, агрегирование, операторы работы с множествами, *DISTINCT*, фразу *ORDER BY*, фразу *GROUP BY*, фразы *START WITH* и *CONNECT BY*, аналитические функции, выражения-коллекции или подзапросы. (В таких случаях вы можете создать триггер типа *INSTEAD OF* для обновления базовых таблиц представления.)

Следующие примеры демонстрируют использование расширений оператора *UPDATE* в Oracle. Для начала предположим, что таблица **sales** является большой секционированной таблицей. Вы можете обновить определенную секцию этой таблицы, используя фразу *PARTITION*. Например:

```
UPDATE sales PARTITION (sales_yr_2004) s
   SET s.payterms = 'Net 60'
 WHERE qty > 100;
```

Вы можете использовать фразу *SET VALUE* для присваивания значения каждому столбцу записи, отобранной во фразе *WHERE*:

```
UPDATE big_sales bs
SET VALUE(bs) = (SELECT VALUE(s) FROM sales s
                 WHERE bs.title_id = s.title_id)
                 AND bs.stor_id = s.stor_id)
WHERE bs.title_id = 'TC7777';
```

Вы можете получить обновленные значения с помощью фразы *RETURNING*. В следующем примере обновляется одна строка, и новые значения сохраняются в локальных переменных:

```
UPDATE employee
   SET job_id = 13, job_level = 140
 WHERE last_name = 'Josephs'
```

```
RETURNING last_name, job_id
     INTO :var1, :var2;
```

PostgreSQL

PostgreSQL поддерживает стандартный синтаксис *UPDATE* с небольшими отличиями: поддерживается фраза *FROM*, а также работа с массивами, но без фразы *ARRAY*. Синтаксис следующий:

```
UPDATE [ONLY] {таблица | представление}
SET столбец1 = {DEFAULT | NULL | скалярное_выражение},
    столбец2 = {DEFAULT | NULL | скалярное_выражение}
    [, ...]
[FROM {таблица1 [AS псевдоним], таблица2 [AS псевдоним]
    [, ...]}]
[ WHERE условие_поиска | WHERE CURRENT OF имя_курсора ]
```

Большинство элементов синтаксиса соответствуют ANSI. Единственный нестандартный элемент следующий:

FROM

Позволяет построить условие поиска обновляемых строк на базе соединения. *FROM* не используется, когда для поиска обновляемых строк достаточно одной (обновляемой) таблицы.

В следующем примере мы хотим обновить поле **job_lvl** для сотрудников, имеющих в **job_id** значение 12 и в **min_lvl** значение 25. Мы могли бы сделать это с помощью большого сложного подзапроса либо построив соединение с помощью *FROM*:

```
UPDATE employee
   SET job_lvl = 80
  FROM employee AS e, jobs AS j
 WHERE e.job_id = j.job_id
   AND j.job_id = 12
   AND j.min_lvl = 25;
```

Все остальные возможности оператора, описанные в ANSI, полностью поддерживаются.

SQL Server

SQL Server поддерживает большую часть возможностей *UPDATE*, описанных в ANSI, за исключением ключевых слов *ONLY* и *ARRAY* и работы с массивами. Однако в SQL Server в оператор *UPDATE* добавлена возможность использовать фразу *WITH*, подсказки оптимизатору во фразе *OPTION*, а также работа с переменными.

```
UPDATE {таблица | представление | набор_строк}
[WITH (подсказка1, подсказка2[, ...])]
SET {столбец = {DEFAULT | NULL | скалярное_выражение}
  | переменная = скалярное_выражение
  | переменная = столбец = скалярное_выражение }[, ...]
[FROM {таблица1 | представление1 |
  вложенное_представление1 | набор_строк1}[, ...]]
  [AS псевдоним]
[JOIN {таблица2[, ...]}]
```

```
WHERE {условие_поиска | CURRENT OF [GLOBAL] курсор}
[OPTION (подсказка1, подсказка2[, ...])]
```

Элементы синтаксиса оператора *UPDATE* следующие:

WITH подсказка

Позволяет использовать подсказки оптимизатору для влияния на создаваемый план выполнения. Так как оптимизатор запросов обычно создает достаточно хорошие планы выполнения, то применяйте подсказки, только имея хорошее представление об используемых в операторе таблицах, индексах и данных. Без этого использование подсказок может только ухудшить, а не улучшить производительность.

переменная

Переменные должны быть объявлены перед оператором *UPDATE* в формате *DECLARE @переменная*. Конструкция *SET @переменная = столбец1 = выражение1* присваивает переменной конечное значение обновляемого столбца, в то время как *SET @переменная = столбец1, столбец1 = выражение* присваивает переменной значение столбца до обновления.

FROM

Позволяет построить условие поиска обновляемых строк на базе соединения. *FROM* не используется, когда для поиска обновляемых строк достаточно одной (обновляемой) таблицы. Функции для работы с множествами строк описываются далее в этом разделе.

AS псевдоним

Присваивает простой в использовании псевдоним таблице, представлению, вложенному табличному подзапросу или функции, возвращающей набор строк.

JOIN

Позволяет использовать стандартный синтаксис соединений во фразе *FROM*.

GLOBAL

Небольшое расширение описанной в ANSI фразы *WHERE CURRENT OF*. Фраза *WHERE CURRENT OF* позволяет обновить только строку, на которой в текущей момент позиционирован определенный курсор. По умолчанию подразумевается, что курсор локальный, а при использовании глобальных курсоров можно использовать ключевое слово *GLOBAL*.

OPTION подсказка

Позволяет использовать подсказки оптимизатору для влияния на создаваемый план выполнения. Так же, как и при использовании табличных подсказок оптимизатору в *WITH*, применяйте подсказки, только имея хорошее представление об используемых в операторе таблицах, индексах и данных. Без этого использование подсказок может только ухудшить, а не улучшить производительность.

Основным расширением оператора *UPDATE* по сравнению с ANSI является фраза *FROM*. *FROM* позволяет использовать соединения таблиц для обновления только тех строк в целевой таблице, для которых выполняются определенные условия в связанных строках присоединяемых таблиц. В следующем примере показывает

обновление таблицы в стиле ANSI с использованием достаточно запутанного подзапроса, а затем то же обновление в стиле SQL Server с использованием соединения. Оба оператора решают одну и ту же задачу, но очень разными способами:

```
-- стиль ANSI
UPDATE titles
SET pubdate = GETDATE( )
WHERE title_id IN
(SELECT title_id
FROM titleauthor
WHERE au_id IN
(SELECT au_id
FROM authors
WHERE au_lname = 'White'))
-- стиль Microsoft Transact-SQL
UPDATE titles
SET pubdate = GETDATE( )
FROM authors AS a
JOIN titleauthor AS t2 ON a.au_id = t2.au_id
WHERE t2.title_id = titles.title_id
AND a.au_lname = 'White'
```

Обновление в стиле Transact-SQL состоит по сути в соединении двух таблиц – **authors** и **titelauthor** – в таблицу **titles**. Для выполнения обновления в стиле ANSI сначала нужно получить из таблицы **auhors** значение **au_id** и передать в **titleauthor**, а затем получить **title_id** и передать в основной оператор *UPDATE*.

В следующем примере обновляется столбец state для первых 10 авторов из таблицы **authors**:

```
UPDATE authors
SET state = 'ZZ'
FROM (SELECT TOP 10 * FROM authors ORDER BY au_lname) AS t1
WHERE authors.au_id = t1.au_id
```

Важно заметить, что обычно обновление N первых записей представляет определенную трудность, если только не существует явная нумерация строк, которую можно использовать в *WHERE*. Однако в этом примере вложенный табличный подзапрос во фразе *FROM* использует *TOP* для получения первых 10 строк, таким образом экономя значительные усилия программиста.

См. также

DECLARE
SELECT
WHERE

WHERE

Фраза *WHERE* определяет условия поиска для таких операторов, как *SELECT*, *UPDATE* и *DELETE*. Любые строки, не удовлетворяющие указанному условию, не затрагиваются оператором. Условие поиска может включать вычисления, булевы операторы и предикаты SQL (например, *LIKE* или *BETWEEN*).

СУБД	Уровень поддержки
MySQL	Поддерживается
Oracle	Поддерживается
PostgreSQL	Поддерживается
SQL Server	Поддерживается

Синтаксис SQL2003

```
{ WHERE условие_поиска | WHERE CURRENT OF имя_курсора }
```

Ключевые слова

WHERE условие_поиска

Определяет условие отбора строк, затрагиваемых оператором.

WHERE CURRENT OF имя_курсора

Ограничивает действие оператора текущей строкой указанного курсора.

Общие правила

Фраза *WHERE* используется в операторах *SELECT*, *DELETE*, *INSERT...SELECT*, *UPDATE* и в любых операторах, содержащих запросы и подзапросы (таких как *DECLARE*, *CREATE TABLE*, *CREATE VIEW* и т. д.).

Условия поиска, каждое из которых уже обсуждалось в книге отдельно, включают:

Сравнение с множеством строк (=ALL, >ALL, <= ALL, SOME/ANY)

Для получения списка издателей, живущих в тех же городах, что и их авторы, можно использовать следующий запрос:

```
SELECT pub_name
FROM   publishers
WHERE  city = SOME (SELECT city FROM authors);
```

Комбинирование условий (AND, OR и NOT)

Например, для получения списка авторов с продажами не менее 75 единиц, либо с гонораром не менее 60, можно использовать следующий запрос:

```
SELECT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE ta.title_id IN (SELECT title_id FROM sales
                      WHERE qty >= 75)
   OR (a.au_id IN (SELECT au_id FROM titleauthor
                   WHERE royaltyper >= 60)
   AND a.au_id IN (SELECT au_id FROM titleauthor
                   WHERE au_ord = 2));
```

Операторы сравнения (такие как =, < >, <, >, <= и >=)

Следующий запрос извлекает имена и фамилии авторов, не имеющих на текущий момент контракта (поле **contract** равно 0):

```
SELECT au_lname, au_fname
```

```
FROM authors
WHERE contract = 0;
```

Операторы работы со списками (IN и NOT IN)

Получить список всех авторов, для которых нет записей в таблице **titleauthor**, можно так:

```
SELECT au_fname, au_lname
FROM authors
WHERE au_id NOT IN (SELECT au_id FROM titleauthor);
```

Операторы сравнения с NULL (IS NULL и IS NOT NULL)

Для получения списка наименований, для которых с начала года не было ни одной продажи, можно использовать такой запрос:

```
SELECT title_id, SUBSTRING(title, 1, 25) AS title
FROM titles
WHERE ytd_sales IS NULL;
```

Не используйте в запросах =NULL. NULL – это неизвестное значение и оно не может быть равно чему-либо еще. Использование =NULL не равносильно использованию IS NULL.

Операторы сравнения с шаблонами (LIKE и NOT LIKE)

Например, мы можем получить список авторов, фамилии которых начинаются на С:

```
SELECT au_id
FROM authors
WHERE au_lname LIKE 'C%';
```

Операторы сравнения с диапазонами (BETWEEN и NOT BETWEEN)

Например, мы можем получить список авторов, фамилии которых по алфавиту находятся между Smith и White:

```
SELECT au_lname, au_fname
FROM authors
WHERE au_lname BETWEEN 'smith' AND 'white';
```

Советы и хитрости

При использовании во фразе *WHERE* некоторых типов данных (таких как *LOB*) или определенных кодировок (включая *UNICODE*) могут потребоваться специальные конструкции.

Скобки используются для определения порядка вычисления фразы *WHERE*. Выражение, заключенное в скобки, вычисляется раньше других. Вложенные скобки задают иерархию вычислений. Скобки, вложенные глубже всего, вычисляются самими первыми. Существуют две причины, по которым использовать скобки следует осторожно:

- Всегда должно быть одинаковое число открывающих и закрывающих скобок. Нарушение этого баланса приводит к ошибке.
- Всегда нужно внимательно следить за расположением скобок, так как расположение скобки не в том месте может значительно менять результат запроса.

К примеру, рассмотрим следующий запрос, возвращающий 6 строк из базы данных **pubs** на SQL Server:

```
SELECT DISTINCT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE ta.title_id IN (SELECT title_id FROM sales
WHERE qty >= 75)
OR (a.au_id IN (SELECT au_id FROM titleauthor
WHERE royaltyper >= 60)
AND a.au_id IN (SELECT au_id FROM titleauthor
WHERE au_ord = 2))
```

Результат этого запроса следующий:

```
au_id
-----------
213-46-8915
724-80-9391
899-46-2035
998-72-3567
```

Изменение всего лишь одной пары скобок приводит к другому результату:

```
SELECT DISTINCT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE (ta.title_id IN (SELECT title_id FROM sales
WHERE qty >= 75)
OR a.au_id IN (SELECT au_id FROM titleauthor
WHERE royaltyper >= 60))
AND a.au_id IN (SELECT au_id FROM titleauthor
WHERE au_ord = 2)
```

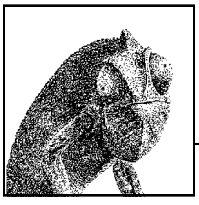
На этот раз результат будет следующим:

```
au_id
-----------
213-46-8915
724-80-9391
899-46-2035
```

Все платформы, рассматриваемые в этой книге, поддерживают стандарт ANSI.

См. также

ALL/ANY/SOME
BETWEEN
DECLARE CURSOR
DELETE
EXISTS
IN
LIKE
SELECT
UPDATE



4
Функции SQL

В SQL функция – это частный случай команды. В каждом диалекте SQL реализован свой набор таких команд. По существу, функция представляет собой команду, название которой состоит из одного слова и которая возвращает единственное значение. Значение функции зависит от входных параметров. Например, функция может вычислять среднее арифметическое для списка значений, выбранных из базы данных. Однако у многих функций входных параметров нет вообще. Примером может служить функция *CURRENT_TIME*, возвращающая текущее время.

В стандарте ANSI описан целый ряд полезных функций. В этой главе мы рассмотрим их все и приведем подробные описания и примеры для каждой платформы. Дополнительно любая СУБД поддерживает множество собственных функций, не вошедших в стандарт. Мы опишем назначение и параметры таких функций тоже.

В большинстве СУБД имеется также возможность создавать определяемые пользователем функции (user-defined functions – UDF). Подробнее о них см. главу 3.

Классификация функций

Существует несколько способов разбиения функций на группы. В следующих разделах описаны отличия, важные для понимания механизмов работы функций.

Детерминированные и недетерминированные функции

Функция может быть детерминированной или недетерминированной. *Детерминированная функция* возвращает один и тот же результат при вызове с одинаковыми значениями входных параметров. *Недетерминированная функция* может возвращать разные результаты при каждом вызове, даже если на вход подаются одни и те же значения.

Почему так важно, что при одинаковых входных параметрах возвращается один и тот же результат? Это важно, чтобы понимать, как использовать функции в представлениях, внутри определяемых пользователем функций или в храни-

мых процедурах. Ограничения различны в разных реализациях, но иногда в коде вышеперечисленных объектов разрешается использовать только детерминированные функции. Например, SQL Server позволяет строить индекс по столбцу-выражению только в случае, когда выражение не содержит недетерминированных функций. Правила работы с функциями, действующие на конкретной платформе, должны быть описаны в прилагаемой к ней документации.

Агрегатные и скалярные функции

Функции можно классифицировать также в зависимости от того, применяются они к значениям, взятым только из одной строки базы данных, или к набору строк. *Агрегатные функции* применяются к набору строк и возвращают единственное итоговое значение. *Скалярные функции* также возвращают единственное значение, зависящее от переданных скалярных аргументов. Некоторые скалярные функции, например *CURRENT_TIME*, вызываются без аргументов.

Оконные функции

Оконные функции похожи на агрегатные в том смысле, что применяются сразу к нескольким строкам. Разница в том, как эти строки задаются. Агрегатная функция применяется к набору строк, определяемому указанной в запросе фразой *GROUP BY*. В случае же оконных функций набор строк задается при каждом вызове, поэтому разные вызовы некоторой функции в одном и том же запросе могут работать с разными наборами строк.

Агрегатные функции в ANSI SQL

Агрегатная функция возвращает одно значение, зависящее от множества входных данных. Если она входит в список выражений в команде *SELECT*, то эта команда должна содержать фразу *GROUP BY*, возможно, в сочетании с *HAVING*. Наличие *GROUP BY* или *HAVING* необязательно, если агрегатная функция – единственное выражение, отбираемое командой *SELECT*. В табл. 4.1 перечислены агрегатные функции, описанные в стандарте ANSI SQL.

Таблица 4.1. Агрегатные функции в ANSI SQL

Функция	Назначение
AVG(*expression*)	Вычисляет среднее для колонки, указанной в выражении *expression*
CORR(*dependent*, independent)	Вычисляет коэффициент корреляции
COUNT(*expression*)	Вычисляет выражение *expression* для каждой строки в группе и возвращает количество значений, отличных от NULL
COUNT(*)	Вычисляет общее количество строк в указанной таблице или представлении
COVAR_POP(*dependent*, *independent*)	Вычисляет ковариацию генеральной совокупности
COVAR_SAMP(*dependent*, *independent*)	Вычисляет выборочную ковариацию

Функция	Назначение
CUME_DIST(*value_list*) WITHIN GROUP (ORDER BY *sort_list*)	Вычисляет относительный ранг гипотетической строки в группе строк, где рангом называется количество строк, меньших или равных гипотетической строке, поделенное на количество строк в данной группе
DENSE_RANK(*value_list*) WITHIN GROUP (ORDER BY *sort_list*)	Вычисляет плотный ранг (ни один ранг не пропускается) для гипотетической строки (*value_list*) в группе строк, описываемой фразой *GROUP BY*
MIN(*expression*)	Находит минимальное значение в столбце, определяемом выражением *expression*
MAX(*expression*)	Находит максимальное значение в столбце, определяемом выражением *expression*
PERCENT_RANK(*value_list*) WITHIN GROUP (ORDER BY *sort_list*)	Вычисляет относительный ранг гипотетической строки путем деления ранга этой строки минус 1 на количество строк в группе
PERCENTILE_CONT(*percentile*) WITHIN GROUP (ORDER BY *sort_list*)	Вычисляет интерполированное значение, которое, будучи прибавлено к группе, даст указанный процентиль *percentile*
PERCENTILE_DISC(*percentile*) WITHIN GROUP (ORDER BY *sort_list*)	Возвращает значение, для которого наименьшая величина функции кумулятивного распределения больше или равна *percentile*
RANK(*value_list*) WITHIN GROUP (ORDER BY *sort_list*)	Вычисляет ранг гипотетической строки (*value_list*) в группе строк, определяемой фразой *GROUP BY*
REGR_AVGX(*dependent*, *independent*)	Вычисляет среднее значение независимой переменной
REGR_AVGY(*dependent*, *independent*)	Вычисляет среднее значение зависимой переменной
REGR_COUNT(*dependent*, *independent*)	Вычисляет количество пар, остающееся в группе после удаления всех пар, в которых хотя бы одно значение равно NULL
REGR_INTERCEPT(*dependent*, *independent*)	Находит точку пересечения с осью y линии регрессии, вычисленной методом наименьших квадратов
REGR_R2(*dependent*, *independent*)	Возвращает квадрат коэффициента корреляции
REGR_SLOPE(*dependent*, *independent*)	Возвращает наклон линии регрессии, вычисленной методом наименьших квадратов
REGR_SXX(*dependent*, *independent*)	Вычисляет сумму квадратов независимых переменных
REGR_SXY(*dependent*, *independent*)	Вычисляет сумму попарных произведений переменных
REGR_SYY(*dependent*, *independent*)	Вычисляет сумму квадратов зависимых переменных

Таблица 4.1 (продолжение)

Функция	Назначение
STDDEV_POP(*expression*)	Вычисляет стандартное отклонение генеральной совокупности для колонки, указанной в выражении *expression*
STDDEV_SAMP(*expression*)	Вычисляет выборочное стандартное отклонение для колонки, указанной в выражении *expression*
SUM(*expression*)	Вычисляет сумму значений в колонке, указанной в выражении *expression*
VAR_POP(*expression*)	Вычисляет дисперсию генеральной совокупности для колонки, указанной в выражении *expression*
VAR_SAMP(*expression*)	Вычисляет выборочную дисперсию для колонки, указанной в выражении *expression*

Строго говоря, функции *ALL*, *ANY* и *SOME* также считаются агрегатными функциями. Однако мы рассматривали их в контексте критериев поиска по диапазону, поскольку именно так они чаще всего и используются. Подробнее об этих функциях см. главу 3.

Количество значений, обрабатываемых агрегатной функцией, зависит от того, сколько строк вернул запрос. Этим агрегатные функции отличаются от скалярных, которые применяются только к одной строке за один вызов.

Общий синтаксис вызова агрегатной функции таков:

```
aggregate_function_name( [ALL | DISTINCT] expression )
```

Здесь *aggregate_function_name* может быть *AVG*, *COUNT*, *MAX*, *MIN* или *SUM* (см. табл. 4.1). Ключевое слово *ALL*, подразумеваемое по умолчанию, означает, что в процессе агрегирования учитываются все строки. Если же задано ключевое слово *DISTINCT*, то учитываются только отличающиеся значения.

При вычислении любой агрегатной функции, кроме *COUNT(*)*, значения NULL игнорируются.

AVG и SUM

Функция *AVG* вычисляет среднее арифметическое значений по столбцу или выражению, а функция *SUM* – сумму значений. Обе работают только с числовыми значениями и игнорируют NULL. Чтобы вычислить среднее или сумму различных значений столбца или выражения, укажите ключевое слово *DISTINCT*.

Синтаксис в стандарте ANSI SQL

```
AVG( [ALL | DISTINCT] expression )
SUM( [ALL | DISTINCT] expression )
```

MySQL, PostgreSQL и SQL Server

На этих платформах поддерживается синтаксис *AVG* и *SUM*, утвержденный стандартом ANSI SQL.

Oracle

Oracle поддерживает синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
AVG( [ALL | DISTINCT] expression ) OVER (window_clause)
SUM( [ALL | DISTINCT] expression ) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

Примеры

В следующем запросе для каждого типа книг вычисляется средний объем продаж с начала года:

```
SELECT type, AVG( ytd_sales ) AS "average_ytd_sales" FROM titles GROUP BY type;
```

А в этом запросе для каждого типа книг вычисляется общий объем продаж с начала года:

```
SELECT type, SUM( ytd_sales ) FROM titles GROUP BY type;
```

CORR

Функция *CORR* возвращает коэффициент корреляции между наборами зависимых и независимых переменных.

Синтаксис в стандарте ANSI SQL

При вызове функции указываются две переменные – зависимая (*dependent*) и независимая (*independent*):

```
CORR(dependent, independent)
```

Те пары, в которых хотя бы одна переменная равна NULL, игнорируются. Если не встретилось ни одной пары, содержащей два отличных от NULL значения, то функция возвращает NULL.

Oracle

Oracle поддерживает синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
CORR(dependent, independent) OVER (window_ clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

PostgreSQL

PostgreSQL поддерживает синтаксис ANSI SQL.

MySQL and SQL Server

На этих платформах функция *CORR* не реализована.

Пример

В следующем примере функция *CORR* применяется к данным, которые отбирает первый *SELECT*:

```
SELECT * FROM test2;
         Y          X
---------- ----------
         1          3
         2          2
         3          1

SELECT CORR(y,x) FROM test2;
CORR(Y,X)
---------
       -1
```

COUNT

Функция *COUNT* вычисляет количество строк, определяемых выражением.

Синтаксис в стандарте ANSI SQL

```
COUNT(*)
COUNT( [ALL | DISTINCT] expression )
```

COUNT(*)

Подсчитывает все строки в указанной таблице, в том числе содержащие NULL.

COUNT([ALL|DISTINCT] expression)

Подсчитывает количество строк, для которых указанный столбец или выражение не равны NULL. Если задано ключевое слово *DISTINCT*, то дубликаты игнорируются, то есть возвращается количество различных значений. Если же задано ключевое слово *ALL* (оно подразумевается по умолчанию), то возвращаются все строки, для которых значение выражения отлично от NULL.

MySQL, PostgreSQL и SQL Server

На этих платформах поддерживается синтаксис *COUNT*, утвержденный стандартом ANSI SQL.

Oracle

Oracle поддерживает синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
COUNT ({* | [DISTINCT] expression}) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

Примеры

Следующий запрос подсчитывает, сколько всего строк в таблице:

```
SELECT COUNT(*) FROM publishers;
```

А ниже мы находим количество различных стран, в которых есть издательства:

```
SELECT COUNT(DISTINCT country) "Count of Countries"
FROM publishers
```

COVAR_POP

Функция *COVAR_POP* вычисляет ковариацию генеральной совокупности, состоящей из набора зависимых и независимых переменных.

Синтаксис в стандарте ANSI SQL

При вызове функции указываются две переменные – зависимая и независимая:

```
COVAR_POP(dependent, independent)
```

Те пары, в которых хотя бы одна переменная равна NULL, игнорируются. Если после отбрасывания NULL-значений в группе не осталось ни одной пары, то функция возвращает NULL.

Oracle

Oracle поддерживает синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
COVAR_POP(dependent, independent) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

PostgreSQL

PostgreSQL поддерживает синтаксис ANSI SQL.

MySQL и SQL Server

На этих платформах функция *COVAR_POP* не реализована.

Пример

В следующем примере функция *COVAR_POP* применяется к данным, которые отбирает первый *SELECT*:

```
SELECT * FROM test2;
         Y          X
---------- ----------
         1          3
         2          2
         3          1

SELECT COVAR_POP(y,x) FROM test2;
COVAR_POP(Y,X)
--------------
    -.66666667
```

COVAR_SAMP

Функция *COVAR_SAMP* вычисляет выборочную ковариацию зависимой и независимой переменных.

Синтаксис в стандарте ANSI SQL

При вызове функции указываются две переменные – зависимая и независимая:

```
COVAR_SAMP(dependent, independent)
```

Те пары, в которых хотя бы одна переменная равна NULL, игнорируются. Если после отбрасывания NULL-значений в группе не осталось ни одной пары, то функция возвращает NULL.

Oracle

Oracle поддерживает синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
COVAR_SAMP(dependent, independent) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

PostgreSQL

PostgreSQL поддерживает синтаксис ANSI SQL.

MySQL и SQL Server

На этих платформах функция *COVAR_SAMP* не реализована.

Пример

В следующем примере функция *COVAR_SAMP* применяется к данным, которые отбирает первый *SELECT*:

```
SELECT * FROM test2;
         Y          X
---------- ----------
         1          3
         2          2
         3          1

SELECT COVAR_SAMP(y,x) FROM test2;
COVAR_SAMP(Y,X)
---------------
             -1
```

CUME_DIST

Вычисляет относительный ранг гипотетической строки в группе строк по следующей формуле:

```
(строки,_предшествующие_гипотетической, + строки,_равные_гипотетической) /
количество_строк_в_группе
```

Имейте в виду, что *количество строк в группе* включает и гипотетическую строку.

Синтаксис в стандарте ANSI SQL

```
CUME_DIST(value_list) WITHIN GROUP (ORDER BY sort_list)
value_list ::= expression[, expression...]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Элементы списка *value_list* позиционно соответствуют элементам списка *sort_list*. Поэтому количество выражений в обоих списках должно быть одинаково.

Oracle

Oracle поддерживает синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
CUME_DIST( ) OVER ([partitioning] ordering)
```

О том, что означают параметры *partitioning* и *ordering*, см. раздел «Оконные функции в ANSI SQL» ниже.

MySQL, PostgreSQL и SQL Server

На этих платформах функция *CUME_DIST* не реализована.

Пример

В следующем примере вычисляется относительный ранг гипотетической новой строки (num=4, odd=1) внутри каждой группы строк из таблицы **test4** в предположении, что группы отличаются значением в столбце **odd**:

```
SELECT * FROM test4;
       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1

SELECT odd, CUME_DIST(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4 GROUP BY odd;
       ODD CUME_DIST(4,1)WITHINGROUP(ORDERBYNUM,ODD)
---------- ----------------------------------------
         0                                        1
         1                                       .8
```

В группе odd=0 новая строка располагается после строк (0,0), (2,0) и (4,0). Она совпадает сама с собой. Общее количество строк в группе с учетом гипотетической равно четырем. Поэтому относительный ранг вычисляется по формуле:

```
(3 предшествующих строки + 1 совпадающая) / (3 в группе + 1 гипотетическая)= 4 / 4 = 1
```

В группе odd=1 новая строка располагается после строк (1,1), (3,1) и еще одной (3,1). Как и в предыдущем случае, совпадающая строка только одна – сама гипо-

тетическая. Общее количество строк в группе с учетом гипотетической равно пяти. Относительный ранг равен:

```
(3 предшествующих строки + 1 совпадающая) / (4 в группе + 1 гипотетическая)= 4 / 5 = .8
```

DENSE_RANK

Вычисляет ранг указанной вами гипотетической строки в группе. Имеется в виду *плотный ранг*, то есть значения рангов не содержат пропусков, даже если в группе имеются строки с одинаковым рангом.

Синтаксис в стандарте ANSI SQL

```
DENSE_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
value_list ::= expression[, expression...]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Элементы списка *value_list* позиционно соответствуют элементам списка *sort_list*. Поэтому количество выражений в обоих списках должно быть одинаково.

Oracle и SQL Server

Oracle и SQL Server поддерживают синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
DENSE_RANK( ) OVER ([partitioning] ordering)
```

О том, что означают параметры *partitioning* и *ordering*, см. раздел «Оконные функции в ANSI SQL» ниже.

MySQL и PostgreSQL

На этих платформах функция *DENSE_RANK* не реализована.

Пример

В следующем примере вычисляется плотный ранг гипотетической новой строки (num=4, odd=1) внутри каждой группы строк из таблицы **test4** в предположении, что группы отличаются значением в столбце **odd**:

```
SELECT * FROM test4;
       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1

SELECT odd, DENSE_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4 GROUP BY odd;
       ODD DENSE_RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
---------- ----------------------------------------
         0                                        4
         1                                        3
```

В группе odd=0 новая строка располагается после строк (0, 0), (2, 0) и (4, 0) и, следовательно, находится в позиции 4. В группе odd=1 новая строка располагается после строк (1, 1), (3, 1) и второй строки (3, 1). В данном случае ранг обеих строк-дубликатов равен #2, поэтому ранг новой строки равен #3. Сравните с функцией *RANK*, которая возвращает иной результат.

MIN и MAX

Функции *MIN(expression)* и *MAX(expression)* находят соответственно минимальное и максимальное значения выражения *expression* (оно может быть строкой, датой/временем или числом) в наборе строк. Разрешается указывать ключевое слово *DISTINCT* или *ALL*, но на результат это не влияет.

Синтаксис в стандарте ANSI SQL

```
MIN( [ALL | DISTINCT] expression )
MAX( [ALL | DISTINCT] expression )
```

MySQL

MySQL поддерживает синтаксис ANSI SQL для функций *MIN* и *MAX*, а также эквивалентные им функции *LEAST* и *GREATEST*.

Oracle

Oracle поддерживает синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
MIN({ALL|[DISTINCT] expression}) OVER (window_clause)
MAX({ALL|[DISTINCT] expression}) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

PostgreSQL и SQL Server

На этих платформах поддерживается синтаксис ANSI SQL.

Примеры

Следующий запрос возвращает самую лучшую и самую худшую сумму продаж с начала года для всех хранящихся в базе книг:

```
SELECT MIN(ytd_sales), MAX(ytd_sales)
FROM   titles;
```

Агрегатные функции также часто используются во фразе *HAVING* запросов с фразой *GROUP BY*. Следующий запрос отбирает все категории (типы) книг, для которых средняя по категории цена больше $15.00:

```
SELECT type 'Category', AVG( price ) 'Average Price'
FROM   titles
GROUP BY type
HAVING AVG(price) > 15
```

PERCENT_RANK

Вычисляет относительный ранг гипотетической строки путем деления ранга этой строки минус 1 на количество строк в группе (не включая гипотетическую строку).

Синтаксис в стандарте ANSI SQL

```
PERCENT_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
value_list ::= expression[, expression...]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Элементы списка *value_list* позиционно соответствуют элементам списка *sort_list*. Поэтому количество выражений в обоих списках должно быть одинаковым.

Oracle

Oracle поддерживает синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
PERCENT_RANK( ) OVER ([partitioning] ordering)
```

О том, что означают параметры *partitioning* и *ordering*, см. раздел «Оконные функции в ANSI SQL» ниже.

MySQL, PostgreSQL и SQL Server

На этих платформах функция *PERCENT_RANK* не реализована.

Пример

В следующем примере вычисляется процентный ранг гипотетической новой строки (num=4, odd=1) внутри каждой группы строк из таблицы **test4** в предположении, что группы отличаются значением в столбце **odd**:

```
SELECT * FROM test4;
       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1

SELECT odd, PERCENT_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4 GROUP BY odd;
       ODD PERCENT_RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
---------- ---------------------------------------------
         0                                            1
         1                                          .75
```

В группе odd=0 новая строка располагается после строк (0, 0), (2, 0) и (4, 0) и, следовательно, находится в позиции 4. Ее ранг равен (ранг 4-й строки − 1) / 3 строки = 100%. В группе odd=1 новая строка располагается после строк (1, 1), (3, 1) и второй

строки(3, 1), то есть опять-таки находится в позиции 4. Вычисление ранга дает (ранг 4-й строки – 1) / 4 строки = 3/4 = 75%.

PERCENTILE_CONT

Вычисляет интерполированное значение, соответствующее заданному вами процентилю.

Синтаксис в стандарте ANSI SQL

В следующей команде *percentile* – число от 0 до 1:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY sort_list)
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Oracle

Oracle допускает только одно выражение во фразе *ORDER BY*:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY expression)
```

Кроме того, в Oracle синтаксис расширен для использования в аналитических приложениях:

```
PERCENTILE_CONT(percentile) WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

О том, что означает параметр *partitioning*, см. раздел «Оконные функции в ANSI SQL» ниже.

MySQL, PostgreSQL и SQL Server

На этих платформах функция *PERCENTILE_CONT* не реализована.

Пример

В следующем примере данные из таблицы **test4** сгруппированы по столбцу **odd**, и функция *PERCENTILE_CONT* возвращает 50-процентный процентиль для каждой группы:

```
SELECT * FROM test4;
       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1

SELECT odd, PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY NUM)
FROM test4 GROUP BY odd;
       ODD PERCENTILE_CONT(0.50)WITHINGROUP(ORDERBYNUM)
---------- ---------------------------------------------
         0                                            2
         1                                            3
```

PERCENTILE_DISC

Находит в группе значение, для которого наименьшая величина функции кумулятивного распределения больше или равна заданному процентилю.

Синтаксис в стандарте ANSI SQL

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY sort_list)
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

где *percentile* – число от 0 до 1.

Oracle

Oracle допускает только одно выражение во фразе *ORDER BY*:

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY expression)
```

Кроме того, в Oracle синтаксис расширен для использования в аналитических приложениях:

```
PERCENTILE_DISC(percentile) WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

О том, что означает параметр *partitioning*, см. раздел «Оконные функции в ANSI SQL» ниже.

MySQL, PostgreSQL и SQL Server

На этих платформах функция *PERCENTILE_DISC* не реализована.

Пример

Следующий пример аналогичен примеру для функции *PERCENTILE_CONT*, но для каждой группы возвращает значение, наиболее близкое, но не превосходящее процентиля 60%:

```
SELECT * FROM test4;
       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1

SELECT odd, PERCENTILE_DISC(0.60) WITHIN GROUP (ORDER BY NUM)
FROM test4 GROUP BY odd;
PERCENTILE_CONT(0.60)WITHIN GROUP(ORDERBYNUM)
---------------------------------------------
                                           2
                                           3
```

RANK

Вычисляет ранг заданной вами гипотетической строки в группе. Ранг не является плотным. Если в группе есть строки с одинаковым рангом, то некоторые ранги могут быть пропущены. Если вам нужен плотный ранг, пользуйтесь функцией *DENSE_RANK*.

Синтаксис в стандарте ANSI SQL

```
RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
value_list ::= expression[, expression...]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Элементы списка *value_list* позиционно соответствуют элементам списка *sort_list*. Поэтому количество выражений в обоих списках должно быть одинаково.

Oracle и SQL Server

Oracle и SQL Server поддерживают синтаксис ANSI SQL, а также следующую форму для аналитических приложений:

```
RANK( ) OVER ([partitioning] ordering)
```

О том, что означают параметры *partitioning* и *ordering*, см. раздел «Оконные функции в ANSI SQL» ниже.

MySQL и PostgreSQL

На этих платформах функция *RANK* не реализована.

Пример

В следующем примере вычисляется ранг гипотетической новой строки (`num=4`, `odd=1`) внутри каждой группы строк из таблицы **test4** в предположении, что группы отличаются значением в столбце **odd**:

```
SELECT * FROM test4;
       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1

SELECT odd, RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4 GROUP BY odd;
       ODD RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
---------- ------------------------------------
         0                                    4
         1                                    4
```

В обоих случаях ранг гипотетической новой строки равен 4. В группе `odd=0` новая строка располагается после строк (`0,0`), (`2,0`) и (`4,0`), т.е. в позиции 4. В группе

odd=1 новая строка оказывается после строк (1, 1), (3, 1) и второй строки (3, 1). Хотя две последних строки одинаковы и, значит, имеют один и тот же ранг, новой строке все равно приписывается ранг 4, поскольку ей предшествуют три строки. Сравните с поведением функции *DENSE_RANK*.

REGR, семейство функций

В стандарте ANSI SQL определено семейство функций с именами, начинающимися с *REGR_*. Все они так или иначе связаны с линейной регрессией и относятся к линии регрессии, полученной методом наименьших квадратов.

Синтаксис в стандарте ANSI SQL

Ниже приведены синтаксис и краткое описание каждой функции семейства *REGR_*:

*REGR_AVGX(*dependent, independent*)*

Среднее значение (аналогично *AVG(x)*) *независимой* переменной.

*REGR_AVGY(*dependent, independent*)*

Среднее значение (аналогично *AVG(y)*) *зависимой* переменной.

*REGR_COUNT(*dependent, independent*)*

Количество пар, остающееся в группе после удаления всех пар, в которых хотя бы одно значение равно NULL.

*REGR_INTERCEPT(*dependent, independent*)*

Находит точку пересечения линии регрессии, вычисленной методом наименьших квадратов, с осью y.

*REGR_R2(*dependent, independent*)*

Вычисляет коэффициент детерминации.

*REGR_SLOPE(*dependent, independent*)*

Вычисляет наклон линии регрессии.

*REGR_SXX(*dependent, independent*)*

Сумма квадратов значений независимой переменной.

*REGR_SXY(*dependent, independent*)*

Сумма попарных произведений значений.

*REGR_SYY(*dependent, independent*)*

Сумма квадратов значений зависимой переменной.

Функции семейства *REGR_* учитывают только пары, в которых оба числа отличны от NULL. Пары, в которых хотя бы одно значение равно NULL, игнорируются.

Oracle и PostgreSQL

Oracle и PostgreSQL поддерживают синтаксис ANSI SQL для всех функций *REGR_*. Oracle поддерживает также следующую форму для аналитических приложений:

```
REGR_function(dependent, independent) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

MySQL и SQL Server

На этих платформах семейство функций *REGR_* не реализовано.

Пример

В следующем примере демонстрируется, что функция *REGR_COUNT* игнорирует пары, в которых хотя бы одно значение равно NULL. Таблица **test3** содержит три пары, в которых оба элемента отличны от NULL, и три пары, в которых хотя бы один элемент равен NULL.

```
SELECT * FROM test3;
         Y          X
---------- ----------
         1          3
         2          2
         3          1
         4       NULL
      NULL          4
      NULL       NULL
```

Функция *REGR_COUNT* игнорирует пары, в которых есть NULL, и подсчитывает лишь пары, где оба элемента отличны от NULL.

```
SELECT REGR_COUNT(y,x) FROM test3;
REGR_COUNT(Y,X)
---------------
              3
```

Все остальные функции *REGR_* также отбрасывают пары, содержащие хотя бы один NULL, а лишь потом производят вычисления.

STDDEV_POP

Функция *STDDEV_POP* вычисляет *стандартное отклонение* генеральной совокупности для набора числовых значений.

Синтаксис в стандарте ANSI SQL

```
STDDEV_POP(numeric_expression)
```

MySQL, Oracle и PostgreSQL

MySQL, Oracle и PostgreSQL поддерживают синтаксис ANSI SQL. Oracle поддерживает также следующую форму для аналитических приложений:

```
STDDEV_POP(numeric_expression) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

SQL Server

Пользуйтесь функцией *STDEVP*.

Пример

В следующем примере вычисляется стандартное отклонение генеральной совокупности для значений 1, 2 и 3:

```
SELECT * FROM test;
         X
----------
         1
         2
         3

SELECT STDDEV_POP(x) FROM test;
STDDEV_POP(X)
-------------
   .816496581
```

STDDEV_SAMP

Вычисляет *выборочное стандартное отклонение* для набора числовых значений.

Синтаксис в стандарте ANSI SQL

```
STDDEV_SAMP(numeric_expression)
```

MySQL и PostgreSQL

MySQL и PostgreSQL поддерживают синтаксис ANSI SQL.

Oracle

Oracle поддерживает стандартный синтаксис, а также предлагает функцию *STDDEV*, которая работает аналогично *STDDEV_SAMP*, но возвращает 0 (а не NULL) в случае, когда набор содержит только одно значение.

Oracle поддерживает также следующую форму для аналитических приложений:

```
STDDEV_SAMP(numeric_expression) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

SQL Server

Пользуйтесь функцией *STDEV* (с одной буквой *D*!).

Пример

В следующем примере вычисляется выборочное стандартное отклонение для значений 1, 2 и 3:

```
SELECT * FROM test;
         X
----------
         1
         2
         3
```

```
SELECT STDDEV_SAMP(x) FROM test;
STDDEV_SAMP(X)
--------------
             1
```

VAR_POP

Вычисляет дисперсию генеральной совокупности для набора числовых значений.

Синтаксис в стандарте ANSI SQL

```
VAR_POP(numeric_expression)
```

MySQL и PostgreSQL

MySQL и PostgreSQL поддерживают синтаксис ANSI SQL.

Oracle

Oracle поддерживает стандартный синтаксис, а также следующую форму для аналитических приложений:

```
VAR_POP(numeric_expression) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

SQL Server

Пользуйтесь функцией *VARP*.

Пример

В следующем примере вычисляется дисперсия генеральной совокупности для значений 1, 2 и 3:

```
SELECT * FROM test;
         X
----------
         1
         2
         3

SELECT VAR_POP(x) FROM test;
VAR_POP(X)
----------
.666666667
```

VAR_SAMP

Вычисляет выборочную дисперсию для набора числовых значений.

Синтаксис в стандарте ANSI SQL

```
VAR_SAMP(numeric_expression)
```

MySQL и PostgreSQL

MySQL и PostgreSQL поддерживают синтаксис ANSI SQL.

Oracle

Oracle поддерживает стандартный синтаксис, а также предлагает функцию *VARIANCE*, которая работает аналогично *VAR_SAMP*, но возвращает 0 (а не NULL) в случае, когда набор содержит только одно значение:

Oracle поддерживает также следующую форму для аналитических приложений:

```
VAR_SAMP(numeric_expression) OVER (window_clause)
```

О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» ниже.

SQL Server

Пользуйтесь функцией *VAR*.

Пример

В следующем примере вычисляется выборочная дисперсия для значений 1, 2 и 3:

```
SELECT * FROM test;
         X
----------
         1
         2
         3

SELECT VAR_SAMP(x) FROM test;
VAR_SAMP(X)
-----------
          1
```

Оконные функции в ANSI SQL

Стандарт ANSI SQL2003 допускает фразу *window_clause* при вызове агрегатных функций. Тем самым агрегатные функции превращаются в оконные. Синтаксис оконных функций поддерживают Oracle и SQL Server. В этом разделе описываются особенности применения фразы *window_clause* на этих платформах.

В Oracle оконные функции чаще называются *аналитическими*.

Оконные (или аналитические) функции похожи на стандартные агрегатные функции в том смысле, что применяются к нескольким строкам или группам строк, возвращенным в результате выполнения запроса. Однако группы строк, к которым применяются оконные функции, определяются не фразой *GROUP BY*, а фразами *partitioning* и *windowing*. Кроме того, порядок следования строк внутри групп определяется фразой *ordering*, однако это упорядочение влияет только

на способ вычисления функции, но не имеет никакого отношения к порядку, в котором запрос возвращает строки.

Оконные функции должны стоять в конце запроса, после них может находиться только фраза *ORDER BY*. Из-за позднего вычисления оконные функции *нельзя* использовать внутри фраз *WHERE*, *GROUP BY* или *HAVING*.

Синтаксис оконных функций в ANSI SQL2003

В стандарте SQL2003 определен следующий синтаксис оконных функций:

```
FUNCTION_NAME(expr) OVER {window_name | (window_specification)}
window_specification ::= [window_name] [partitioning] [ordering] [framing]
partitioning ::= PARTITION BY value[, value...] [COLLATE collation_name]
ordering ::= ORDER [SIBLINGS] BY rule[, rule...]
rule ::= {value | position | alias} [ASC | DESC] [NULLS {FIRST | LAST}]
framing ::= {ROWS | RANGE} {start | between} [exclusion]
start ::= {UNBOUNDED PRECEDING | unsigned-integer PRECEDING | CURRENT ROW}
between ::= BETWEEN bound AND bound
bound ::= {start | UNBOUNDED FOLLOWING | unsigned-integer FOLLOWING}
exclusion ::= {EXCLUDE CURRENT ROW | EXCLUDE GROUP |
EXCLUDE TIES | EXCLUDE NO OTHERS}
```

Синтаксис оконных функций в Oracle

В Oracle принят следующий синтаксис оконных функций:

```
FUNCTION_NAME(expr) OVER (window_clause)
window_clause ::= [partitioning] [ordering [framing]]
partitioning ::= PARTITION BY value[, value...]
ordering ::= ORDER [SIBLINGS] BY rule[, rule...]
rule ::= {value | position | alias} [ASC | DESC]
[NULLS {FIRST | LAST}]
framing ::= {ROWS | RANGE} {not_range | begin AND end}
not_range ::= {UNBOUNDED PRECEDING |
CURRENT ROW |
value PRECEDING}
begin ::= {UNBOUNDED PRECEDING |
CURRENT ROW |
value {PRECEDING | FOLLOWING}}
end ::= {UNBOUNDED FOLLOWING |
CURRENT ROW |
value {PRECEDING | FOLLOWING}}
```

Синтаксис оконных функций в SQL Server

В SQL Server принят следующий синтаксис оконных функций:

```
FUNCTION_NAME(expr) OVER ([window_clause])
window_clause ::= [partitioning] [ordering]
partitioning ::= PARTITION BY value[, value...]
ordering ::= ORDER BY rule[, rule...]
rule ::= column [ASC | DESC]
```

Разбиение (partitioning)

Разбиение строк, к которым применяется фраза *partitioning*, – это аналог выражения *GROUP BY* в стандартной команде *SELECT*. Фраза *partitioning* принимает список выражений, с помощью которых результирующий набор разбивается на группы. Далее в примерах мы будем работать со следующей таблицей:

```
SELECT * FROM odd_nums;
       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
```

Ниже показан результат разбиения по столбцу **ODD**. Сумма четных чисел равна 2 (0+2), а сумма нечетных – 4 (1+3). Во втором столбце результирующего набора представлена сумма всех значений в той секции, которой принадлежит *данная строка*, но при этом возвращаются все строки таблицы (детальные строки). Следующий запрос генерирует суммарные результаты в контексте детальных строк:

```
SELECT NUM, SUM(NUM) OVER (PARTITION BY ODD) S FROM ODD_NUMS;
      NUM          S
--------- ----------
         0          2
         2          2
         1          4
         3          4
```

Если вообще опустить фразу *partitioning*, то будут суммироваться все числа в столбце **NUM**. Таким образом, весь результирующий набор трактуется как одна большая секция:

```
SELECT NUM, SUM(NUM) OVER ( ) S FROM ODD_NUMS;
      NUM          S
--------- ----------
         0          6
         1          6
         2          6
         3          6
```

Упорядочение (ordering)

Порядок следования строк, к которым применяется аналитическая функция, задается с помощью фразы *ordering*. Однако эта фраза никоим образом не определяет упорядочение строк в результирующем наборе; для этого предназначена фраза *ORDER BY* в конце всего запроса. В следующем примере использования функции *FIRST_VALUE* (Oracle) иллюстрируется эффект различных способов упорядочивания строк в секциях:

```
SELECT NUM,
SUM(NUM) OVER (PARTITION BY ODD) S,
FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM ASC) first_asc,
FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM DESC) first_desc
FROM ODD_NUMS;
```

```
       NUM          S  FIRST_ASC FIRST_DESC
---------- ---------- ---------- ----------
         0          2          0          2
         2          2          0          2
         1          4          1          3
         3          4          1          3
```

Как видите, фраза *ORDER BY* внутри оконной функции влияет на порядок строк в соответствующих секциях на этапе вычисления функции. Фраза *ORDER BY NUM ASC* сортирует секции в порядке возрастания, поэтому первым значением в секции четных чисел оказывается 0, а первым значением в секции нечетных чисел – 1. Фраза *ORDER BY NUM DESC* дает противоположный эффект.

Этот запрос иллюстрирует еще один важный момент: оконные функции позволяют агрегировать и упорядочивать результаты различными способами в рамках одного и того же запроса.

Группировка и скользящие окна (framing)

Многие аналитические функции позволяют также задать виртуальное скользящее окно, окружающее некоторую строку внутри секции; для этого предназначена фраза *framing*. Такие скользящие окна полезны, например, для вычисления промежуточных сумм.

В следующем примере, ориентированном на платформу Oracle, показано применение фразы *framing* в аналитическом варианте функции *SUM* для вычисления промежуточных сумм значений в первом столбце. Поскольку фразы *partitioning* нет, то при каждом вызове *SUM* обрабатывается весь результирующий набор. Однако фраза *ORDER BY* сортирует строки для *SUM* в порядке возрастания значений **NUM**, а фраза *BETWEEN* (соответствует *window_clause*) говорит, что при каждом вызове *SUM* следует включать значения **NUM** только из строк, предшествующих текущей, включая ее саму. При каждом последующем вызове *SUM* в сумму включается на одно значение **NUM** больше, чем при предыдущем, в порядке от наименьшего значения к наибольшему:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS;
      NUM          S
--------- ----------
        0          0
        1          1
        2          3
        3          6
```

В этом примере результирующий набор упорядочен так же, как последовательность промежуточных сумм. Но это вовсе не обязательно. В следующем примере порождаются те же самые результаты, но в другом порядке. Как видите, промежуточные суммы вычислены правильно, однако строки упорядочены по-другому. Упорядоченность результирующего набора совершенно не зависит от порядка, заданного внутри оконной функции:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS
```

```
ORDER BY NUM DESC;
       NUM          S
---------- ----------
         3          6
         2          3
         1          1
         0          0
```

Перечень оконных функций

В стандарте ANSI SQL говорится, что любую агрегатную функцию можно использовать в качестве оконной. Oracle и SQL Server в этом отношении стараются придерживаться стандарта, поэтому почти ко всем агрегатным функциям (к стандартным уж точно) можно применить описанный выше синтаксис оконных функций.

Помимо агрегатных функций в стандарте ANSI SQL определены и другие оконные функции, описанные ниже. Во всех примерах используются следующие данные, которые являются вариацией на тему таблицы **ODD_NUMS**, использованной выше для иллюстрации разбиения, упорядочения и группировки:

```
SELECT * FROM test4;
       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1
```

Оконные функции, реализованные только в Oracle (не имеющие аналогов в SQL Server), перечислены в разделе «Платформо-зависимые расширения» ниже в этой главе.

CUME_DIST

Вычисляет кумулятивное распределение или ранг текущей строки относительно других строк в той же секции. Вычисление производится по формуле:

количество предшествующих или равных строк / общее количество строк в секции

Поскольку результат для данной строки зависит от количества предществующих ей строк в той же секции, при вызове этой функции следует всегда включать фразу *ORDER BY*.

Синтаксис в стандарте ANSI SQL

```
CUME_DIST( ) OVER {window_name|(window_specification)}
```

Oracle

Oracle не разрешает использовать в этой функции фразу *framing*, но требует наличия фразы *ordering*:

```
CUME_DIST( ) OVER ([partitioning] ordering)
```

SQL Server

В SQL Server функция *CUME_DIST* не реализована.

Пример

В следующем примере (для Oracle) функция *CUME_DIST* используется для вычисления относительного ранга каждой строки при условии разбиения на секции по столбцу **ODD** и сортировки по столбцу **NUM**:

```
SELECT NUM, ODD, CUME_DIST( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;
       NUM        ODD   CUMEDIST
---------- ---------- ----------
         0          0 .333333333
         2          0 .666666667
         4          0          1
         1          1        .25
         3          1        .75
         3          1        .75
         5          1          1
```

Ниже объясняется, как вычисляется ранг строки, для которой **NUM**=0:

1. Из-за наличия фразы *ORDER BY* строки в секции располагаются в следующем порядке:
   NUM=0
   NUM=2
   NUM=4
2. Строк, предшествующих той, для которой **NUM**=0, не существует.
3. Существует одна строка, равная той, для которой **NUM**=0, – она сама. Следовательно, делимое равно 1.
4. Всего в секции три строки, то есть делитель равен 3.
5. Результат равен 1/3 или 0.33 в периоде, что и показано в примере.

DENSE_RANK

Присваивает ранг каждой строке в секции в предположении, что строки каким-то образом упорядочены. Ранг строки вычисляется путем подсчета количества предшествующих ей строк и прибавления к результату 1. Строки-дубликаты (в смысле упорядочения, заданного фразой *ORDER BY*) имеют одинаковый ранг. В отличие от функции *RANK*, наличие строк с одинаковым рангом не приводит к пропускам в последовательности вычисленных рангов.

Синтаксис в стандарте ANSI SQL

```
DENSE_RANK( ) OVER {window_name | (window_specification)}
```

Oracle и SQL Server

И Oracle, и SQL Server требуют наличия фразы *ordering* и не допускают фразы *framing*:

```
DENSE_RANK( ) OVER ([partitioning] ordering)
```

Пример

Сравните результаты применения функции *DENSE_RANK* (для Oracle) с теми, что были приведены в разделе, посвященном функции *RANK*:

```
SELECT NUM, DENSE_RANK( ) OVER (ORDER BY NUM) rank
FROM test4;
       NUM       RANK
---------- ----------
         0          1
         1          2
         2          3
         3          4
         3          4
         4          5
         5          6
```

Обе строки, в которых **NUM=3**, имеют ранг 4, а следующая за ними строка – ранг 5. Никаких пропусков в последовательности рангов не образуется, отсюда и название «плотный».

PERCENT_RANK

Вычисляет относительный ранг строки, для чего делит ее ранг минус 1 на количество строк в секции минус 1:

```
(rank - 1) / (rows - 1)
```

Сравните эту формулу с формулой для вычисления *CUME_DIST*.

Синтаксис в стандарте ANSI SQL

```
PERCENT_RANK( ) OVER ({window_name | (window_specification)}
```

Oracle

Oracle требует наличия фразы *ordering* и не допускает фразы *framing*:

```
PERCENT_RANK( ) OVER ([partitioning] ordering)
```

SQL Server

В SQL Server функция *PERCENT_RANK* не реализована.

Пример

В следующем примере (для Oracle) вычисляются относительные ранги значений в столбце **NUM** при условии разбиения на секции по столбцу **ODD**:

```
SELECT NUM, ODD, PERCENT_RANK( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;
       NUM        ODD   CUMEDIST
---------- ---------- ----------
         0          0          0
         2          0         .5
```

```
         4          0          1
         1          1          0
         3          1  333333333
         3          1 .333333333
         5          1          1
```

Ниже объясняется, как вычисляется ранг строки, для которой **NUM**=2:

1. Строка с **NUM**=2 является второй в секции, следовательно, ее ранг равен 2.
2. Вычитаем 1 из 2 и получаем делимое 1.
3. Общее количество строк в секции равно 3.
4. Вычитаем 1 из 3 и получаем делитель 2.
5. Результат равен 1/2 или 0.5, что и показано в примере.

RANK

Присваивает ранг каждой строке в секции в предположении, что строки каким-то образом упорядочены. Ранг строки вычисляется путем подсчета количества предшествующих ей строк и прибавления к результату 1. Строки-дубликаты (в смысле упорядочения, заданного фразой *ORDER BY*) имеют одинаковый ранг, что приводит к появлению пропусков в последовательности вычисленных рангов.

Синтаксис в стандарте ANSI SQL

```
RANK( ) OVER {window_name | (window_specification)}
```

Oracle и SQL Server

И Oracle, и SQL Server требуют наличия фразы *ordering* и не допускают фразы *framing*:

```
RANK( ) OVER ([partitioning] ordering)
```

Пример

В следующем примере (для Oracle) вычисляются ранги значений в столбце **NUM** из таблицы **test4**:

```
SELECT NUM, RANK( ) OVER (ORDER BY NUM) rank
FROM test4;
      NUM       RANK
--------- ----------
        0          1
        1          2
        2          3
        3          4
        3          4
        4          6
        5          7
```

Поскольку обе строки с **NUM**=3 имеют один и тот же ранг (4), то следующая по порядку строка получает ранг 6. Ранг 5 пропускается.

ROW_NUMBER

Присваивает уникальные номера строкам в секции.

Синтаксис в стандарте ANSI SQL

```
ROW_NUMBER( ) OVER ({window_name | (window_specification)}
```

Oracle и SQL Server

И Oracle, и SQL Server требуют наличия фразы *ordering* и не допускают фразы *framing*:

```
ROW_NUMBER( ) OVER ([partitioning] ordering)
```

Пример

```
SELECT NUM, ODD, ROW_NUMBER( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;
NUM               ODD   CUMEDIST
---------- ---------- ----------
         0          0          1
         2          0          2
         4          0          3
         1          1          1
         3          1          2
         3          1          3
         5          1          4
```

Скалярные функции в стандарте ANSI SQL

Скалярная функция возвращает одиночное значение при каждом вызове. В стандарте SQL определено много скалярных функций – для манипулирования датами и временем, строками и числами, а также для получения системной информации, например имени текущего пользователя. В табл. 4.2 перечислены различные категории скалярных функций.

Таблица 4.2. Категории скалярных функций

Категория	Назначение
Встроенные	Выполняют операции над значениями или параметрами, встроенными в базу данных. В Oracle термин «встроенная» применяется для описания всех вообще предоставляемых функций, которые так или иначе «встроены» в СУБД. Такая интерпретация, безусловно, отличается от того, что понимается под встроенными скалярными функциями в этой книге.
CASE и *CAST*	Хотя эти две функции применяются к скалярным входным данным, они помещены в отдельную категорию. Функция *CASE* позволяет реализовать в SQL-командах логику *IF/THEN*, а функция *CAST* предназначена для преобразования типов данных.

Категория	Назначение
Дата и время	Выполняют операции над *темпоральными (время и/или дата)* типами данных или возвращают значения темпорального типа. В стандарте SQL2003 нет функций, которые принимают на входе темпоральные данные и возвращают результат темпорального типа. Наиболее близка к этому функция *EXTRACT* (рассматривается в разделе «Числовые скалярные функции»), которая принимает на входе темпоральные значения, а возвращает число. Функции, которые возвращают темпоральные значения, но не принимают никаких аргументов, рассматриваются в разделе «Встроенные скалярные функции».
Числовые	Выполняют операции над числовыми данными и возвращают числовые значения.
Строковые	Выполняют операции над символьными данными (например, типа *CHAR, VARCHAR, NCHAR, NVARCHAR* или *CLOB*) и возвращают значения строкового или числового типа.

Встроенные скалярные функции

Описанные в стандарте ANSI SQL встроенные скалярные функции служат для идентификации сеанса текущего пользователя и его характеристик, например текущих привилегий сессии. Все встроенные скалярные функции недетерминированы. Встроенные функции *CURRENT_DATE, CURRENT_TIME* и *CURRENT_TIMESTAMP*, приведенные в табл. 4.3, попадают в категорию «дата и время». Хотя все четыре СУБД, обсуждаемые в этой книге, предлагают множество дополнительных встроенных функций, в стандарте SQL определены только те, что перечислены в табл. 4.3.

Таблица 4.3. Встроенные скалярные функции в ANSI SQL

Функция	Назначение
CURRENT_DATE	Возвращает текущую дату
CURRENT_TIME	Возвращает текущее время
CURRENT_TIMESTAMP	Возвращает текущие дату и время
CURRENT_USER или	Возвращает владельца текущего сеанса работы с сервером базы данных
SESSION_USER	Возвращает текущий идентификатор авторизации, если он отличается от идентификатора пользователя
SYSTEM_USER	Возвращает имя текущего пользователя в смысле операционной системы

MySQL

MySQL поддерживает все встроенные скалярные функции ANSI SQL. Дополнительно MySQL предлагает функцию *NOW*, являющуюся синонимом *CURRENT_TIMESTAMP*.

Oracle

Oracle поддерживает функции *USER*, *CURRENT_DATE* и *CURRENT_TIMESTAMP*.

PostgreSQL и SQL Server

PostgreSQL и SQL Server поддерживают все встроенные скалярные функции. Дополнительно PostgreSQL предлагает функцию *NOW*, являющуюся синонимом *CURRENT_TIMESTAMP*.

Примеры

В следующих примерах демонстрируется получение значений, возвращаемых встроенными функциями. Отметим, что разные СУБД возвращают даты в различных форматах:

```
/* MySQL */
SELECT CURRENT_TIMESTAMP;
'2001-12-15 23:50:26'

/* Microsoft SQL Server */
SELECT CURRENT_TIMESTAMP
GO
'Dec 15,2001 23:50:26'

/* Oracle */
SELECT USER FROM dual;
dylan
```

Функции CASE и CAST

В стандарте ANSI SQL описана функция *CASE*, которая позволяет наделить SQL-команды *SELECT* и *UPDATE* логикой *IF/THEN*. В стандарте также определена функция *CAST* для преобразования типов данных. Все рассматриваемые в настоящей книге СУБД поддерживают как *CASE*, так и *CAST*.

CASE

Функция *CASE* позволяет использовать аналог конструкции *IF/THEN/ELSE* в командах *SELECT* и *UPDATE*. Она вычисляет указанные условия и возвращает одно из списка возможных значений.

Функция *CASE* имеет две формы: *простую* и *поисковую*. В простом выражении *CASE* одно входное значение `input_value` сравнивается с несколькими другими значениями и возвращается результат, ассоциированный с первым совпадением. Поисковое выражение *CASE* позволяет проанализировать несколько логических условий и возвращает результат, ассоциированный с первым логическим условием, вычисление которого дало истину.

В любой СУБД поддерживается синтаксис функции *CASE*, определенный в ANSI SQL.

Синтаксис в стандарте ANSI SQL

```
-- Простая операция сравнения
CASE input_value
```

```
WHEN when_condition THEN resulting_value
[... n]
[ELSE else_result_value]
END

-- Булевская поисковая операция
CASE
WHEN Boolean_condition THEN resulting_value
[... n]
[ELSE else_result_expression]
END
```

В простой форме функции *CASE* значение *input_value* сравнивается с выражением в каждой ветви *WHEN*. Возвращается значение *resulting_value*, указанное в первой встретившейся ветви, для которой *input_value = when_condition*. Если ни одно выражение *when_condition* не совпало с *input_value*, то возвращается значение *else_result_value*. Если ветвь *ELSE* опущена, то возвращается NULL.

Структура более сложной поисковой булевской операции примерно такая же, но в каждой ветви *WHEN* находится отдельная операция сравнения.

В любом случае допустимо несколько ветвей *WHEN*, но не более одной ветви *ELSE*.

Примеры

Ниже приведен пример простой операции сравнения, в которой функция *CASE* применяется для представления столбца **contract** в понятном человеку виде:

```
SELECT au_fname,
       au_lname,
       CASE contract
          WHEN 1 THEN 'Yes'
          ELSE 'No'
       END 'contract'
FROM   authors
WHERE  state = 'CA'
```

В следующем более сложном примере поисковая функция *CASE* используется в команде *SELECT* для вычисления того, сколько различных изданий попадают в различные диапазоны суммарного объема продаж с начала года:

```
SELECT CASE
          WHEN ytd_sales IS NULL THEN 'Неизвестно'
          WHEN ytd_sales <= 200 THEN 'Не более 200'
          WHEN ytd_sales <= 1000 THEN 'От 201 до 1000'
          WHEN ytd_sales <= 5000 THEN 'От 1001 до 5000'
          WHEN ytd_sales <= 10000 THEN 'От 5001 до 10000'
          ELSE 'Свыше 10000'
       END 'Продажи с начала года',
       COUNT(*) 'Количество изданий'
FROM   titles
GROUP BY CASE
          WHEN ytd_sales IS NULL THEN 'Неизвестно'
          WHEN ytd_sales <= 200 THEN 'Не более 200'
          WHEN ytd_sales <= 1000 THEN 'От 201 до 1000'
```

```
            WHEN ytd_sales <= 5000 THEN 'От 1001 до 5000'
            WHEN ytd_sales <= 10000 THEN 'От 5001 до 10000'
            ELSE 'Свыше 10000'
         END
ORDER BY MIN( ytd_sales )
```

Этот запрос возвращает такие результаты:

```
Продажи с начала года  Количество изданий
---------------------- -----------------
Неизвестно                             2
Не более 200                           1
От 201 до 1000                         2
От 1001 до 5000                        9
От 5001 до 10000                       1
Свыше 10000                            3
```

Далее приведен пример команды *UPDATE*, которая применяет скидки ко всем книгам. К книгам на тему персональных компьютеров применяется скидка 25%, ко всем остальным – 10%, за исключением тех, для которых с начала года продано более 10000 экземпляров; на последние дается скидка всего 5%. В этом запросе для корректировки цены используется поисковое выражение *CASE*:

```
UPDATE titles
SET    price = price *
       CASE
          WHEN ytd_sales > 10000     THEN 0.95 -- скидка  5%
          WHEN type = 'popular_comp' THEN 0.75 -- скидка 25%
          ELSE 0.9                             -- скидка 10%
       END
WHERE  pub_date IS NOT NULL
```

На этом примере демонстрируется выполнение трех разных операций *UPDATE* в одной команде.

CAST

Функция *CAST* применяется для явного преобразования типа данных выражения. Ее синтаксис, описанный в стандарте ANSI SQL, поддерживают все производители.

Синтаксис в стандарте ANSI SQL

```
CAST(expression AS data_type [(length)])
```

Функция *CAST* преобразует произвольное ***выражение*** (*expression*), например значение столбца или переменной, к другому типу данных. Для типов, характеризуемых длиной (например, *CHAR* или *VARCHAR*), дополнительно задается длина *length*.

Имейте в виду, что некоторые преобразования, например из *DECIMAL* в *INTEGER*, влекут за собой округление. Кроме того, операция преобразования может завершиться с ошибкой, если конечный тип недостаточно широк для хранения преобразованного значения.

Пример

В этом примере числовое значение объема продаж с начала года преобразуется к типу *CHAR* и конкатенируется со строковым литералом и частью названия книги. Столбец **ytd_sales** преобразуется в *CHAR(5)*, а строка в столбце **title** усекается до 30 знаков, чтобы результат оказался более удобен для восприятия:

```
SELECT CAST(ytd_sales AS CHAR(5)) + ' экземпляров ' +
CAST(title AS VARCHAR(30))
FROM titles
WHERE ytd_sales IS NOT NULL
  AND ytd_sales > 10000
ORDER BY ytd_sales DESC
```

Этот запрос возвращает такие результаты:

```
--------------------------------------------------
22246 экземпляров The Gourmet Microwave
18722 экземпляров You Can Combat Computer Stress
15096 экземпляров Fifty Years in Buckingham Pala
```

Числовые скалярные функции

Перечень числовых функций, официально зафиксированных стандартом ANSI SQL, сравнительно невелик, но на различных платформах предлагаются дополнительные математические и статистические функции. MySQL и PostgreSQL поддерживают многие стандартные функции. Другие СУБД предоставляют те же самые возможности с помощью собственных функций, которые, однако, называются не так, как в стандарте. Поддерживаемые числовые функции перечислены в табл. 4.4.

Таблица 4.4. Числовые функции в ANSI SQL

Функция	Назначение
ABS	Возвращает абсолютную величину числа.
BIT_LENGTH	Возвращает длину аргумента в битах.
CEIL или *CEILING*	Округляет дробное число до ближайшего большего целого. Целые числа возвращаются без изменения.
CHAR_LENGTH	Возвращает целое число, равное количеству символов в переданной строке.
EXP	Возводит константу *e* в указанную степень.
EXTRACT	Возвращает указанный компонент (YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, TIMEZONE_MINUTE) темпорального значения.
FLOOR	Округляет дробное число до ближайшего меньшего целого. Целые числа возвращаются без изменения.
LN	Возвращает натуральный логарифм переданного числа.
MOD	Возвращает остаток от деления одного числа на другое.
OCTET_LENGTH	Возвращает целое число, равное количеству октетов (байт) в переданном значении. То же самое, что *BIT_LENGTH / 8*.

Таблица 4.4 (продолжение)

Функция	Назначение
POSITION	Возвращает начальную позицию подстроки в строке.
POWER	Возводит число в степень.
SQRT	Извлекает квадратный корень из числа.
WIDTH_BUCKET	Помещает значение в подходящий интервал из множества интервалов, покрывающих заданный диапазон.

ABS

Стандартная функция *ABS* поддерживается на всех платформах.

Синтаксис в стандарте ANSI SQL

```
ABS( expression )
```

ABS возвращает абсолютное значение числового выражения `expression`.

Пример

В следующем примере демонстрируется применение функции *ABS*:

```
/* SQL2003 */
SELECT ABS(-1) FROM NUMBERS
1
```

BIT_LENGTH, CHAR_LENGTH и OCTET_LENGTH

Поддержка скалярных функций для вычисления длины выражения на всех платформах отклоняется от стандарта ANSI. Но несмотря на это, всюду реализована эквивалентная функциональность, пусть даже под разными именами.

Синтаксис в стандарте ANSI SQL

В ANSI SQL скалярные функции для получения длины значения принимают на входе выражение `expression` и возвращают целое число. Функция *BIT_LENGTH* возвращает количество битов в значении выражения, функция *CHAR_LENGTH* – количество символов в строковом выражении, а функция *OCTET_LENGTH* – количество октетов (байт) в строковом выражении. Если выражение равно NULL, то все три функции возвращают NULL.

```
BIT_LENGTH( expression )
CHAR_LENGTH( expression )
OCTET_LENGTH( expression )
```

MySQL и PostgreSQL

И MySQL, и PostgreSQL поддерживают функции *BIT_LENGTH*, *CHAR_LENGTH*, *OCTET_LENGTH*, а также определенный в ANSI SQL синоним *CHARACTER_LENGTH*.

Oracle

Oracle поддерживает функцию *LENGTHB*, которая возвращает целое число, равное количеству байтов в выражении. Для определения длины выражения в символах Oracle предлагает функцию *LENGTH*, являющуюся синонимом *CHAR_LENGTH*.

SQL Server

В SQL Server имеется функция *LEN*, которая реализует все три варианта.

Пример

В следующем примере показано, как в разных СУБД вычисляется длина строки и значения, хранящегося в столбце:

```
/* MySQL и PostgreSQL */
SELECT CHAR_LENGTH('hello');
SELECT OCTET_LENGTH(book_title) FROM titles;

/* Microsoft SQL Server */
SELECT DATALENGTH(title) FROM titles
WHERE type = 'popular_comp'
GO

/* Oracle */
SELECT LENGTH('HORATIO') "Length of characters"
FROM dual;
```

CEIL

Функция *CEIL* возвращает наименьшее целое число, большее или равное значению входного параметра.

Синтаксис в стандарте ANSI SQL

ANSI SQL поддерживает следующие два варианта:

```
CEIL( expression )
CEILING( expression )
```

MySQL и PostgreSQL

MySQL и PostgreSQL поддерживают как *CEIL*, так и *CEILING*.

Oracle

Oracle поддерживает только *CEIL*.

SQL Server

SQL Server поддерживает только *CEILING*.

Примеры

Если на вход подается положительное, не целое число, то *CEIL* округляет его до ближайшего большего целого:

```
SELECT CEIL(100.1) FROM dual;
CEIL(100.1)
```

```
------------
         101
```

Если же параметр отрицателен, то такое округление «вверх» дает число, которое по абсолютной величине меньше параметра:

```
SELECT CEIL(-100.1) FROM dual;
CEIL(-100.1)
-------------
        -100
```

Функция *FLOOR* ведет себя противоположно *CEIL*.

EXP

Функция *EXP* возвращает значение числа *e* (приблизительно 2.718281), возведенное в указанную степень.

Синтаксис в стандарте ANSI SQL

На всех платформах поддерживается стандартный синтаксис ANSI SQL:

```
EXP( expression )
```

Пример

В следующем примере функция *EXP* применяется для получения приближения к *e*:

```
SELECT EXP(1) FROM dual;
    EXP(1)
----------
2.71828183
```

Обратная к *EXP* функция называется *LN*.

EXTRACT

Эта скалярная функция служит для выделения различных компонентов даты.

Синтаксис в стандарте ANSI SQL

Определенная в стандарте ANSI SQL функция *EXTRACT* принимает параметр `date_part` и выражение `expression`, вычисление которого дает дату. MySQL, Oracle и PostgreSQL поддерживают стандартный синтаксис:

```
EXTRACT( date_part FROM expression )
```

MySQL

В MySQL реализовано несколько больше, чем требуется стандартом. Стандарт ANSI не предусматривает возврата нескольких полей в результате одного обращения к *EXTRACT* (например, *DAY_HOUR*). MySQL же стремится достичь того, что в PostgreSQL реализуется парой функций *DATE_TRUNC* и *DATE_PART*. В MySQL параметр `date_part` может принимать значения, перечисленные в табл. 4.5.

Таблица 4.5. Компоненты даты в MySQL

Компонент	Назначение
MICROSECOND	Микросекунды
SECOND	Секунды
MINUTE	Минуты
HOUR	Час
DAY	День
WEEK	Неделя
MONTH	Месяц
QUARTER	Квартал
YEAR	Год
SECOND_MICROSECOND	Секунды и микросекунды
MINUTE_MICROSECOND	Минуты и микросекунды
MINUTE_SECOND	Минуты и секунды
HOUR_MICROSECOND	Час и микросекунды
HOUR_SECOND	Час, минуты и секунды
HOUR_MINUTE	Час и минуты
DAY_MICROSECOND	День и микросекунды
DAY_SECOND	День, час, минуты и секунды
DAY_MINUTE	День, час и минуты
DAY_HOUR	День и час
YEAR_MONTH	Год и месяц

Oracle

Oracle поддерживает синтаксис ANSI SQL, допустимые компоненты даты перечислены в табл. 4.6.

Таблица 4.6. Компоненты даты в Oracle

Компонент	Назначение
DAY	День месяца (1–31)
HOUR	Час (0–23)
MINUTE	Минуты (0–59)
MONTH	Месяц (1–12)
SECOND	Секунды (0–59)
TIMEZONE_HOUR	Час в смещении часового пояса
TIMEZONE_MINUTE	Минуты в смещении часового пояса

Таблица 4.6 (продолжение)

Компонент	Назначение
TIMEZONE_REGION	Название часового пояса
TIMEZONE_ABBR	Сокращенное название часового пояса
YEAR	Год

PostgreSQL

PostgreSQL поддерживает синтаксис ANSI SQL и добавляет еще несколько компонентов даты (см. табл. 4.7).

Таблица 4.7. Компоненты даты в PostgreSQL

Компонент	Назначение
CENTURY	До версии 8.0 возвращалось значение YEAR, поделенное на 100. Начиная с версии 8.0 возвращается правильный год по григорианскому календарю, согласно которому первый год имеет номер 0001, а столетия с номером 0 не существует (с -1 вы переходите сразу на 1 столетие).
DAY	День месяца (1–31).
DECADE	Поле YEAR, поделенное на 10.
DOW	День недели (0–6, причем воскресенью сопоставляется 0). Работает только для значений типа *TIMESTAMP*.
DOY	Порядковый номер дня в году (1–366). Для годов, не являющихся високосными, максимальное значение равно 365. Работает только для значений типа *TIMESTAMP*.
EPOCH	Количество секунд между началом «эпохи» (1970-01-01 00:00:00-00) и заданным значением. Если переданная в качестве параметра дата предшествует началу эпохи, возвращается отрицательное число.
HOUR	Час (0–23).
MICROSECONDS	Поле SECONDS (включая дробную часть), умноженное на 1 000 000.
MILLENNIUM	До версии 8.0 возвращалось значение YEAR, поделенное на 1000. Начиная с версии 8.0 возвращается правильное значение по григорианскому календарю, согласно которому первое тысячелетие начинается в 0001 году, а столетия с номером 0 не существует. Второе тысячелетие начинается с года 1001, третье – с года 2001.
MILLISECONDS	Поле SECONDS (включая дробную часть), умноженное на 1 000.
MINUTE	Минуты (0–59).
MONTH	Месяц (1–12).
QUARTER	Квартал года (1–4), в который попадает заданное значение. Можно использовать только для значений типа *TIMESTAMP*.

Компонент	Назначение
SECOND	Секунды (0–59).
TIMEZONE	Смещение часового пояса в секундах.
TIMEZONE_HOUR	Час в смещении часового пояса.
TIMEZONE_MINUTE	Минута в смещении часового пояса.
WEEK	Номер недели в году, в которую попадает заданное значение.
YEAR	Год.

SQL Server

SQL Server предоставляет функцию *DATEPART(*`date_part`, `expression`*)* в качестве синонима функции *EXTRACT(*`date_part` *FROM* `expression`*)*, определенной в ANSI SQL. Поддерживаемые SQL Server компоненты даты перечислены в табл. 4.8.

Таблица 4.8. Компоненты даты в SQL Server

Компонент	Назначение
Day	День месяца в выражении типа *DATETIME*. Допустимы также сокращения *d* и *dd*.
dayofyear	Порядковый номер дня в году. Допустимы также сокращения *y* и *dy*.
Hour	Час. Допустимо также сокращение *hh*.
ISO_WEEK	Номер недели согласно стандарту ISO 8601 (1–53). Допустимы также сокращения *isowk* и *isoww*.
microsecond	Микросекунды (0–999999). Допустимо также сокращение *mcs*.
millisecond	Миллисекунды. Допустимо также сокращение *ms*.
minute	Минуты (0–60). Допустимы также сокращения *n* и *mi*.
month	Месяц (1–12). Допустимы также сокращения *m* и *mm*.
nanosecond	Наносекунды (0–999999999). Допустимо также сокращение *ns*.
quarter	Квартал года, в который попадает заданное значение типа *DATETIME*. Допустимы также сокращения *q* и *qq*.
second	Секунды (0–59). Также возможны сокращения *s* и *ss*.
TZoffset	Часовой пояс. Допустимо также сокращение *tz*.
week	Номер недели в году. Допустимы также сокращения *wk* и *ww*.
weekday	День недели. Допустимо также сокращение *dw*.
year	Год. Допустимы также сокращения *yy* и *yyyy*, возвращающие двузначное и четырехзначное значение года соответственно.

Пример

В следующем примере извлекаются различные компоненты даты:

```
/* MySQL */
SELECT EXTRACT(YEAR FROM "2013-07-02");
```

```
2013
SELECT EXTRACT(YEAR_MONTH FROM "2013-07-02 01:02:03");
201307
SELECT EXTRACT(DAY_MINUTE FROM "2013-07-02 01:02:03");
20102

/* PostgreSQL */
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
20
```

FLOOR

Функция *FLOOR* возвращает наибольшее целое число, меньшее или равное значению входного параметра.

Синтаксис в стандарте ANSI SQL

На всех платформах поддерживается следующий синтаксис, определенный в ANSI SQL:

```
FLOOR( expression )
```

Примеры

Если на вход подается положительное число, то *FLOOR* просто отбрасывает все цифры после запятой:

```
SELECT FLOOR(100.1) FROM dual;
FLOOR(100.1)
------------
         100
```

Если же параметр отрицателен, то такое округление «вниз» дает число, которое по абсолютной величине больше параметра:

```
SELECT FLOOR(-100.1) FROM dual;
FLOOR(-100.1)
-------------
        -101
```

Функция *CEIL* ведет себя противоположно *FLOOR*.

LN

Функция *LN* возвращает натуральный логарифм числа, то есть степень, в которую нужно возвести число *e* (математическая постоянная, приблизительно равная 2.718281), чтобы получить заданное значение.

Синтаксис в стандарте ANSI SQL

```
LN( expression )
```

MySQL, Oracle и PostgreSQL

MySQL, Oracle и PostgreSQL поддерживают синтаксис ANSI SQL. На платформах MySQL и PostgreSQL есть также функция LOG, являющаяся синонимом LN.

SQL Server

В SQL Server натуральный логарифм называется *LOG*:

```
LOG( expression )
```

Пример

В следующем примере (для Oracle) показан результат вычисления натурального логарифма числа, близкого к значению *e*:

```
SELECT LN(2.718281) FROM dual;
LN(2.718281)
------------
  .999999695
```

Обратная к *LN* функция называется *EXP*.

MOD

Функция *MOD* возвращает остаток от деления одного числа на другое. На всех платформах поддерживается синтаксис, определенный в стандарте ANSI SQL.

Синтаксис в стандарте ANSI SQL

```
MOD( dividend, divider)
```

Согласно стандарту, функция *MOD* возвращает остаток от деления делимого `dividend` на делитель `divider`. Если делитель равен 0, то возвращается делимое.

Пример

В следующем примере иллюстрируется применение функции *MOD* в команде *SELECT*:

```
SELECT MOD(12, 5) FROM NUMBERS
2
```

POSITION

Функция *POSITION* возвращает номер начальной позиции подстроки в строке.

Синтаксис в стандарте ANSI SQL

```
POSITION( substring IN string )
```

Согласно стандарту, *POSITION* возвращает начало первого вхождения `substring` в `string`. Если подстрока `substring` не встречается в `string`, то возвращается 0. Если хотя бы один аргумент равен NULL, то возвращается NULL.

MySQL и PostgreSQL

MySQL и PostgreSQL поддерживают синтаксис ANSI SQL для функции *POSITION*.

Oracle

В Oracle эквивалентная функция называется *INSTR*.

SQL Server

Вместо *POSITION* SQL Server поддерживает функции *CHARINDEX* и *PATINDEX*. Отличаются они лишь тем, что *PATINDEX* допускает использование метасимволов в критерии поиска.

Примеры

```
/* MySQL */
SELECT LOCATE('bar', 'foobar');
4

/* MySQL и PostgreSQL */
SELECT POSITION('fu' IN 'snafhu');
0

/* Microsoft SQL Server */
SELECT CHARINDEX( 'de', 'abcdefg' )
GO
4

SELECT PATINDEX( '%fg', 'abcdefg' )
GO
6
```

POWER

Функция *POWER* применяется для возведения числа в степень.

Синтаксис в стандарте ANSI SQL

На всех платформах поддерживается синтаксис ANSI SQL:

```
POWER( base, exponent )
```

Функция *POWER* возвращает результат возведения основания *base* в степень *exponent*. Если *base* отрицательно, то *exponent* должно быть целым числом.

Примеры

Возведение в степень положительного числа не вызывает сложностей:

```
SELECT POWER(10,3) FROM dual;
POWER(10,3)
-----------
       1000
```

Результат возведения любого числа в степень 0 равен 1:

```
SELECT POWER(0,0) FROM dual;
POWER(0,0)
----------
         1
```

При возведении в отрицательную степень десятичная запятая сдвигается влево:

```
SELECT POWER(10,-3) FROM dual;
POWER(10,-3)
------------
        .001
```

SQRT

Функция *SQRT* возвращает квадратный корень из числа.

Синтаксис в стандарте ANSI SQL

На всех платформах поддерживается синтаксис ANSI SQL:

```
SQRT( expression )
```

Пример

```
SELECT SQRT(100) FROM dual;
 SQRT(100)
----------
        10
```

WIDTH_BUCKET

Функция *WIDTH_BUCKET* распределяет значения по интервалам (buckets) гистограмы с одинаковой шириной интервалов.

Синтаксис в стандарте ANSI SQL

Ниже параметр *expression* обозначает значение, которое нужно поместить в интервал. Обычно *expression* включает столбцы, отбираемые запросом:

```
WIDTH_BUCKET( expression, min, max, buckets )
```

Параметр *buckets* определяет количество интервалов, создаваемых внутри диапазона, заданного границами *min* и *max*. При этом граница *min* включается в диапазон, а *max* – нет. Значение выражения *expression* попадает в один из интервалов, номер которого и возвращает функция. Если значение *expression* оказывается меньше, чем *min*, то функция возвращает *0*, если *expression* равно либо больше, чем *max*, то функция возвращает *buckets+1*.

MySQL и SQL Server

На платформах MySQL и SQL Server функция *WIDTH_BUCKET* не реализована.

Oracle и PostgreSQL

И Oracle, и PostgreSQL поддерживают синтаксис *WIDTH_BUCKET*, определенный в ANSI SQL.

Примеры

В следующем примере диапазон целочисленных значений от 1 до 10 разбит на два интервала:

```
SELECT x, WIDTH_BUCKET(x,1,10,2)
FROM pivot;
         X WIDTH_BUCKET(X,1,10,2)
---------- ----------------------
         1                      1
         2                      1
         3                      1
```

```
         4                    1
         5                    1
         6                    2
         7                    2
         8                    2
         9                    2
        10                    3
```

А вот более интересный пример. Здесь 11 значений (от 1 до 10) делятся на три интервала и демонстрируется, что левая граница включается в интервал, а правая – нет:

```
SELECT x, WIDTH_BUCKET(x,1,10,3)
FROM pivot;
         X WIDTH_BUCKET(X,1,10,3)
---------- ----------------------
         1                      1
         2                      1
         3                      1
         4                      2
         5                      2
         6                      2
         7                      3
         8                      3
         9                      3
       9.9                      3
        10                      4
```

Особое внимание обратите на строки, где *X=1*, *X=9.9* и *X=10*. Значение, равное *min* (в данном примере 1), попадает в первый интервал, поскольку левая граница первого интервала определена как $x >= min$. Напротив, значение, равное *max*, оказывается *вне* самого правого интервала. В данном примере 10 попадает в интервал переполнения с номером $buckets + 1$. С другой стороны, значение 9.9 попадает в интервал 3, подтверждая, что правая граница этого интервала определена как $x < max$.

Строковые функции и операторы

Основные строковые функции и операторы предоставляют ряд возможностей и возвращают в качестве результата строку. Некоторые функции являются *двухместными*, то есть получают на входе две строки. В табл. 4.9 перечислены строковые функции, определенные в стандарте ANSI SQL.

Таблица 4.9. Строковые функции и операторы SQL

Функция или оператор	Назначение
Оператор конкатенации	Сцепляет два или более строковых выражений, составленных из литералов, значений столбцов или переменных, в одну строку.
CONVERT	Преобразует строку в другое представление, не изменяя кодировки.

Функция или оператор	Назначение
LOWER	Преобразует все символы в строке в нижний регистр.
OVERLAY	Возвращает результат замены одной подстроки другой.
SUBSTRING	Возвращает часть строки.
TRANSLATE	Преобразует строку из одной кодировки в другую.
TRIM	Удаляет начальные и/или конечные символы.
UPPER	Преобразует все символы в строке в верхний регистр.

Оператор конкатенации

В стандарте ANSI SQL определен оператор конкатенации (||), который сцепляет две строки в одну.

MySQL

В MySQL имеется функция *CONCAT*, которая делает то же самое, что оператор конкатенации в ANSI SQL, а оператор || означает логическое *ИЛИ*.

Oracle и PostgreSQL

И Oracle, и PostgreSQL поддерживают определенную в ANSI SQL нотацию оператора конкатенации (||). В Oracle также имеется функция *CONCAT*, являющаяся синонимом этого оператора.

SQL Server

В SQL Server для конкатенации строки используется знак плюс (+). Кроме того, в SQL Server имеется системный параметр *CONCAT_NULL_YIELDS_NULL*, который управляет тем, что происходит, когда одной из конкатенируемых строк является NULL.

Примеры

```
/* Синтаксис ANSI SQL */
'string1' || 'string2' || 'string3'
'string1string2string3'

/* MySQL */
CONCAT('string1', 'string2')
'string1string2'
```

Если хотя бы одно из конкатенируемых значений равно NULL, то и результат равен NULL. Если конкатенируется числовое значение, то оно неявно преобразуется в строку символов.

```
SELECT CONCAT('My ', 'bologna ', 'has ', 'a ', 'first ', 'name...');
'My bologna has a first name...'
SELECT CONCAT('My ', NULL, 'has ', 'first ', 'name...');
NULL
```

CONVERT и TRANSLATE

Функция *CONVERT* изменяет представление символьной строки, не затрагивая текущую кодировку и схему упорядочения (collation). Например, с помощью *CONVERT* можно изменить количество битов на один символ.

Функция *TRANSLATE* изменяет кодировку строки. Так, с помощью *TRANSLATE* можно перевести строку из английской кодировки в кодировку Kanji (японскую) или Cyrillic (русскую). Трансляция должна быть предварительно определена: либо по умолчанию, либо в результате выполнения команды *CREATE TRANSLATION*.

Синтаксис в стандарте ANSI SQL

```
CONVERT(char_value USING conversion_char_name)
TRANSLATE(char_value USING translation_name)
```

Функция *CONVERT* преобразует строку `char_value` в кодировку с именем, заданным параметром `conversion_char_name`. Функция *TRANSLATE* преобразует строку `char_value` в кодировку, определенную в трансляции `translation_name`.

MySQL

MySQL поддерживает синтаксис ANSI SQL для функции *CONVERT*, но не поддерживает функцию *TRANSLATE*.

Oracle

Oracle поддерживает функции *CONVERT* и *TRANSLATE* с той же семантикой, что в стандарте ANSI SQL. Синтаксис Oracle выглядит следующим образом:

```
CONVERT(char_value, target_char_set, source_char_set)
TRANSLATE(char_value USING {CHAR_CS | NCHAR_CS})
```

В реализации Oracle функция *CONVERT* возвращает текст `char_value` в целевой кодировке. `char_value` – это подлежащая преобразованию строка, `target_char_set` – имя кодировки, в которую следует преобразовать строку, а `source_char_set` – имя кодировки, в которой хранится исходная строка.

Функция *TRANSLATE* в Oracle следует синтаксису ANSI, но поддерживает только два значения кодировки: вы можете выбрать либо кодировку СУБД (*CHAR_CS*), либо национальную кодировку (*NCHAR_CS*).

Oracle также поддерживает другую функцию *TRANSLATE*, при вызове которой ключевое слово *USING* опускается. Она не имеет никакого отношения к преобразованию кодировок.

Имена исходной и целевой кодировок можно передавать в виде строковых литералов, переменных или значений, извлеченных из столбцов таблицы. Отметим, что в процессе преобразования в кодировку, где представлены не все символы исходной строки, некоторые символы могут быть заменены специальными символами подстановки.

Oracle поддерживает несколько распространенных кодировок, в том числе *US7ASCII*, *WE8DECDEC*, *WE8HP*, *F7DEC*, *WE8EBCDIC500*, *WE8PC850* и *WE8-ISO8859P1*. Например:

```
SELECT CONVERT('Gro2', 'US7ASCII', 'WE8HP')FROM DUAL;
Gross
```

PostgreSQL

PostgreSQL поддерживает стандартную функцию *CONVERT*, а преобразование можно определить с помощью команды *CREATE CONVERSION*. Функция *TRANSLATE* в PostgreSQL заменяет все вхождения подстроки в строку:

```
TRANSLATE(character_string, from_text, to_text)
```

Вот несколько примеров:

```
SELECT TRANSLATE('12345abcde', '5a', 'XX');
'1234XXbcde'

SELECT TRANSLATE(title, 'computer', 'PC')
FROM titles
WHERE type = 'Personal_computer'

SELECT CONVERT('PostgreSQL' USING iso_8859_1_to_utf_8)
'PostgreSQL'
```

SQL Server

В SQL Server функция *TRANSLATE* не реализована. Функция *CONVERT* в SQL Server – весьма мощное средство, позволяющее изменять тип данных выражения, но в остальном не имеет ничего общего с определением *CONVERT* в стандарте ANSI. Функционально она эквивалентна функции *CAST*:

```
CONVERT (data_type[(length) | (precision, scale)], expression[, style])
```

Параметр *style* служит для определения формата преобразования даты. Дополнительную информацию см. в документации по SQL Server. Пример:

```
SELECT title, CONVERT(char(7), ytd_sales)
FROM titles
ORDER BY title
GO
```

LOWER и UPPER

Функции *LOWER* и *UPPER* позволяют изменить регистр всех символов в строке. Они поддерживаются всеми СУБД, рассматриваемыми в настоящей книге. На отдельных платформах реализованы также дополнительные функции форматирования текста.

Синтаксис в стандарте ANSI SQL

```
LOWER(string)
UPPER(string)
```

Функция *LOWER* преобразует строку *string* в нижний регистр, а функция *UPPER* – в верхний.

MySQL

MySQL поддерживает стандартные функции *UPPER* и *LOWER*, а также их синонимы *UCASE* и *LCASE*.

Oracle, PostgreSQL и SQL Server

На этих платформах поддерживаются стандартные функции *UPPER* и *LOWER*.

Пример

```
SELECT LOWER('You Talkin To ME?'), UPPER('you talking to me?!');
you talkin to me?, YOU TALKING TO ME?!
```

OVERLAY

Функция *OVERLAY* замещает часть исходной строки новой строкой.

Синтаксис в стандарте ANSI SQL

```
OVERLAY(string PLACING embedded_string FROM start
[FOR length])
```

Если хотя бы один аргумент равен NULL, то функция *OVERLAY* возвращает NULL. Строка *embedded_string* заменяет *length* символов в строке *string*, начиная с символа в позиции *start*. Если аргумент *length* не задан, то *embedded_string* заменяет все символы, начиная с позиции *start*.

MySQL, Oracle и SQL Server

На этих платформах функция *OVERLAY* не реализована. Ее можно смоделировать с помощью комбинации функции *SUBSTRING* и оператора конкатенации.

PostgreSQL

PostgreSQL поддерживает функцию *OVERLAY* в том виде, в котором она определена в стандарте ANSI.

Примеры

Вот пример использования функции *OVERLAY*:

```
/* ANSI SQL и PostgreSQL */
SELECT OVERLAY('DONALD DUCK' PLACING 'TRUMP' FROM 8) FROM NAMES;
'DONALD TRUMP'
```

SUBSTRING

Функция *SUBSTRING* возвращает указанную подстроку строки.

Синтаксис в стандарте ANSI SQL

```
SUBSTRING(extraction_string FROM starting_position [FOR length]
[COLLATE collation_name])
```

Если хотя бы один аргумент равен NULL, то *SUBSTRING* возвращает NULL. Параметр *extraction_string* – это исходная строка; она может быть задана в виде литерала, столбца, имеющего один из симольных типов, или переменной символьного типа. Параметр *starting_position* – целое число, задающее начальную позицию подстроки. Необязательный параметр *length* – целое число, равное количеству символов в подстроке. Если ключевое слово *FOR* опущено, то возвращается

подстрока, начинающаяся в позиции *starting_position* и продолжающаяся до конца строки *extraction_string*.

MySQL

MySQL в основном поддерживает стандарт ANSI, но не позволяет задавать фразу *COLLATE*. Синтаксис выглядит следующим образом:

```
SUBSTRING(extraction_string [FROM starting_position] [FOR length])
```

Oracle

В Oracle функция *SUBSTR* очень близка к стандартной функции *SUBSTRING*, но не поддерживает фразу *COLLATE*. Если значение *starting_position* отрицательно, то Oracle отсчитывает позицию от конца строки *extraction_string*. Если аргумент *length* опущен, то возвращается подстрока от начальной позиции до конца строки. Синтаксис выглядит следующим образом:

```
SUBSTR(extraction_string, starting_position[, length])
```

PostgreSQL

PostgreSQL в основном следует стандарту ANSI, но не поддерживает фразу *COLLATE*. Синтаксис выглядит следующим образом:

```
SUBSTRING(extraction_string [FROM starting_position] [FOR length])
```

SQL Server

В SQL Server реализация близка к стандарту ANSI, но фраза *COLLATE* не поддерживается. SQL Server позволяет применять эту функцию к тексту, а также к данным типа image и binary, однако параметры *starting_position* и *length* обозначают количество байтов, а не символов. Синтаксис выглядит следующим образом:

```
SUBSTRING(extraction_string [FROM starting_position] [FOR length])
```

Примеры

Приведенные ниже примеры относятся ко всем четырем рассматриваемым платформам. Лишь второй пример, в котором начальная позиция отрицательна, работает только в Oracle (напомним, что в этой СУБД функция *SUBSTRING* называется *SUBSTR*), а на всех остальных платформах завершается с ошибкой.

```
/* Oracle, позиция отсчитывается слева */
SELECT SUBSTR('ABCDEFG',3,4) FROM DUAL;
'CDEF'

/* Oracle, позиция отсчитывается справа */
SELECT SUBSTR('ABCDEFG',-5,4) FROM DUAL;
'CDEF'

/* MySQL */
SELECT SUBSTRING('Be vewy, vewy quiet' FROM 5);
'wy, vewy quiet'

/* PostgreSQL и SQL Server */
SELECT au_lname, SUBSTRING(au_fname, 1, 1)
```

```
FROM authors
WHERE au_lname = 'Carson'
Carson C
```

TRIM

Функция *TRIM* удаляет символы в начале, конце или с обеих сторон символьной строки или *BLOB*-значения. По умолчанию удаляются все вхождения указанного символа с обеих сторон строки. Если удаляемый символ не указан, то *TRIM* удаляет пробелы.

Синтаксис в стандарте ANSI SQL

```
TRIM( [ [{LEADING | TRAILING | BOTH}] [removal_char] FROM ]
target_string
[COLLATE collation_name] )
```

removal_char – подлежащий удалению символ, а *target_string* – строка, из которой нужно удалить символы. Если аргумент *removal_char* не задан, то *TRIM* удаляет пробелы. Фраза *COLLATE* говорит, что результат функции следует преобразовать в указанную кодировку (которая должна быть определена).

MySQL, Oracle и PostgreSQL

На этих платформах поддерживается стандартный синтаксис функции *TRIM*.

SQL Server

В SQL Server имеются функции *LTRIM* и *RTRIM*, которые удаляют начальные и конечные пробелы соответственно. Никакие другие символы, кроме пробелов, эти функции удалить не позволяют.

Примеры

```
SELECT TRIM(' wamalamadingdong ');
'wamalamadingdong'

SELECT LTRIM( RTRIM(' wamalamadingdong ') );
'wamalamadingdong'

SELECT TRIM(LEADING '19' FROM '1976 AMC GREMLIN');
'76 AMC GREMLIN'

SELECT TRIM(BOTH 'x' FROM 'xxxWHISKEYxxx');
'WHISKEY'

SELECT TRIM(TRAILING 'snack' FROM 'scooby snack');
'scooby '
```

Платформо-зависимые расширения

В следующих разделах приводится полный перечень функций, поддерживаемых в каждой СУБД. Эти функции платформо-зависимы. Так, не гарантируется, что функция, реализованная на платформе MySQL, будет поддержана другими производителями.

Функции, поддерживаемые MySQL

В этом разделе приведен алфавитный перечень функций, поддерживаемых на платформе MySQL, – с примерами применения и результатами выполнения.

ACOS (number)

Возвращает арккосинус числа *number*, находящегося в диапазоне от -1 до 1. Результат принимает значения от 0 до π и выражен в радианах. Например:

```
SELECT ACOS( 0 ) -> 1.570796
```

ADDDATE (date, days)

Возвращает результат прибавления *days* дней к дате *date*. Например:

```
SELECT ADDDATE('2008-05-14', 1) -> '2008-05-15'
```

ADDDATE(date, INTERVAL expr unit)

Возвращает результат прибавления *expr* единиц времени *unit* к дате *date*. Например:

```
SELECT ADDDATE('2008-05-14', INTERVAL 1 DAY) -> '2008-05-15'
```

AES_DECRYPT(crypt_str, key_str)

Возвращает результат дешифрирования строки *crypt_str*, зашифрованной шифром AES с ключом *key_str*.

AES_ENCRYPT(str, key_str)

Возвращает результат шифрования строки *str* шифром AES с ключом *key_str*. Например:

```
SELECT AES_DECRYPT(AES_ENCRYPT('secret sauce', 'password'), 'password') -> 'secret
sauce'
```

ASCII(text)

Возвращает ASCII-код первого символа строки *text*. Например:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Возвращает арксинус числа *number*, находящегося в диапазоне от -1 до 1. Результат принимает значения от -π/2 до π/2 и выражен в радианах. Например:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(number)

Возвращает арктангенс произвольного числа *number*. Результат принимает значения от -π/2 до π/2 и выражен в радианах. Например:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATAN2(x, y)

Возвращает арктангенс для двух переменных x и y. Функция *ATAN2(x, y)* аналогична *ATAN(y/x)* с тем отличием, что для определения квадранта, в котором находится результат, используются знаки аргументов *x* и *y*. Например:

```
ATAN2(3.1415, 1) -> 1.262619
```

BENCHMARK(count, expr)

Вычисляет выражение *expr count* раз. Результат всегда равен 0. Например:

```
BENCHMARK(1000000,ATAN2(3.1415, 1)) -> 0
```

BIN(number)

Возвращает строку, содержащую двоичное значение числа *number* типа *BIGINT*.

BINARY(string)

Преобразует *string* в двоичную строку.

BIT_AND(expr)

Агрегатная функция, возвращающая результат применения поразрядной операции И ко всем битам в *expr*. Вычисление выполняется с 64-битовой (*BIGINT*) точностью. Если подходящих строк не найдено, возвращается -1. Например:

```
BIT_AND(mycolumn) -> 0
```

BIT_COUNT(number)

Возвращает количество единичных битов в числе *number*. Например:

```
BIT_COUNT(5) -> 2
```

BIT_OR(expr)

Агрегатная функция, возвращающая результат применения поразрядной операции ИЛИ ко всем битам в *expr*. Вычисление выполняется с 64-битовой (*BIGINT*) точностью. Если подходящих строк не найдено, возвращается 0. Например:

```
BIT_OR(mycolumn) -> 1
```

BIT_XOR(expr)

Агрегатная функция, возвращающая результат применения поразрядной операции ИСКЛЮЧАЮЩЕЕ ИЛИ ко всем битам в *expr*. Вычисление выполняется с 64-битовой (*BIGINT*) точностью. Если подходящих строк не найдено, возвращается 0. Например:

```
BIT_XOR(mycolumn) -> 1
```

CHAR(number[, ...])

Возвращает строку, состоящую из символов, заданных своими ASCII-кодами. Аргументы со значением NULL игнорируются. Например:

```
CHAR(120,121,122) -> 'xyz'
```

CHARSET(str)

Возвращает кодировку строки *str*. Например:

```
CHARSET('oolong') -> 'latin1'
```

COALESCE(list)

Возвращает первый отличный от NULL элемент списка. Например:

```
COALESCE( NULL, 1, 2 ) -> 1
```

COERCIBILITY(expr)

Возвращает показатель приводимости схемы упорядочения значения *expr*. Результатом является целое число от 0 до 5; чем меньше значение, тем выше его приоритет. Например:

```
COERCIBILITY( 'darjeeling' ) -> 4
```

COLLATION(str)

Возвращает схему упорядочения строки *str*. Например:

```
COLLATION( _utf8'assam' ) -> 'utf8_general_ci'
```

COMPRESS(string)

Возвращает результат упаковки строки *string*.

CONCAT_WS(separator, str1, str2[,...])

Специальная форма функции *CONCAT*, которая вставляет разделитель *separator* между соседними конкатенируемыми строками. Если аргумент *separator* равен NULL, то функция возвращает NULL. Например:

```
CONCAT_WS(', ', au_lname, au_fname ) -> 'Jefferson, Thomas'
```

CONNECTION_ID( )

Возвращает идентификатор соединения. У каждого соединения имеется уникальный идентификатор. Например:

```
CONNECTION_ID( ) -> 305102
```

CONV(number, from_base, to_base)

Возвращает строковое представление числа *number* после перевода из системы счисления с основанием *from_base* в систему с основанием *to_base*. Если хотя бы один аргумент равен NULL, функция возвращает NULL. Например:

```
CONV(12,10,2) -> 1100
```

COS(number)

Возвращает косинус числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT COS(0) -> 1.000000
```

COT(number)

Возвращает котангенс числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT COT(0) -> NULL
```

CRC32(expr)

Возвращает контрольную сумму CRC32 выражения *expr*. Например:

```
SELECT CRC32('mysql') -> 2501908538
```

CURDATE( )

Возвращает текущую дату в формате YYYY-MM-DD или YYYYMMDD, в зависимости от того, вызвана функция в строковом или числовом контексте. Например:

```
CURDATE( ) -> '2003-06-24'
```

CURTIME( )

Возвращает текущее время в формате HH:MM:SS или HHMMSS, в зависимости от того, вызвана функция в строковом или числовом контексте. Например:

```
CURTIME( ) -> '20:40:20'
```

DATABASE( )

Возвращает имя текущей базы данных. Например:

```
DATABASE( ) -> 'PUBS'
```

DATE_ADD(date, INTERVAL expr type)
DATE_SUB(date, INTERVAL expr type)
ADDDATE(date, INTERVAL expr type)
SUBDATE(date, INTERVAL expr type)

Эти функции производят арифметические вычисления с датами. *ADDATE* и *SUBDATE* – синонимы соответственно *DATE_ADD* и *DATE_SUB*. Функция *DATE_ADD* возвращает результат сложения даты *date* с интервалом *INTERVAL*, а *DATE_SUB* – результат вычитания интервала *INTERVAL* из даты *date*. Например:

```
DATE_ADD('1999-04-15', INTERVAL 1 DAY) -> '1999-04-16'
DATE_SUB('1999-04-15', INTERVAL 1 DAY) -> '1999-04-14'
```

DATE_ FORMAT(date, format)

Форматирует дату в соответствии с форматной строкой *format*. Например:

```
DATE_FORMAT('2008-04-15', '%M-%D-%Y') -> 'April-15th-2008'
```

В табл. 4.10 описаны допустимые спецификаторы формата.

Таблица 4.10. Спецификаторы формата в MySQL

Спецификатор формата	Назначение
%a	Сокращенное название дня (Sun–Sat)
%b	Сокращенное название месяца (Jan–Dec)
%c	Номер месяца (1–12)
%D	День месяца с окончанием (1st, 2nd, 3rd...)
%d	День месяца, записанный двумя цифрами (01, 02, ...)
%e	День месяца (1, 2, ...)
%H	Час (00–23)
%h	Час (01–12)

Спецификатор формата	Назначение
%I	Час (01–12)
%i	Минута (00–59)
%j	Номер дня в году (001–366)
%k	Час (0–23)
%l	Час (1–12)
%M	Полное название месяца (January–December)
%m	Месяц (01–12)
%p	AM или PM
%r	Время в 12-часовом формате (hh:mm:ss AM или PM)
%S или *%s*	Секунды (00–59)
%T	Время в 24-часовом формате (hh:mm:ss)
%U	Номер недели в году (00–53, первым днем недели считается воскресенье)
%u	Номер недели в году (00–53, первым днем недели считается понедельник)
%V	Номер недели в году (01–53, первым днем недели считается воскресенье)
%v	Номер недели в году (01–53, первым днем недели считается понедельник)
%W	Название дня недели (Sunday–Saturday)
%w	Номер дня недели (0–6, где 0 соответствует воскресенью, а 6 – субботе)
%X	Четырехзначный номер года в случае, когда первым днем недели считается воскресенье
%x	Четырехзначный номер года в случае, когда первым днем недели считается понедельник
%Y	Год, записанный четырьмя цифрами
%y	Год, записанный двумя цифрами
%%	Литерал "%"

DAYNAME(`date`)

Возвращает название дня недели, приходящегося на дату *`date`*. Например:

```
DAYNAME('1999-04-15') -> 'Thursday'
```

DAYOFMONTH(`date`)

Возвращает порядковый номер дня в месяце (число от 1 до 31) для даты *date*. Например:

```
DAYOFMONTH('1999-04-15') -> 15
```

DAYOFWEEK(date)

Возвращает номер дня недели для даты *date* (1 = воскресенье, 2 = понедельник, ... 7 = суббота). Например:

```
DAYOFWEEK('1999-04-15') -> 5
```

DAYOFYEAR(date)

Возвращает номер дня в году для даты *date* (число от 1 до 366). Например:

```
DAYOFYEAR('1999-04-15') -> 105
```

DECODE(crypt_str, pass_str)

Дешифрирует строку *crypt_str*, используя *pass_str* в качестве пароля; строка *crypt_str* должна быть получена от функции *ENCODE*. Например:

```
DECODE(ENCODE('foo','bar'),'bar') -> 'foo'
```

DEFAULT(column_name)

Возвращает значение по умолчанию для столбца *column_name*. Например:

```
SELECT DEFAULT(a_column) FROM a_table -> 0
```

DEGREES(number)

Возвращает результат преобразования аргумента *number* из радианов в градусы. Например:

```
DEGREES(3.1415926) -> 179.99999692953
```

DES_DECRYPT(crypt_str, key_str)

Возвращает результат дешифрирования строки *crypt_str*, зашифрованной шифром DES с ключом *key_str*.

DES_ENCRYPT(str, key_str)

Возвращает результат шифрования строки *str* шифром DES с ключом *key_str*. Например:

```
SELECT DES_DECRYPT(DES_ENCRYPT('secret sauce', 'password'), 'password') -> 'secret
sauce'
```

ELT(n, str1, str2, str3[, ... n])

Возвращает *str1*, если *n* = 1, *str2*, если *n* = 2, и т. д. Если *n* меньше 1 или больше количества аргументов, то функция возвращает NULL. Функция *ELT* служит дополнением к функции *FIELD*. Например:

```
ELT(1, 'Hi', 'There') -> 'Hi'ELT(2, 'Hi', 'There') -> 'There'
```

ENCODE(str, pass_str)

Шифрует строку *str*, используя в качестве пароля строку *pass_str*. Для дешифрирования результата используется функция *DECODE*. Результат представляет собой двоичную строку той же длины, что исходная. Например:

```
DECODE(ENCODE('foo','bar'),'bar') -> 'foo'
```

ENCRYPT(str[, salt])

Шифрует строку *str*, применяя системный вызов UNIX *crypt*. В качестве аргумента *salt* должна быть передана строка из двух символов. Например:

```
ENCRYPT('password') -> 'ZB7yqPUHvNnmo'
```

EXPORT_SET(bits, on, off,[separator,[number_of_bits]])

Возвращает строку *str*, в которой вместо каждого единичного бита в числе *bits* подставляется строка *on*, а вместо каждого нулевого бита – строка *off*. Строки отделяются друг от друга разделителем *separator* (по умолчанию запятая (,)). Параметр *number_of_bits* (количество битов) необязателен; если он опущен, подразумевается значение *64*. Например:

```
EXPORT_SET(4,'T','F')
F,F,T,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,
F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F
```

EXTRACTVALUE(xml, xpath)

Извлекает текст из фрагмента XML, заданного параметром *xml*, соответствующий XPath-пути *xpath*. Например:

```
EXPORT_SET('<catalog><tea>oolong</tea><tea>darjleeing</tea></catalog>',
           '/catalog/tea')
'oolong darjeeling'
```

FIELD(str, str1, str2, str3[,...])

Возвращает индекс строки *str* в заданном наборе аргументов или 0, если строка *str* не встречается. Функция *FIELD* служит дополнением к функции *ELT*. Например:

```
FIELD('GOOSE','DUCK','DUCK','GOOSE','DUCK') -> 3
```

FIND_IN_SET(str, strlist)

Возвращает индекс строки *str* в списке строк *strlist*, разделенных запятыми. Эта функция эквивалентна обращению *FIELD(str, CONCAT_WS(', ', str1, str2, str3[, ...]))*. Например:

```
FIND_IN_SET('b','a,b,c,d') -> 2
```

FORMAT(number, decimals)

Представляет число *number* в формате #,###,###.## после округления до *decimals* знаков после запятой. Если *decimals* равно 0, то в результате не будет ни десятичной точки, ни дробной части. Например:

```
FORMAT(12345.2132,2) -> 12,345.21
FORMAT(12345.2132,0) -> 12,345
```

FOUND_ROWS( )

Возвращает количество строк, которое вернул бы ранее выполненный запрос, содержащий фразу *LIMIT*. Обращаться к функции *FOUND_ROWS( )* следует сразу же после выполнения запроса с ограничением на число возвращаемых строк. Например:

```
SELECT FOUND_ROWS( ) -> 31415926
```

FROM_DAYS(number)

Зная номер дня (начало отсчета – первый год нашей эры) *number*, возвращает значение типа *DATE*. Эту функцию не следует использовать для дат, предшествующих принятию григорианского календаря (1582 год), из-за того, что в результате перехода на него несколько дней было потеряно. Например:

```
FROM_DAYS(888888) -> 2433-09-10
```

FROM_UNIXTIME(unix_timestamp)

Возвращает представление аргумента *unix_timestamp* (количество секунд, прошедших с 1 января 1970 года) в формате YYYYMM-DD HH:MM:SS или YYYYMMDDHHMMSS в зависимости от того, вызывается функция в строковом или числовом контексте. Например:

```
FROM_UNIXTIME(888123892) -> 1998-02-21 21:04:52
```

FROM_UNIXTIME(unix_timestamp, format)

Возвращает представление аргумента *unix_timestamp* (количество секунд, прошедших с 1 января 1970 года) в формате, определяемом форматной строкой *format*. Форматная строка может содержать те же символы, что перечислены в описании функции *DATE_FORMAT*. Например:

```
FROM_UNIXTIME(888123892,'%Y %D %M') -> '1998 21st February'
```

GET_LOCK(str, timeout)

Пытается получить блокировку с именем *str*, ожидая не более *timeout* секунд. Возвращает 1, если блокировку удалось получить, и NULL – в случае ошибки или тайм-аута. Например:

```
GET_LOCK('lochness',10) -> 1
```

GREATEST(x, y[, ...])

Возвращает максимум из значений аргументов. Например:

```
GREATEST(8,2,4) -> 8
```

GROUP_CONCAT([DISTINCT] expr [ORDER BY order [ASC | DESC]] [SEPARATOR sep])

Возвращает результат конкатенации отличных от NULL значений в группе, определяемой выражением *expr*. Параметр *order* задает сортировку внутри группы, а *sep* – разделитель, вставляемый между конкатенированными значениями. Например:

```
SELECT estate, GROUP_CONCAT(tea SEPARATOR ';') FROM catalog GROUP BY estate
```

HEX(number)

Возвращает строковое представление шестнадцатеричного значения *number*. Эквивалентно вызову *CONV(number, 10, 16)*. Например:

```
HEX(255) -> FF
```

HOUR(time)

Возвращает число, представляющее час указанного времени *time* (от 0 до 23). Например:

```
HOUR('08:20:15') -> 8
```

IF(expr1, expr2, expr3)

Если *expr1* равно *TRUE*, то возвращает *expr2*, иначе *expr3*. Например:

```
IF(1,'yes','no') -> 'yes'
IF(0,'yes','no') -> 'no'
```

IFNULL(expr1, expr2)

Если *expr1* отлично от NULL, то возвращает *expr1*, иначе *expr2*. Например:

```
IFNULL(0,'NULL') -> 0
IFNULL(NULL,'NULL') -> 'NULL'
```

INET_ATON(expr)

Возвращает числовое представление сетевого IP-адреса *expr*. Например:

```
INET_ATON('127.0.0.1') -> 2130706433
```

INET_NTOA(num)

Возвращает строковое представление сетевого IP-адреса, записанного в виде числа *num*. Например:

```
INET_NTOA(2130706433) -> '127.0.0.1'
```

INSERT(str, pos, len, newstr)

Возвращает строку *str*, в которой *len* символов, начиная с позиции *pos*, заменены строкой *newstr*. Например:

```
INSERT('paper',2,3,'ea') -> 'pear'
```

INSTR(str, substr)

Возвращает позицию первого вхождения подстроки *substr* в строке *str*. Например:

```
INSTR('ducks','c') -> 3
```

INTERVAL(num1, num2, num3, num4[, ... n])

Возвращает *0*, если *num1* < *num2*; 1, если *num1* < *num3* и т. д. Требуется, чтобы выполнялось условие *num2* < *num3* < *num4* < ... < *numN*. Например:

```
INTERVAL(5,1,6) -> 1
INTERVAL(5,2,3,7,9) -> 2
```

IS_FREE_LOCK(lock)

Возвращает *1*, если блокировка *lock* свободна, и 0, если она занята. В случае ошибки функция может возвращать NULL. Например:

```
IS_FREE_LOCK('lochness') -> 0
```

IS_USED_LOCK(lock)

Возвращает идентификатор соединения, в котором захвачена блокировка с идентификатором *lock*, и NULL, если эта блокировка свободна. Например:

```
IS_USED_LOCK('lochness') -> 0
```

ISNULL(expr)

Если *expr* равно NULL, то возвращает 1, иначе 0. Например:

```
ISNULL(1) -> 0
ISNULL(NULL) -> 1
```

LAST_DAY(expr)

Возвращает дату последнего дня в месяце, которому принадлежит дата *expr*. Например:

```
LAST_DAY('2012-01-01') -> '2012-01-31'
```

LAST_INSERT_ID([expr])

Возвращает последнее автоматически сгенерированное значение, которое было вставлено в столбец с атрибутом *AUTO_INCREMENT*. Например:

```
LAST_INSERT_ID( ) -> 0
```

LCASE(str)

Синоним *lower(str)*. Например:

```
LCASE('DUCK') -> 'duck'
```

LEAST(X, Y[, ... n])

Возвращает минимум из значений аргументов. Например:

```
LEAST(10,5,3,7) -> 3
```

LEFT(str, len)

Возвращает *len* самых левых символов из строки *str*. Например:

```
LEFT('Ducks', 4) -> 'Duck'
```

LENGTH(str)

Возвращает длину строки *str*. Например:

```
LENGTH('DUCK') -> 4
```

LOAD_FILE(file_name)

Читает файл с именем *file_name* и возвращает его содержимое в виде строки. Файл должен находиться на сервере, а пользователь должен задать полное имя к файлу и иметь права доступа к нему.

LOCATE(substr, str), POSITION(substr IN str)

Возвращает позицию первого вхождения подстроки *substr* в строку *str* или 0, если *substr* не встречается в *str*. *LOCATE* является синонимом стандартной функции *POSITION(substr IN str)*. Например:

```
LOCATE('al','Donald') -> 4
POSITION('al' IN 'Donald') -> 4
```

LOCATE(substr, str, pos)

Возвращает позицию первого вхождения подстроки *substr* в строку *str*, начиная с позиции *pos*, или 0, если *substr* не встречается в *str*. Например:

```
LOCATE('World', 'Hello, World!') -> 8
```

LOG(X)

Возвращает натуральный логарифм *X*. Например:

```
LOG(50) -> 3.912023
```

LOG2(X)

Возвращает двоичный логарифм *X*. Например:

```
LOG2(50) -> 5.64386
```

LOG10(X)

Возвращает десятичный логарифм *X*. Например:

```
LOG10(50) -> 1.698970
```

LPAD(str, len, padstr)

Возвращает строку *str*, дополненную слева строкой *padstr* до длины *len* символов. Например:

```
LPAD('ucks',6,'d') -> 'dducks'
```

LTRIM(str)

Возвращает строку *str*, из которой удалены начальные пробелы. Например:

```
LTRIM(' Howdy! ') -> 'Howdy! '
```

MAKE_SET(bits, str1, str2[, ...n])

Возвращает множество *set* (строку, состоящую из подстрок, разделенных запятыми), содержащее те из аргументов *strX*, которым соответствует единичный бит в битовом векторе *bits*; строке *str1* соответствует бит 0, строке *str2* – бит 1 и т. д. Те из аргументов *str1*, *str2*, ..., которые равны NULL, не добавляются к результату. Например:

```
MAKE_SET(1 | 4,'hello','nice','world') -> 'hello,world'
```

MAKEDATE(y, n)

Возвращает дату, соответствующую году *y* и дню года с порядковым номером *n*. Например:

```
MAKEDATE(2008, 1) -> '2008-01-01'
```

MAKETIME(hour, minute, second)

Возвращает время, соответствующее заданным часу, минуте и секунде. Например:

```
MAKETIME(2, 30, 0) -> '02:30:00'
```

MATCH(col1, col2,...) AGAINST (expr [search_modifier])

Выполняет полнотекстовый поиск выражения *expr* в указанных столбцах. Дополнительную информацию о полнотекстовом поиске см. в документации по MySQL.

MD5(string)

Вычисляет для строки *string* контрольную сумму по алгоритму MD5. Результатом является шестнадцатеричное число из 32 цифр. Например:

```
MD5('somestring') -> 1f129c42de5e4f043cbd88ff6360486f
```

MICROSECOND(time)

Возвращает число, представляющее микросекунды указанного времени *time* (от 0 до 999999). Например:

```
MICROSECOND('08:20:15.000050') -> 50
```

MID(str, pos, len)

Синоним *SUBSTRING(str, pos, len)*

MINUTE(time)

Возвращает число, представляющее минуты указанного времени *time* (от 0 до 59). Например:

```
MINUTE('08:20:15') -> 20
```

MONTH(date)

Возвращает число, представляющее месяц указанной даты *date* (от 1 до 12). Например:

```
MONTH('1999-04-15') -> 4
```

MONTHNAME(date)

Возвращает название месяца для даты *date*. Например:

```
MONTHNAME('1999-04-15') -> 'April'
```

NOW( ), SYSDATE( )

Возвращает текущие дату и время в формате YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS в зависимости от того, вызвана ли функция в строковом или числовом контексте. Например:

```
NOW( ) -> 2003-06-24 20:40:24
SYSDATE( ) -> 2003-06-24 20:40:24
CURRENT_TIMESTAMP -> 2003-06-24 20:40:24
```

NULLIF(expr1, expr2)

Возвращает NULL, если *expr1* равно *expr2*, в противном случае *expr1*. Например:

```
NULLIF(2,29) -> 2
NULLIF(29,29) -> NULL
```

OCT(n)

Возвращает число *n*, записанное в восьмеричном виде. Эквивалентно обращению *CONV(N,10,8)*. Если *n* равно NULL, возвращает NULL. Например:

```
OCT(255) -> 377
```

OLD_PASSWORD(str)

Вычисляет свертку пароля, зная открытый пароль *str*. Эта функция используется для шифрования паролей доступа к MySQL. Префикс «OLD_» был добавлен в версии 4.1, когда схема хеширования паролей изменилась в целях повышения безопасности. Например:

```
OLD_PASSWORD('password') -> '5d2e19393cc5ef67'
```

ORD(str)

Возвращает ординальное число многобайтного символа, представленного строкой *str*. Вычисление производится по формуле: (ASCII-код первого байта) * 256 + (ASCII-код второго байта) * 256 * 256) + (ASCII-код третьего байта) * 256 * 256 * 256[,...]. Если *str* – не многобайтный символ, то эта функция возвращает то же значение, что и функция *ASCII*. Например:

```
ORD('29') -> 50
```

PASSWORD(str)

Вычисляет свертку пароля, зная открытый пароль *str*. Эта функция используется для шифрования паролей доступа к MySQL. Например:

```
PASSWORD('password') -> '5d2e19393cc5ef67'
```

PERIOD_ADD(period, months)

Прибавляет *months* месяцев к периоду *period* (заданному в формате YYMM или YYYYMM). Возвращает значение в формате YYYYMM. Например:

```
PERIOD_ADD(9902,3) -> 199905
```

PERIOD_DIFF(period1, period2)

Возвращает количество месяцев между двумя периодами: *period1* и *period2* (оба должны быть представлены в формате YYMM или YYYYMM). Например:

```
PERIOD_DIFF(9902,9905) -> -3
```

PI( )

Возвращает значение числа π. Например:

```
PI( ) -> 3.141593
```

POW(X, Y), POWER(X, Y)

Возвращает значение *X*, возведенное в степень *Y*. Например:

```
POW(2, 8) -> 256.000000
```

QUARTER(date)

Возвращает квартал, соответствующий дате *date* (число от 1 до 4). Например:

```
QUARTER('1999-04-15') -> 2
```

QUOTE(str)

Возвращает строку *str*, в которой специальные символы экранированы так, чтобы значение можно было безопасно использовать в командах SQL. Например:

```
QUOTE('\'start and end with quote\'') -> '\'start and end with quote\''
```

RADIANS(X)

Возвращает результат преобразования аргумента *X* из градусов в радианы. Например:

```
RADIANS(180) -> 3.1415926535898
```

RAND( ), RAND(N)

Возвращает случайное число с плавающей точкой в диапазоне от 0 до 1.0. Если задан целочисленный аргумент *N*, то он используется для инициализации генератора псевдослучайных чисел. Например:

```
RAND( ) -> 0.29588872501244
```

expr REGEXP pat, expr RLIKE pat

Возвращает 1, если *expr* сопоставляется с регулярным выражением *pat*, в противном случае 0 (REGEXP и RLIKE - синонимы). Например:

```
SELECT 'oolong' REGEXP '^[a-z]' -> 1
```

RELEASE_LOCK(str)

Освобождает блокировку с именем *str*, захваченную при вызове функции *GET_LOCK*. Возвращает 1, если блокировка освобождена, или NULL, если блокировки в таким именем не существует либо она захвачена в другом потоке (в последнем случае блокировка не освобождается). Например:

```
RELEASE_LOCK('lochness') -> 1
```

REPEAT(str, count)

Возвращает строку *str*, повторенную *count* раз. Например:

```
REPEAT('Duck', 3) -> 'DuckDuckDuck'
```

REPLACE(str, from_str, to_str)

Возвращает строку *str*, в которой все вхождения подстроки *from_str* заменены строкой *to_str*. Например:

```
REPLACE('change', 'e', 'ing') -> 'changing'
```

REVERSE(str)

Возвращает обращенную строку *str*. Например:

```
REVERSE('STOP') -> 'POTS'
```

RIGHT(str, len)

Возвращает *len* самых правых символов из строки *str*. Например:

```
RIGHT('Hello, World!', 6) -> 'World!'
```

ROUND(X[, D])

Возвращает аргумент *X*, округленный до *D* десятичных знаков после запятой. Если *D* равно 0, то результат не содержит ни десятичной точки, ни дробной части. Например:

```
ROUND(12345.6789, 2) -> 12345.68
```

ROW_COUNT()

Возвращает количество строк, обновленных предыдущей командой. Например:

```
SELECT ROW_COUNT( ) -> 4
```

RPAD(str, len, padstr)

Возвращает строку *str*, дополненную справа строкой *padstr* до длины *len* символов. Например:

```
RPAD('duck',6,'s') -> 'duckss'
```

RTRIM(str)

Возвращает строку *str*, из которой удалены конечные пробелы. Например:

```
RTRIM(' welcome ') -> ' welcome'
```

SCHEMA( )

Синоним функции *DATABASE*.

SEC_TO_TIME(seconds)

Возвращает аргумент *seconds*, представленный в формате HH:MM:SS или HHMMSS, в зависимости от того, вызвана функция в строковом или числовом контексте. Например:

```
SEC_TO_TIME(256) -> 00:04:16
```

SECOND(time)

Возвращает число, представляющее секунды указанного времени *time* (от 0 до 59). Например:

```
SECOND('08:20:15') -> 15
```

SESSION_USER( )

Синоним функции *USER*.

SHA(X) или SHA1(X)

Возвращает 160-битовую контрольную сумму аргумента *X*, вычисленную по алгоритму SHA1. Например:

```
SHA('abc') -> 'a9993e364706816aba3e25717850c26c9cd0d89d'
```

SIGN(X)

Возвращает знак аргумента: -1, 0 или 1 в зависимости от того, является *X* отрицательным, нулевым или положительным числом. Например:

```
SIGN(-3.1415926) -> -1
```

SIN(number)

Возвращает синус числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT SIN( 0 ) -> 0.000000
```

SLEEP(s)

Вызывает приостановку выполнения на *s* секунд. Например:

```
SLEEP( 60 ) -> 0
```

SOUNDEX(str)

Возвращает soundex-строку, соответствующую *str* (для сравнения по звучанию). Например:

```
SOUNDEX('thimble') -> 'T514'
```

expr1 SOUNDS LIKE expr2

То же самое, что выражение

```
SOUNDEX(expr1) = SOUNDEX(expr2)
```

SPACE(n)

Возвращает строку, состоящую из *n* пробелов. Например:

```
SPACE(5) -> '  '
```

STD(expr), STDDEV(expr)

Возвращает стандартное отклонение *expr*. Для совместимости с Oracle имеется также вариант этой функции с именем *STDDEV*. Например:

```
STD(5) -> NULL
```

STR_TO_DATE(str, format)

Возвращает дату, полученную в результате разбора строки *str* в соответствии с форматной строкой *format*. Эта функция противоположна *DATE_FORMAT*; полный перечень допустимых спецификаторов формата см. в описании *DATE_FORMAT*. Например:

```
STR_TO_DATE('28/08/1976', '%d/%m/%Y') -> '1976-08-28'
```

STRCMP(expr1, expr2)

Возвращает 0, если аргументы равны; -1, если первый аргумент меньше второго согласно текущему порядку сортировки; 1 в противном случае. Например:

```
STRCMP('DUCKY', 'DUCK') -> 1
STRCMP('DUCK', 'DUCK') -> 0
```

SUBSTRING(str, pos), SUBSTRING(str FROM pos)

Возвращают подстроку строки *str*, начинающуюся в позиции *pos*. Например:

```
SUBSTRING('Hello, World!', 8) -> 'World!'
SUBSTRING('Hello, World!' FROM 8) -> 'World!'
```

SUBSTRING(str, pos, len), MID(str, pos, len)

Возвращают подстроку строки *str*, начинающуюся в позиции *pos* и длиной *len* символов. Обе эти функции являются синонимами *SUBSTRING(str FROM pos FOR len)*, определенной в стандарте ANSI SQL92. Например:

```
SUBSTRING('Hello, World!', 8, 10) -> 'World!'
SUBSTRING('Hello, World!' FROM 8 FOR 10) -> 'World!'
```

SUBSTRING_INDEX(str, delim, count)

Возвращает часть строки *str*, остающуюся после *count* вхождений разделителя *delim*. Например:

```
SUBSTRING_INDEX('www.mysql.com', '.', 2) -> 'www.mysql'
```

SUBTIME(expr1, expr2)

Возвращает результат вычитания момента времени *expr2* из *expr1*. Например:

```
SUBTIME('2008-01-31 16:30:00.999999', '0 0:30:0.999996') -> '2008-01-31
16:00:00.000003'
```

TAN(number)

Возвращает тангенс числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT TAN( 3.1415 ) -> -0.000093
```

TIME(expr)

Возвращает часть значения *expr*, содержащую время. Например:

```
TIME('1976-08-28 08:20:15') -> '08:20:15'
```

TIME_FORMAT(time, format)

Используется аналогично *DATE_FORMAT*, но в строке *format* можно употреблять только спецификаторы формата, относящиеся к часам, минутам и секундам. Все прочие спецификаторы порождают значение NULL или 0. Например:

```
TIME_FORMAT('2003-04-15 08:20:15', '%r') -> 08:20:15 AM
```

TIME_TO_SEC(time)

Возвращает результат преобразования аргумента *time* в секунды. Например:

```
TIME_TO_SEC('08:20:15') -> 30015
```

TIMEDIFF(expr1, expr2)

Возвращает разность между моментами времени *expr1* и *expr2*. Например:

```
TIMEDIFF('1976-08-28 08:20:15', '1976-08-28 08:20:16') ->
'00:00:01.000000'
```

TIMESTAMP(expr1[, expr2])

Возвращает значение временной метки, для которой дата берется из выражения *expr1*, а время – из выражения *expr2*. Например:

```
TIMESTAMP('1976-08-28', '08:20:16') -> '1976-08-28 08:20:16'
```

TIMESTAMPADD(unit, interval, expr)

Возвращает значение временной метки, сконструированной путем прибавления интервала *interval* к выражению *expr1* типа *DATE*. Единица измерения времени в интервале задается параметром *unit*. Например:

```
TIMESTAMPADD(DAY, 2, '1976-08-28') -> '1976-08-30 00:00:00'
```

TIMESTAMPDIFF(unit, expr1, expr2)

Возвращает целое число, являющееся результатом вычитания *expr1* из *expr2*. Единица измерения времени в результате задается параметром *unit*. Например:

```
TIMESTAMPADD(DAY, '1976-08-30', '1976-08-28') -> -2
```

TO_DAYS(date)

Возвращает порядковый номер дня, соответствующий заданной дате *date* (количество дней, истекших с начала года 0). Например:

```
TO_DAYS('1999-04-15') -> 730224
```

TRUNCATE(X, D)

Возвращает число *X*, в котором оставлено *D* десятичных знаков после запятой (остальные отброшены без округления). Если *D* равно 0, то результат не содержит ни десятичной точки, ни дробной части. Например:

```
TRUNCATE('123.456', 2) -> '123.45'
TRUNCATE('123.456', 0) -> '123'
TRUNCATE('123.456', -1) -> '120'
```

UCASE(str)

Синоним *UPPER(str)*. Например:

```
UCASE('duck') -> 'DUCK'
```

UNCOMPRESS(string)

Возвращает распакованную строку *string*.

UNCOMPRESS_LENGTH(string)

Возвращает длину распакованной строки *string*.

UNHEX(str)

Возвращает двоичную строку, составленную из шестнадцатеричных символов строки *str*.

UNIX_TIMESTAMP( ), UNIX_TIMESTAMP(date)

При вызове без аргументов возвращает временную метку Unix (количество секунд с момента 1970-01-01 00:00:00 GMT). Если передан аргумент *date*, то возвращает его значение в виде количества секунд с момента 1970-01-01 00:00:00 GMT. Например:

```
UNIX_TIMESTAMP( ) -> 1056512427
UNIX_TIMESTAMP('1999-04-15') -> 924159600
```

UPDATEXML(xml_target, xpath_expr, new_xml)

Возвращает фрагмент XML, полученный путем подстановки *new_xml* в места *xml_target*, определяемые XPath-выражением *xpath_expr*. Например:

```
UPDATEXML('<c><v>unknown</v></c>', '//v', '<v>Acme</v>') -> '<c><v>Acme</v></c>'
```

USER( ), SYSTEM_USER( ), SESSION_USER( )

Эти функции возвращают имя текущего пользователя MySQL. Например:

```
USER( ) -> 'login@machine.com'
SYSTEM_USER( ) -> 'login@machine.com'
SESSION_USER( ) -> 'login@machine.com'
```

UTC_DATE( )

Возвращает UTC-дату (универсальное координированное время). Например:

```
UTC_DATE( ) -> '2008-05-15'
```

UTC_TIME( )

Возвращает UTC-время. Например:

```
UTC_TIME( ) -> '01:01:00'
```

UTC_TIMESTAMP( )

Возвращает UTC-дату и время. Например:

```
UTC_TIMESTAMP( ) -> '2008-05-15 01:01:00'
```

UUID( )

Возвращает универсальный уникальный идентификатор (UUID). Например:

```
UUID( ) -> '1ae84bc9-5e4d-8f22-1f2e-123456789abc'
```

VARIANCE(expr)

Синоним *VAR_POP*.

VERSION( )

Возвращает строку, содержащую версию MySQL-сервера. Например:

```
VERSION( ) -> '4.0.12-standard'
```

WEEK(date), WEEK(date, first)

Если задан один аргумент, возвращает номер недели, соответствующий дате *date* (число от 1 до 53). (Некоторые года могут содержать начало 53-й недели.) В варианте с двумя аргументами функция *WEEK* позволяет указать, следует ли считать началом недели воскресенье (0) или понедельник (1). Например:

```
WEEK('1999-04-15') -> 15
```

WEEKDAY(date)

Возвращает номер дня недели, соответствующего дате *date* (0 = понедельник, 1 = вторник,..., 6 = воскресенье). Например:

```
WEEKDAY('1999-04-15') -> 3
```

WEEKOFYEAR(date)

Возвращает номер календарной недели, соответствующей дате *date* (число от 1 до 53 включительно). Например:

```
WEEKOFYEAR('2008-01-01') -> 1
```

op1 XOR op2

Возвращает результат применения операции ИСКЛЮЧАЮЩЕЕ ИЛИ к операндам *op1* и *op2*. Например:

```
SELECT 1 XOR 1, 1 XOR 0 -> 0, 1
```

YEAR(date)

Возвращает год, соответствующий дате *date* (число от 1000 до 9999). Например:

```
YEAR('1999-04-15') -> 1999
```

YEARWEEK(date), YEARWEEK(date, first)

Возвращает год и номер недели в году, соответствующие дате *date*. Второй аргумент интерпретируется точно так же, как второй аргумент функции *WEEK*. Отметим, что для первой и последней недели года возвращенный год может отличаться от года, указанного в дате *date*. Например:

```
YEARWEEK('1999-04-15') -> 199915
```

Функции, поддерживаемые Oracle

В этом разделе приведен алфавитный перечень функций, специфичных для СУБД Oracle, – с примерами.

ACOS(number)

Возвращает арккосинус числа *number*, находящегося в диапазоне от -1 до 1. Результат принимает значения от 0 до π и выражен в радианах. Например:

```
SELECT ACOS( 0 ) FROM DUAL -> 1.570796
```

ADD_MONTHS(date, months)

Возвращает результат прибавления *months* месяцев к дате *date*. Например:

```
SELECT ADD_MONTHS('15-APR-1999', 3) FROM DUAL -> 15-JUL-99
```

APPENDCHILDXML(xml_fragment, xpath, value[,namespace])

Вставляет строку *value* в место фрагмента XML *xml_fragment*, определяемое XPath-выражением *xpath*, и возвращает результат. Необязательный аргумент *namespace* задает информацию о пространстве имен для XPath-выражения. Например:

```
SELECT APPENDCHILDXML(XMLTYPE('<a><b>B</b></a>'), '/a', XMLType('<c>C</c>')) FROM
DUAL
'<a>
   <b>B</b>
   <c>C</c>
</a>'
```

ASCII(text)

Возвращает ASCII-код первого символа строки *text*. Например:

```
SELECT ASCII('x') FROM DUAL -> 120
```

ASCIISTR(text)

Преобразует строку *text* из произвольной кодировки в кодировку ASCII. Те символы *text*, которым нет эквивалента в кодировке ASCII, заменяются строкой *XXXX*, где *XXXX* – шестнадцатеричная кодовая позиция в кодировке UTF-16. Например:

```
SELECT ASCIISTR('Привет Peter') FROM DUAL -> '\040F\0430\0401\045E\0490\0432 Peter'
```

ASIN(number)

Возвращает арксинус числа *number*, находящегося в диапазоне от -1 до 1. Результат принимает значения от $-\pi/2$ до $\pi/2$ и выражен в радианах. Например:

```
SELECT ASIN( 0 ) FROM DUAL -> 0.000000
```

ATAN(number)

Возвращает арктангенс произвольного числа *number*. Результат принимает значения от $-\pi/2$ до $\pi/2$ и выражен в радианах. Например:

```
SELECT ATAN( 3.1415 ) FROM DUAL -> 1.262619
```

ATAN2(number, nbr)

Возвращает арктангенс числа. Функция *ATAN2(x, y)* аналогична *ATAN(y/x)* с тем отличием, что для определения квадранта, в котором находится результат, используются знаки аргументов *x* и *y*. Например:

```
SELECT ATAN2(3.1415, 1) FROM DUAL -> 1.26261873
```

BFILENAME(directory, filename)

Возвращает указатель *BFILE*, ассоциированный с физическим двоичным LOB-файлом, хранящимся в каталоге *directory* файловой системы сервера под именем *filename*.

BIN_TO_NUM(expr[,... n])

Возвращает десятичное число, эквивалентное битовому вектору, представленному списком параметров *expr*. Например:

```
SELECT BIN_TO_NUM(1,0,1) FROM DUAL -> 5
```

BITAND(integer1, integer2)

Возвращает результат применения поразрядной операции И к двум целочисленным аргументам. Например:

```
SELECT BITAND(101, 2) FROM DUAL -> 0
```

CARDINALITY(nested_table)

Возвращает количество элементов (кардинальное число) во вложенной таблице *nested_table*. Если таблица *nested_table* пуста, возвращает NULL. Например:

```
SELECT CARDINALITY(mytable) FROM DUAL -> 6
```

CHARTOROWID(char)

Преобразует значение символьного типа (*CHAR* или *VARCHAR2*) в тип *ROWID*.

CHR(number [USING NCHAR_CS])

Возвращает символ с двоичным кодом *number* в кодировке базы данных (если фраза *USING NCHAR_CS* опущена) или в национальной кодировке (если фраза *USING NCHAR_CS* включена).

CLUSTER_ID( ), CLUSTER_PROBABILITY( ), CLUSTER_SET( )

Поддержка технологии добычи данных (Data Mining). Дополнительную информацию об этих функциях см. в документации по Oracle Data Mining Java API или по пакету *DBMS_DATA_MINING*.

COALESCE(list)

Возвращает первый отличный от NULL элемент списка. Например:

```
SELECT COALESCE( NULL, 1, 2 ) FROM DUAL -> 1
```

COLLECT(column)

Для каждой группы создает вложенную таблицу, содержащую все значения в столбце *column*. Это агрегатная функция.

COMPOSE(string)

Возвращает результат преобразования *string* в полностью нормализованную строку UNICODE.

CONCAT(string1, string2)

Возвращает результат конкатенации строк *string1* и *string2*. Функция *CONCAT* эквивалентна оператору конкатенации ||. Например:

```
SELECT CONCAT( au_lname, au_fname ) FROM AUTHORS -> 'JeffersonThomas'
```

CONVERT(char_value, target_char_set, source_char_set)

Преобразует символьную строку из одной кодировки в другую. Возвращает результат преобразования строки *char_value*, заданной в кодировке *source_char_set*, в кодировку *target_char_set*.

CORR_K(expr1, expr2[, return_type])
CORR_S(expr1, expr2[, return_type])

Функция *CORR_K* возвращает коэффициент корреляции Кендалла (tau-b), а *CORR_S* – коэффициент корреляции Спирмена между двумя наборами пронумерованных пар (*expr1* и *expr2*). Аргумент *return_type* типа *VARCHAR2* может быть опущен или принимать одно из следующих значений: '*COEFFICIENT*', '*ONE_SIDED_SIG*', '*TWO_SIDED_SIG*'. Значение '*COEFFICIENT*' (подразумеваемое по умолчанию, если аргумент отсутствует) означает, что нужно вернуть коэффициент корреляции. Значения '*ONE_SIDED_SIG*' и '*TWO_SIDED_SIG*' означают, что нужно вернуть одностороннюю или двустороннюю значимость корреляции соответственно.

COS(number)

Возвращает косинус числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT COS(0) FROM DUAL -> 1.000000
```

COSH(number)

Возвращает гиперболический косинус числа *number*. Например:

```
SELECT COSH(180) FROM DUAL -> 7.4469E+77
```

CUBE_TABLE('expr')

Извлекает двумерное реляционное представление из OLAP-куба (одно измерение). Дополнительную информацию см. в документации по Oracle OLAP.

CV([dimension_column])

Имеет смысл только в контексте межстроковых вычислений, выполняемых внутри фразы *MODEL* команды *SELECT*. Эта функция возвращает текущее значение столбца *dimension_column*. Она может встречаться только в правой части правила, так как возвращает значение столбца *dimension_column* из левой части того же правила.

DATAOBJ_TO_PARTITION(table, partition_id)

Возвращает идентификатор секции для секционированной таблицы, заданной аргументами. Дополнительную информацию см. в документации Oracle по этой функции.

DBTIMEZONE

Возвращает смещение часового пояса сервера базы данных от UTC. Например:

```
SELECT DBTIMEZONE FROM DUAL -> +00:00
```

DECODE(expr, search, result[, search, result[,... n]][, default])

Сравнивает выражение *expr* по очереди с каждым значением *search*. При первом совпадении возвращает следующее за ним выражение *result*. Например:

```
DECODE ('B','A',1,'B',2,...'Z',26,'?') -> 2
```

Если совпадение не найдено, то *DECODE* возвращает значение *default* или NULL, если параметр *default* опущен. Дополнительную информацию см. в документации по Oracle. Предпочтительнее пользоваться конструкцией *CASE*, поскольку она является частью стандарта ANSI SQL.

DECOMPOSE(string [{CANONICAL|COMPATIBILITY}])

Возвращает результат декомпозиции строки *string* в кодовые позции UNICODE. Второй аргумент определяет тип декомпозиции. Подразумеваемое по умолчанию значение *CANONICAL* позволяет восстановить исходную UNICODE-строку.

DELETEXML(xml_fragment, xpath[, namespace])

Удаляет из фрагмента XML *xml_fragment* узлы, соответствующие XPath-выражению *xpath*, и возвращает результат. Необязательный параметр *namespace* задает пространство имен для XPath-выражения. Например:

```
SELECT DELETEXML(XMLTYPE('<a><b>Sifl</b><c>Olly</c></a>'), '/a/c') FROM DUAL
'<a><b>Sifl</b></a>'
```

DEPTH(number)

Возвращает глубину пути, заданную в условии *UNDER_PATH* в XML-запросе. Дополнительную информацию см. в документации по Oracle.

DEREF(expression)

Возвращает объект, на который ссылается выражение *expression*, которое должно быть ссылкой (*REF*) на объект.

DUMP(expression[, return_format[, starting_at3 length]]])

Возвращает строку типа *VARCHAR2*, содержащую характеристики выражения *expression*: код типа данных, длину в байтах и внутреннее представление. Значение возвращается в формате *return_format*. Например:

```
SELECT DUMP('abc', 1016) FROM DUAL
Typ=96 Len=3 CharacterSet=AL32UTF8: 61,62,63
```

EMPTY_BLOB( ), EMPTY_CLOB( )

Возвращает указатель на пустой *LOB*-объект, который можно использовать для инициализации переменной типа *LOB*, а также для стирания колонки или атрибута типа *LOB* в команде *INSERT* или *UPDATE*.

EXISTSNODE(instance, xpath[, namespace])

Возвращает 1, если XPath-запрос, заданный параметром *xpath*, вернет хотя бы один узел из фрагмента *instance*; иначе возвращает 0. Необязательный параметр *namespace* задает пространство имен XML в запросе. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

EXTRACT(instance, xpath[, namespace])

Возвращает узлы фрагмента XML *instance*, возвращаемые XPath-запросом *xpath*. Необязательный параметр *namespace* задает пространство имен XML в запросе. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference. (В Oracle есть также функция *EXTRACT*, относящаяся к работе с датами, она была рассмотрена выше в этой главе.) Например:

```
SELECT EXTRACT(XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),
               '/foo/bar') from DUAL
<bar>Hello, World!</bar>
```

EXTRACTVALUE(instance, xpath[, namespace])

Возвращает значение, хранящееся в узле XML-документа, который возвращает XPath-запрос *xpath*. Необязательный параметр *namespace* задает пространство имен XML в запросе. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference. Например:

```
SELECTEXTRACTVALUE(XMLTYPE('<foo><bar>Hello,World!</bar></foo>'),
                   '/foo/bar') from DUAL
Hello, World!
```

FEATURE _ID( ), FEATURE _SET( ), FEATURE_VALUE( )

Поддержка технологии добычи данных. Дополнительную информацию об этих функциях см. в документации по Oracle Data Mining Java API или по пакету *DBMS_DATA_MINING*.

FIRST

Агрегатная функция, которая возвращает заданное значение из строки с первым рангом в смысле порядка, определяемого фразой *ORDER BY*. Синтаксис выглядит следующим образом:

```
aggregate(aexpr) KEEP (DENSE_RANK FIRST ORDER BY expr[, ... n])
```

где *expr* имеет следующий синтаксис:

```
expr := [ASC | DESC] [NULLS {FIRST | LAST}]
```

Строка с первым рангом относительно порядка, определяемого выражением *expr*, используется в агрегатной функции *aggregate*. Этой функции передается выражение *aexpr*. Например:

```
SELECT MAX(c1) KEEP (DENSE_RANK FIRST ORDER BY c2) FROM FIVE_NUMS
1
```

FIRST_VALUE(expression IGNORE NULLS) OVER (window_clause)

Возвращает первое из упорядоченного множества значений. *FIRST_VALUE* – аналитическая функция. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT FIRST_VALUE(col1) OVER ( ) FROM NUMS
1
1
1
1
```

FROM_TZ(timestamp, timezone)

Возвращает значение *timestamp*, преобразованное к типу *TIMESTAMP WITH TIME ZONE*. Параметр *timestamp* должен иметь тип *TIMESTAMP*, а параметр *timezone* должен быть строкой в формате TZH:TZM. Например:

```
SELECT FROM_TZ(TIMESTAMP '2004-04-15 23:59:59', '8:00') FROM DUAL
'15-APR-04 11.59.59 PM +08:00'
```

GREATEST(expression[,… n])

Возвращает наибольшее выражение в списке. Все выражения, начиная со второго, перед сравнением неявно преобразуются к типу первого выражения. Например:

```
SELECT GREATEST(8,2,4) FROM DUAL -> 8
```

GROUP_ID( )

Возвращает положительное значение для каждой повторяющейся группы, возвращенной запросом с фразой *GROUP BY*. Эта функция полезна для отбрасывания повторяющихся групп, порождаемых при выполнении операций *CUBE*, *ROLLUP* и других расширений *GROUP BY* (см. описание функции *GROUPING*).

GROUPING(column_name)

Возвращает 1, если строка добавлена в результате выполнения операции *CUBE*, *ROLLUP* или иного расширения *GROUP BY*; в противном случае возвращает 0. Например:

```
SELECT royalty, SUM(advance) 'total advance',
GROUPING(royalty) 'grp'
FROM titles
GROUP BY royalty WITH ROLLUP
```

```
royalty   total advance         grp
--------- --------------------- ---
NULL       NULL                 0
10         57000.0000           0
12         2275.0000            0
14         4000.0000            0
16         7000.0000            0
24         25125.0000           0
NULL       95400.0000           1
```

GROUPING_ID(column_name1[, column_name2, ...])

Возвращает десятичное число, равное двоичной величине, построенной путем конкатенации значений *GROUPING* для каждого параметра. Функция полезна, когда запрос возвращает результат, содержащий несколько уровней агрегирования, созданных выражениями *GROUP BY*. Предпочтительнее использовать ее вместо нескольких функций *GROUPING* в одном запросе. Эта функция является сокращенной записью следующей конструкции:

```
BIN_TO_NUM( GROUPING(column_name1)[, GROUPING(column_name2), ...] )
```

HEXTORAW(string)

Преобразует строку *string*, состоящую из шестнадцатеричных цифр, в двоичное значение. Например:

```
SELECT HEXTORAW('0FE') FROM DUAL -> '00FE'
```

INITCAP(string)

Возвращает строку *string*, в которой первая буква каждого слова переведена в верхний регистр, а остальные – в нижний. Например:

```
SELECT INITCAP('thomas jefferson') FROM DUAL -> 'Thomas Jefferson'
```

INSERTCHILDXML(xml_fragment, xpath, child_expr, value_expr[, namespace])

Вставляет узлы, определяемые выражением *child_expr*, из фрагмента XML *value_expr* в фрагмент *xml_fragment* в точке, определяемой XPath-запросом *xpath*, и возвращает результат. Необязательный параметр *namespace* задает пространство имен XML в запросе. Например:

```
SELECT INSERTCHILDXML(XMLTYPE('<a></a>'), '/a', 'b', XMLTYPE('<b>B1</b>')) FROM DUAL
'<a><b>B1</b></a>'
```

INSERTXMLBEFORE(xml_fragment, xpath, value_expr[, namespace])

Вставляет *value_expr* в фрагмент *xml_fragment* в точке, определяемой XPath-запросом *xpath*, и возвращает результат. Необязательный параметр *namespace* задает пространство имен XML в запросе. Например:

```
SELECT INSERTXMLBEFORE(XMLTYPE('<a><b>B2</b></a>'), '/a/b', XMLTYPE('<b>B1</b>'))
FROM DUAL
'<a><b>B1</b><b>B2</b></a>'
```

INSTR(string1, string2[, start_at[, occurrence]])

Возвращает позицию строки *string2* в строке *string1*. Функция просматривает строку *string1*, начиная с позиции *start_at* (целое число), и ищет вхождение строки *string2* с указанным номером *occurrence*. Например:

```
SELECT INSTR('foobar', 'o', 1, 1) FROM DUAL -> 2
```

Используйте функцию *INSTRB* для строк, состоящих из байтов, *INSTRC* – для строк, состоящих из полных символов UNICODE, *INSTR2* – для строк, состоящих из кодовых позиций UNICODE *UCS2*, и *INSTR4* – для строк, состоящих из кодовых позиций UNICODE *UCS4*.

ITERATION_NUMBER

Имеет смысл только в контексте межстроковых вычислений, выполняемых внутри фразы *MODEL* команды *SELECT*. Эта функция показывает, сколько раз в процессе обработки запроса были выполнены правила внутри фразы *MODEL*.

LAG(expression[,offset][,default]) OVER (window_clause)

Аналитическая функция, предоставляющая доступ сразу к нескольким строкам таблицы без выполнения самосоединения (self-join). Она позволяет получить величину с «запаздыванием» в результирующем наборе, которая «отстает» на *offset* строк от текущей строки. Первые *offset* строк в результирующем наборе инициализируются значением *default*, поскольку «запаздывание» для них не определено. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT c1, LAG(c1, 2, 0) OVER (ORDER BY c1) FROM FIVE_NUMS
1       0
2       0
3       1
4       2
5       3
```

LAST

Возвращает заданное значение из строки с последним рангом в смысле порядка, определяемого фразой *ORDER BY*. Синтаксис выглядит следующим образом:

```
aggregate(aexpr) KEEP (DENSE_RANK LAST ORDER BY expr[, ... n])
```

где *expr* имеет следующий синтаксис:

```
expr := [ASC | DESC] [NULLS {FIRST | LAST}]
```

Строка с последним рангом относительно порядка, определяемого выражением *expr*, используется в агрегатной функции *aggregate*. Этой функции передается выражение *aexpr*. Например:

```
SELECT MIN(c1) KEEP (DENSE_RANK LAST ORDER BY c1) FROM FIVE_NUMS
5
```

LAST_DAY(date)

Возвращает дату последнего дня месяца, содержащего дату *date*. Например:

```
SELECT LAST_DAY('15-APR-1999') FROM DUAL -> 30-APR-99
```

LAST_VALUE(expression [IGNORE NULLS]) OVER (window_clause)

Возвращает последнее из упорядоченного множества значений. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT c1, LAST_VALUE(c1) OVER (ORDER BY c1) FROM FIVE_NUMS
1       5
2       5
3       5
4       5
5       5
```

LEAD(expression[,offset][,default]) OVER (window_clause)

Аналитическая функция, предоставляющая доступ сразу к нескольким строкам таблицы без выполнения самосоединения (self-join). Она позволяет получить величину с «опережением» в результирующем наборе, которая «обгоняет» текущую строку на *offset* строк. Последние *offset* строк в результирующем наборе инициализируются значением *default*, поскольку «опережение» для них не определено. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT c1, LEAD(c1, 2) OVER (ORDER BY c1) FROM FIVE_NUMS
1       3
2       4
3       5
4
5
```

LEAST(expression[,...n])

Возвращает наименьшее выражение в списке. Например:

```
SELECT LEAST(10,5,3,7) FROM DUAL -> 3
```

LENGTH(string)

Возвращает длину строки *string* или NULL, если *string* равна NULL. Например:

```
SELECT LENGTH('DUCK') FROM DUAL -> 4
```

LENGTHB(string)

Возвращает длину строки *string* в байтах; в остальных отношениях аналогична функции *LENGTH*. Например:

```
SELECT LENGTHB('DUCK') FROM DUAL -> 4
```

Используйте функцию *LENGTHB* для строк, состоящих из байтов, *LENGTHC* – для строк, состоящих из полных символов UNICODE, *LENGTH2* – для строк, состоящих из кодовых позиций UNICODE *UCS2*, и *LENGTH4* – для строк, состоящих из кодовых позиций UNICODE *UCS4*.

LNNVL(condition)

Возвращает *TRUE*, если условие *condition* ложно или если хотя бы один из операндов в *condition* равен NULL; в противном случае возвращает *FALSE*. Например:

```
SELECT COUNT(*) FROM authors WHERE LNNVL( contract <> 1 ) -> 4
```

LOCALTIMESTAMP[(precision)]

Возвращает значение типа *TIMESTAMP*, соответствующее текущей дате и времени для текущей сессии. Эта функция аналогична *CURRENT_TIME-*

STAMP с тем отличием, что не включает во временную метку информацию о часовом поясе *TIME ZONE*. Например:

```
SELECT LOCALTIMESTAMP FROM DUAL -> '15-APR-05 03.15.00 PM'
```

LOG(base_number, number)

Возвращает логарифм числа *number* по основанию *base_number*. Например:

```
SELECT LOG(50,10) FROM DUAL -> .58859191
```

LPAD(string1,number[,string2])

Возвращает строку *string1*, дополненную слева строкой *string2* до длины *number* символов. По умолчанию подразумевается, что строка *string2* состоит из одного пробела. Например:

```
SELECT LPAD('ucks',5,'d') FROM DUAL -> 'ducks'
```

LTRIM(string[, set])

Удаляет из начала строки *string* все символы, встречающиеся в строке *set*. По умолчанию *set* состоит из одного пробела. Например:

```
SELECT LTRIM(' Howdy! ',' ') FROM DUAL -> 'Howdy! '
```

MAKE_REF({table_name|view_name}, key[, ... n])

Создает ссылку (*REF*) на строку в объектном представлении или в объектной таблице, для которой идентификатор объекта основан на первичном ключе.

MEDIAN(expression) OVER (partitioning)

Возвращает медиану упорядоченного набора чисел или дат. О том, что означает параметр *partitioning*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT MEDIAN(c1) FROM FIVE_NUMS -> 3
```

MONTHS_BETWEEN(date1, date2)

Возвращает количество месяцев между датами *date1* и *date2*. Если *date1* позже *date2*, то результат положителен, иначе отрицателен. Например:

```
SELECT MONTHS_BETWEEN('15-APR-2000', '15-JUL-1999') FROM DUAL -> 9
```

NANVL(a, b)

Возвращает *b*, если *a* — «не число» (*NaN*), иначе возвращает *a*. Вычисление выражения *a* должно давать число типа *BINARY_FLOAT* или *BINARY_DOUBLE*, поскольку только эти типы позволяют сохранить значение *NaN*. Например:

```
SELECT c1, NANVL(c1, 0) FROM NUMS
1.0E+000       1.0E+000
2.0E+000       2.0E+000
Nan            0
```

NEW_TIME(date, time_zone1, time_zone2)

Возвращает дату и время в часовом поясе *time_zone2* в предположении, что исходная дата *date* относится к часовому поясу *time_zone1*. Например:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH12:MI:SS'
SELECT NEW_TIME(TO_DATE('04-15-99 08:22:31', 'MM-DD-YY HH12:MI:SS'), 'AST', 'PST')
```

```
FROM DUAL
15-APR-2099 04:22:31
```

time_zone1 и *time_zone2* могут принимать следующие значения:

'AST', 'ADT'

Атлантическое стандартное или летнее время

'BST', 'BDT'

Берингово стандартное или летнее время

'CST', 'CDT'

Центральное стандартное или летнее время

'EST', 'EDT'

Восточное стандартное или летнее время

'GMT'

Гринвичское время

'HST', 'HDT'

Аляскинское-гавайское стандартное или летнее время

'MST', 'MDT'

Стандартное или летнее горное время

'NST'

Ньюфаундлендское стандартное время

'PST', 'PDT'

Тихоокеанское стандартное или летнее время

'YST', 'YDT'

Юконское стандартное или летнее время

NEXT_DAY(date, string)

Возвращает дату первого дня недели с названием *string*, следующего за датой *date*. Аргумент *string* должен быть полным или сокращенным названием дня на языке текущего сеанса. Например:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY'
SELECT NEXT_DAY('15-APR-1999', 'SUNDAY') FROM DUAL
18-APR-1999
```

NLS_CHARSET_DECL_LEN(bytecnt, csid)

Возвращает количество символов в столбце типа NCHAR шириной *bytecnt* в кодировке с идентификатором *csid*.

NLS_CHARSET_ID(text)

Возвращает идентификатор NLS-кодировки по ее имени, заданном в параметре *text*.

NLS_CHARSET_NAME(number)

Возвращает имя NLS-кодировки с идентификатором *number* в виде строки типа *VARCHAR2*.

NLS_INITCAP(string[, nlsparameter])

Возвращает строку *string*, в которой первая буква каждого слова переведена в верхний регистр, а остальные – в нижний. Параметр *nlsparameter* позволяет указать языковые особенности сортировки.

NLS_LOWER(string[, nlsparameter])

Возвращает строку *string*, в которой все буквы переведены в нижний регистр. Параметр *nlsparameter* позволяет указать языковые особенности сортировки.

NLS_UPPER(string[, nlsparameter])

Возвращает строку *string*, в которой все буквы переведены в верхний регистр. Параметр *nlsparameter* позволяет указать языковые особенности сортировки.

NLSSORT(string[, nlsparameter])

Возвращает строку байтов, используемых для сортировки строки *string*. Параметр *nlsparameter* позволяет указать языковые особенности сортировки.

NTILE(expression) OVER ([partitioning] ordering)

Разбивает упорядоченный набор данных на группы, пронумерованные от 1 до *expression*, и назначает каждой строке номер содержащей ее группы. Строки распределяются по группам, так что количество строк в разных группах отличается не более чем на 1. О том, что означают параметры *partitioning* и *ordering*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT c1, NTILE(4) OVER (ORDER BY c1) FROM FIVE_NUMS
1       1
2       1
3       2
4       3
5       4
```

NULLIF(expr1, expr2)

Возвращает NULL, если *expr1* равно *expr2*, иначе возвращает *expr1*. Из двух аргументов только второй может быть равен NULL. Например:

```
SELECT c1, c2, NULLIF(c1, c2) FROM NUMS
1     2      1
2     2
3     2      3
```

NUMTODSINTERVAL(number, string)

Преобразует *number* в литерал *INTERVAL DAY TO SECOND*, где *number* – число или выражение, принимающее числовое значение, например столбец одного из числовых типов. Второй аргумент, *string*, определяет способ интерпретации *number*; он может принимать значения *'DAY'*, *'HOUR'*, *'MINUTE'*, *'SECOND'*. Например:

```
SELECT NUMTODSINTERVAL(100, 'DAY') FROM DUAL
+000000100 00:00:00.000000000
```

NUMTOYMINTERVAL(number, string)

Преобразует *number* в литерал *INTERVAL DAY TO MONTH*, где *number* – число или выражение, принимающее числовое значение, например столбец одного

из числовых типов. Второй аргумент, *string*, определяет способ интерпретации *number*; он может принимать значения '*YEAR*' или '*MONTH*'. Например:

```
SELECT NUMTOYMINTERVAL(100, 'YEAR') FROM DUAL
+000000100-00
```

NVL(expression1, expression2)

Если выражение *expression1* равно NULL, то возвращает *expression2*, в противном случае – *expression1*. Оба аргумента могут иметь произвольный тип данных. Например:

```
SELECT NVL(2,29) FROM DUAL -> 2
```

NVL2(expression1, expression2, expression3)

Аналогична функции NVL с тем отличием, что если выражение *expression1* *не* равно NULL, то возвращает *expression2*, в противном случае – *expression3*. Аргументы могут иметь произвольный тип данных, кроме *LONG*. Например:

```
SELECT NVL2(1,3,5) FROM DUAL -> 3
```

ORA_HASH(expression[, buckets[, seed]])

Вычисляет хеш-код выражения *expression* и возвращает номер кластера (bucket), исходя из вычисленного значения. Необязательный параметр *buckets* задает максимальное количество кластеров, которое на единицу меньше реально используемого количества, так как нумерация кластеров начинается с нуля. По умолчанию *buckets* равно 4 294 967 295. Необязательный параметр *seed* служит для инициализации хеш-функции, так чтобы можно было порождать различные результаты из одних и тех же данных, изменяя только значение *seed*. По умолчанию *seed* равно 0. В следующем примере все числа псевдослучайным образом распределяются по двум кластерам, после чего возвращаются те, что попали в первый кластер (таких должна быть примерно половина):

```
SELECT C1 FROM FIVE_NUMS WHERE
ORA_HASH( C1, 1, TO_CHAR( SYSTIMESTAMP, 'SSSS.FF' ) ) = 0
1
5
```

PATH(number)

Возвращает путь, заданный в условии *UNDER_PATH* с корреляционной переменной *number* в XML-запросе. Дополнительную информацию см. в документе Oracle SQL Reference.

POWERMULTISET(nested_table)
POWERMULTISET_BY_CARDINALITY(nested_table, cardinality)

Возвращает вложенную таблицу, состоящую из вложенных таблиц, которые содержат все непустые подмножества исходной вложенной таблицы, заданной параметром *nested_table*. У этой функции имеется дополнительный параметр, который позволяет задать минимальную мощность рассматриваемых подмножеств. Дополнительную информацию см. в документе Oracle SQL Reference.

PREDICTION( ), PREDICTION_BOUNDS( ), PREDICTION_COST( ), PREDICTION_DETAILS( ), PREDICTION_PROBABILTY( ), PREDICTION_SET( )

Поддержка технологии добычи данных. Дополнительную информацию об этих функциях см. в документации по Oracle Data MiningJava API или по пакету *DBMS_DATA_MINING*.

PRESENTNNV(cell_reference, expr1, expr2)

Имеет смысл только в контексте межстроковых вычислений, выполняемых внутри фразы *MODEL* команды *SELECT*. Эта функция возвращает *expr1*, когда *cell_reference* существует и не равно NULL, и *expr2* в противном случае.

PRESENTV(cell_reference, expr1, expr2)

Имеет смысл только в контексте межстроковых вычислений, выполняемых внутри фразы *MODEL* команды *SELECT*. Эта функция возвращает *expr1*, когда *cell_reference* существует, и *expr2* в противном случае.

PREVIOUS(cell_reference)

Имеет смысл только в контексте межстроковых вычислений, выполняемых внутри секции *ITERATE...[UNTIL]* во фразе *MODEL* команды *SELECT*. Эта функция возвращает значение, находившееся в *cell_reference* в момент начала итерации.

RATIO_TO_REPORT(value_exprs) OVER (partitioning)

Вычисляет отношение значения в *value_exprs* к сумме значений *value_exprs* по всем секциям. Если *value_exprs* равно NULL, то результат *RATIO_TO_REPORT* также равен NULL. О том, что означает параметр *partitioning*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT c1, RATIO_TO_REPORT(c1) OVER ( ) FROM FIVE_NUMS
1            .066666667
2            .133333333
3            .2
4            .266666667
5            .333333333
```

RAWTOHEX(raw)

Преобразует значение *raw* (символьного типа) в последовательность шестнадцатеричных кодов отдельных символов. Например:

```
SELECT RAWTOHEX('Hi') FROM DUAL -> 4869
```

RAWTONHEX(raw)

Преобразует значение *raw* в строку типа *NVARCHAR2*, содержащую последовательность шестнадцатеричных кодов отдельных символов.

REF(table_alias)

Принимает псевдоним таблицы, ассоциированный со строкой из объектной таблицы или объектного представления. Возвращает специальное значение-ссылку на экземпляр объекта, привязанное к переменной или строке.

REFTOHEX(expression)

Преобразует выражение *expression* в символьное значение, содержащее шестнадцатеричный эквивалент. *expression* должно возвращать REF.

REGEXP_INSTR(string, pattern[, start_at[, occurrence[, roption[, mparam]]]])

Просматривает строку *string*, начиная с позиции *start_at* (целое число, большее 0), в поисках вхождения регулярного выражения *pattern* с порядковым номером *occurrence*. Возвращает позицию начала подстроки, сопоставленной с образцом. По умолчанию параметры *start_at* и *occurrence* равны 1. Параметр *roption*, который может принимать значение 0 или 1, определяет, следует ли вернуть позицию первого символа сопоставленной подстроки или следующего за ним. По умолчанию *roption* равно 0. Параметр *mparam* позволяет модифицировать способ сопоставления и может принимать следующие значения:

- 'i' сравнивать без учета регистра
- 'c' сравнивать с учетом регистра
- 'n' символ '.' сопоставляется с символом новой строки
- 'm' входная строка может содержать символы новой строки; для сопоставления с началом строки применяйте метасимвол '^', а для сопоставления с концом строки – метасимвол '$'.

Например:

```
SELECT REGEXP_INSTR('Hello, World!', '([^ ]*)!', 1, 1) FROM DUAL
8
```

REGEXP_REPLACE(string, pattern[, newstr[, start_at[, occurrence[, mparam]]]])

Просматривает строку *string*, начиная с позиции *start_at* (целое число, большее 0), в поисках вхождения регулярного выражения *pattern* с порядковым номером *occurrence*. Возвращает результат замены всех подстрок, сопоставленных с образцом *pattern*, новой строкой *newstr*. По умолчанию параметры *start_at* и *occurrence* равны 1. Параметр *mparam* позволяет модифицировать способ сопоставления и может принимать следующие значения:

- 'i' сравнивать без учета регистра
- 'c' сравнивать с учетом регистра
- 'n' символ '.' сопоставляется с символом новой строки
- 'm' входная строка может содержать символы новой строки; для сопоставления с началом строки применяйте метасимвол '^', а для сопоставления с концом строки – метасимвол '$'.

Например:

```
SELECT REGEXP_REPLACE('Hello, World!', '([^ ]*!)', 'Reader!') FROM DUAL
'Hello, Reader!'
```

REGEXP_SUBSTR(string, pattern[, start_at[, occurrence[, mparam]]])

Просматривает строку *string*, начиная с позиции *start_at* (целое число, большее 0), в поисках вхождения регулярного выражения *pattern* с порядковым номером *occurrence*. Возвращает саму подстроку, сопоставленную с образцом. По умолчанию параметры *start_at* и *occurrence* равны 1. Параметр *mparam* по-

зволяет модифицировать способ сопоставления и может принимать следующие значения:

- 'i' сравнивать без учета регистра
- 'c' сравнивать с учетом регистра
- 'n' символ '.' сопоставляется с символом новой строки
- 'm' входная строка может содержать символы новой строки; для сопоставления с началом строки применяйте метасимвол '^', а для сопоставления с концом строки – метасимвол '$'.

Например:

```
SELECT REGEXP_SUBSTR('Hello, World!', '([^ ]*!)') FROM DUAL
'World!'
```

REMAINDER(m, n)

Возвращает остаток от деления *m* на *n*. Результат эквивалентен выражению

```
m-n*ROUND(m/n)
```

В функции *MOD* вместо *ROUND* используется *FLOOR*. Например:

```
SELECT REMAINDER(11, 4), MOD(11, 4) FROM DUAL
-1 3
```

REPLACE(string, search_string[, replacement_string])

Возвращает результат замены всех вхождений подстроки *search_string* в строку *string* строкой *replacement_string*. Например:

```
SELECT REPLACE('change', 'e', 'ing') FROM DUAL -> 'changing'
```

ROUND(date[, format])

Возвращает дату *date*, округленную до единицы измерения, заданной параметром *format*. Если этот параметр опущен, дата округляется до ближайшего дня. (Дополнительную информацию о спецификаторах формата см. в описании функции *TO_CHAR*.) Например:

```
SELECT ROUND(TO_DATE('15-APR-1999'), 'MONTH') FROM DUAL
01-APR-1999
```

ROUND(number[, decimal])

Округляет число *number* до *decimal* знаков после запятой. Если параметр *decimal* опущен, округление производится до целого. Отметим, что значение *decimal* может быть отрицательно, тогда округляются знаки слева от запятой. Например:

```
SELECT ROUND(12345.6789, 2) FROM DUAL -> 12345.68
```

ROWIDTOCHAR(rowid), ROWIDTONCHAR(rowid)

Функция *ROWIDTOCHAR* преобразует идентификатор строки *rowid* в значение типа *VARCHAR2*, состоящее из 18 знаков. Функция *ROWIDTONCHAR* преобразует *rowid* в значение типа *NVARCHAR2*, состоящее из 18 знаков. Например:

```
SELECT ROWIDTOCHAR(ROWID) FROM NUMS
ABAsxDAAKAAAAEqAAA
ABAsxDAAKAAAAEqAAB
```

```
ABAsxDAAKAAAAEqAAC
ABAsxDAAKAAAAEqAAD
```

RPAD(string1, number[, string2])

Возвращает строку *string1*, дополненную справа строкой *string2* до длины *number* символов. По умолчанию подразумевается, что строка *string2* состоит из одного пробела. Например:

```
SELECT RPAD('duck',8,'s') FROM DUAL -> 'ducksssss'
```

RTRIM(string[, set])

Удаляет из конца строки *string* все символы, встречающиеся в строке *set*. По умолчанию *set* состоит из одного пробела. Например:

```
SELECT RTRIM(' welcome ', ' ') FROM DUAL -> ' welcome'
```

SCN_TO_TIMESTAMP(scn)

Возвращает временную метку, ассоциированную с системным номером изменения *scn*. Например:

```
SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM NUMS WHERE c1 = 1
15-APR-04 02.56.05.000000000 PM
```

SESSIONTIMEZONE

Возвращает смещение часового пояса для текущего сеанса. Например:

```
SELECT SESSIONTIMEZONE FROM DUAL -> -06:00
```

SET(nested_table)

Возвращает вложенную таблицу, содержащую различающиеся элементы входной вложенной таблицы *nested_table*. Дополнительную информацию см. в документе Oracle SQL Reference.

SIGN(number)

Если *number* < *0*, возвращает *-1*. Если *number* = *0*, возвращает *0*. Если *number* > *0*, возвращает 1. Например:

```
SELECT SIGN(-3.1415926) FROM DUAL -> -1
```

SIN(number)

Возвращает синус числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT SIN( 0 ) -> 0.000000
```

SINH(number)

Возвращает гиперболический синус *number*. Например:

```
SELECT SINH(180) FROM DUAL -> 7.4469E+77
```

SOUNDEX(string)

Возвращает строку символов, содержащую фонетическое представление строки *string*. Эта функция позволяет сравнивать на равенство английские слова, которые пишутся по-разному, но звучат похоже. Например:

```
SELECT SOUNDEX('thimble') FROM DUAL -> 'T514'
```

STATS_BINOMIAL_TEST, STATS_CROSSTAB, STATS_F_TEST, STATS_KS_TEST, STATS_MODE, STATS_MW_TEST, STATS_ONE_WAY_ANOVA, STATS_T_TEST_INDEP, STATS_T_TEST_INDEPU, STATS_T_TEST_ONE, STATS_T_TEST_PAIRED, STATS_WSR_TEST

В Oracle имеется немало статистических функций. Дополнительную информацию о функциях *STATS_** см. в документе Oracle SQL Reference.

STDDEV([DISTINCT|ALL] expression) [OVER (window_clause)]

Возвращает выборочное стандартное отклонение набора чисел, представленных выражением *expression*. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT STDDEV(col1) FROM NUMS -> 5.71547607
```

STDEV_POP(expression) [OVER (window_clause)]

Вычисляет стандартное отклонение генеральной совокупности. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT STDDEV_POP(col1) FROM NUMS -> 4.94974747
```

STDDEV_SAMP(expression) [OVER (window_clause)]

Вычисляет кумулятивное выборочное стандартное отклонение. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT STDDEV_SAMP(col1) FROM NUMS -> 5.71547607
```

SUBSTR(string, start [FROM starting_position] [FOR length])

См. описание функции *SUBSTRING* в разделе «Строковые функции и операторы». Например:

```
SELECT SUBSTR('Hello, World!',8,10) FROM DUAL -> 'World!'
```

SUBSTR(string[, length])

Возвращает часть строки *string*, начинающуюся с позиции *start* длиной *length* символов. Если параметр *length* опущен, возвращаются все символы, начиная с позиции *start*. Если значение *start* отрицательно, то оно интерпретируется как смещение от конца строки. Например:

```
SELECT SUBSTR('Hello, World!', 8) FROM DUAL -> World!
```

Используйте функцию *SUBSTRB* для строк, состоящих из байтов, *SUBSTRC* – для строк, состоящих из полных символов UNICODE, *SUBSTR2* – для строк, состоящих из кодовых позиций UNICODE *UCS2*, и *SUBSTR4* – для строк, состоящих из кодовых позиций UNICODE *UCS4*.

SYS_CONNECT_BY_PATH(column, char)

Для иерархических запросов функция *SYS_CONNECT_BY_PATH* возвращает путь от корня к узлу с именем столбца, заданным параметром *column*. Параметр *char* определяет разделитель узлов в возвращаемом пути. Дополнительную информацию об иерархических запросах в Oracle см. в документе Oracle SQL Reference.

SYS_CONTEXT(namespace, attribute[, length])

Возвращает значение атрибута *attribute*, ассоциированного с контекстным пространством имен *namespace*. Может использоваться в командах SQL и PL/SQL. Необязательный параметр *length* определяет размер значения, возвращаемого функцией; по умолчанию он равен 256 байтов, но может принимать значения от 1 до 4000. Например:

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL
'LOGIN'
```

SYS_DBURIGEN(column[, rowid][, ...][, 'text( )'])

Возвращает URL, который может служить уникальной ссылкой на строку, заданную параметром *column*. Если столбец может содержать дубликаты, то дополнительно можно задать параметр *rowid*, гарантирующий, что URL будет указывать только на одну строку. Необязательный параметр '*text( )*' позволяет сгенерировать URL, который будет указывать не на сам XML-документ, а на содержащийся в нем текст.

SYS_EXTRACT_UTC(datetime)

Возвращает аргумент *datetime*, приведенный к времени UTC. Например:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2004-04-15 11:59:59.00 -08:00')
FROM DUAL
'15-APR-04 07.59.59.000000000 PM'
```

SYS_GUID( )

Генерирует и возвращает глобально уникальный идентификатор (значение типа *RAW*) длиной 16 байтов. Например:

```
SELECT SYS_GUID( ) FROM DUAL -> C0FD3FDC30148EAEE030440A49096A41
```

SYS_TYPEID(object_value)

Возвращает идентификатор типа параметра *object_value*.

SYS_XMLAGG(expr[, format])

Возвращает единый XML-документ, созданный путем агрегирования XML-документов или XML-фрагментов, заданных параметром *expr*. Необязательный параметр *format* можно использовать для форматирования результирующего XML-документа. Дополнительную информацию см. в документе Oracle SQL Reference.

SYS_XMLGEN(expression[, format])

Для каждой строки запроса вычисляет выражение *expression* и генерирует для него XML-документ. Необязательный параметр *format* можно использовать для форматирования результирующего XML-документа. Дополнительную информацию см. в документе Oracle SQL Reference.

SYSDATE

Возвращает текущую дату и время в системе, где работает сервер базы данных. Возвращаемое значение имеет тип *DATE*. Например:

```
SELECT SYSDATE FROM DUAL -> 26-JUN-2003
```

SYSTIMESTAMP

Возвращает текущие дату и время в системе, где работает сервер базы данных. Возвращаемое значение имеет тип *TIMESTAMP*. Например:

```
SELECT SYSTIMESTAMP FROM DUAL
26-JUN-2003 11.15.00.000000 PM -06:00
```

TAN(number)

Возвращает тангенс числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT TAN( 3.1415 ) FROM DUAL -> -0.000093
```

TANH(number)

Возвращает гиперболический тангенс *number*. Например:

```
SELECT TANH(180) FROM DUAL -> 1
```

TIMESTAMP_TO_SCN(timestamp_value)

Возвращает приблизительный (каждому моменту времени может соответствовать несколько изменений) номер системного изменения (SCN), ассоциированный с временной меткой *timestamp_value*. Возвращаемое значение имеет тип *NUMBER*.

TO_BINARY_DOUBLE(expr[, format[, nls_parameter]])

Преобразует *expr* в тип *BINARY_DOUBLE*. Необязательный параметр *format* задает форматирование. Если *expr* – символьное выражение, то параметры *format* и *nls_parameter* эквивалентны и интерпретируются так же, как в функции *TO_CHAR*. Если *expr* – числовое выражение, то задавать параметры *format* и *nls_parameter* запрещено. Например:

```
SELECT c1, TO_BINARY_DOUBLE(c1) FROM NUMS
1          1.0E+000
2          2.0E+000
3          3.0E+000
```

TO_BINARY_FLOAT(expr[, format[, nls_parameter]])

Преобразует *expr* в тип *BINARY_FLOAT*. Необязательный параметр *format* задает форматирование. Если *expr* – символьное выражение, то параметры *format* и *nls_parameter* эквивалентны и интерпретируются так же, как в функции *TO_CHAR*. Если *expr* – числовое выражение, то задавать параметры *format* и *nls_parameter* запрещено. Например:

```
SELECT c1, TO_BINARY_FLOAT(c1) FROM NUMS
1          1.0E+000
2          2.0E+000
3          3.0E+000
```

TO_CHAR(character_expr)

Преобразует выражение *character_expr* в кодировку базы данных. Например:

```
SELECT TO_CHAR('Howdy') FROM DUAL
Howdy
```

TO_CHAR(date|interval[,format[,nls_parameter]])

Преобразует дату *date* или интервал *interval* в формат, определяемый параметром *format*. Если этот параметр опущен, *date* преобразуется в формат, подразумеваемый по умолчанию для даты. Параметр *nls_parameter* обеспечивает дополнительный контроль над форматированием. Например:

```
SELECT TO_CHAR(TO_DATE('15-APR-1999') ,'MON-DD-YYYY') FROM DUAL
APR-15-1999
```

В табл. 4.11 перечислены допустимые спецификаторы формата.

Таблица 4.11. Спецификаторы формата в Oracle

Спецификатор формата	Назначение
AD или *A.D.*	Индикатор новой эры
AM или *A.M.*	Индикатор полудня
BC или *B.C.*	Индикатор старой эры (до Рождества Христова)
D	День недели (1–7)
DAY	Название дня недели
DD	День месяца (1–31)
DDD	Порядковый номер дня в году (1–366)
DL	Дата в длинном формате
DS	Дата в коротком формате
DY	Сокращенное название дня
FF	Дробная часть секунды; для задания точности укажите после спецификатора FF число от 1 до 9
HH или *HH12*	Час (1–12)
HH24	Час (0–23)
J	Юлианский день; количество дней, прошедших с 1 января 4713 года до н.э.
MI	Минута (0–59)
MM	Месяц (01–12)
MON	Сокращенное название месяца
RM	Номер месяца, записанный римскими цифрами (I–XII)
SS	Секунда (0–59)
SSSSS	Количество секунд, прошедших с полуночи (0–86 399)
SYYY	Четырехзначный номер года; году до нашей эры предшествует знак минус
TS	Время в коротком формате
TZD	Информация о летнем времени (пример: PST или PDT)

Спецификатор формата	Назначение
TZH	Час в смещении часового пояса
TZM	Минута в смещении часового пояса
TZR	Обозначение часового пояса
X	Местный символ, отделяющий целую часть от дробной
Y, *YY* или *YYY*	Год, записанный одной, двумя или тремя цифрами
Y, YYY	Год, в записи которого присутствует запятая
YYYY	Год, записанный четырьмя цифрами

TO_CHAR(number[, format[, nls_parameter]])

Преобразует число *number* в значение типа *VARCHAR2* в формате, определяемом параметром *format*. Если этот параметр опущен, *number* преобразуется в строку, достаточно длинную для представления числа. Параметр *nls_parameter* обеспечивает дополнительный контроль над форматированием. Например:

```
SELECT TO_CHAR(123.45, '$999.99') FROM DUAL -> $123.45
```

TO_CLOB(expr)

Преобразует символьное выражение *expr* в тип данных *CLOB*. Например:

```
SELECT LENGTH(TO_CLOB('I am a SQL nut!')) FROM DUAL -> 15
```

TO_DATE(string[, format[, nls_parameter]])

Преобразует строку *string* (типа *CHAR or VARCHAR2*) в тип данных *DATE*. Параметр *nls_parameter* обеспечивает дополнительный контроль над форматированием. Например:

```
SELECT TO_DATE('15/04/1999', 'DD/MM/YYYY') FROM DUAL
15-APR-1999
```

TO_DSINTERVAL(string[, nls_parameter])

Преобразует символьное выражение *string* в тип данных *INTERVAL DAY TO SECOND*. Параметр *nls_parameter* обеспечивает дополнительный контроль над форматированием. Например:

```
SELECT CURRENT_DATE, CURRENT_DATE-TO_DSINTERVAL('14 00:00:00') FROM DUAL
15-APR-2003 01-APR-2003
```

TO_LOB(long_column)

Преобразует значение типа *LONG* или *LONG RAW* в столбце *long_column* в значение типа *LOB*. Может употребляться только для выражений типа *LONG* или *LONG RAW* и только в списке подзапроса *SELECT* в команде *INSERT*.

TO_MULTI_BYTE(string)

Возвращает строку *string*, в которой все однобайтовые символы преобразованы в соответствующие многобайтовые.

TO_NCHAR(expr[, format[, nls_parameter]])

Синоним *TO_CHAR* с тем отличием, что результат имеет тип *NCHAR*. Допустимые спецификаторы форматы см. в описании функции *TO_CHAR*.

TO_NCLOB(expr)

Преобразует символьное выражение *expr* в тип данных *NCLOB*. Например:

```
SELECT LENGTH(TO_NCLOB('I am a SQL nut!')) FROM DUAL -> 15
```

TO_NUMBER(string[, format[, nls_parameter]])

Преобразует состоящую из цифр строку *string* (типа *CHAR* или *VARCHAR2*) в тип данных *NUMBER*, возможно, с форматированием, которое определяется параметром *format*. Параметр *nls_parameter* обеспечивает дополнительный контроль над форматированием. Например:

```
SELECT TO_NUMBER('12345') FROM DUAL -> 12345
```

TO_SINGLE_BYTE(string)

Возвращает строку *string*, в которой все многобайтовые символы преобразованы в соответствующие однобайтовые.

TO_TIMESTAMP(string[, format[, nls_parameter]])

Преобразует символьное выражение *string* в тип данных *TIMESTAMP*, возможно, с форматированием, которое определяется параметром *format*. Параметр *nls_parameter* обеспечивает дополнительный контроль над форматированием (Информацию о допустимых спецификаторах формата см. в описании функции *TO_CHAR*.) Например:

```
SELECT TO_TIMESTAMP(CURRENT_DATE) FROM DUAL -> 04-MAY-04 12.00.00 AM
```

TO_TIMESTAMP_TZ(string[, format[, nls_parameter]])

Преобразует символьное выражение *string* в тип данных *TIMESTAMP WITH TIME ZONE*, возможно, с форматированием, которое определяется параметром *format*. Параметр *nls_parameter* обеспечивает дополнительный контроль над форматированием. (Информацию о допустимых спецификаторах формата см. в описании функции *TO_CHAR*.) Например:

```
SELECT TO_TIMESTAMP_TZ('15-04-2006', 'DD-MM-YYYY') FROM DUAL
15-APR-06 12.00.00.000000000 AM -07:00
```

TO_YMINTERVAL(string)

Преобразует символьное выражение *string* в тип данных *INTERVAL DAY TO MONTH*. Например:

```
SELECT TO_DATE('29-FEB-2000')+TO_YMINTERVAL('04-00') FROM DUAL
29-FEB-04
```

TRANSLATE(char_value, from_text, to_text)

Возвращает символьное значение *char_value*, в котором все символы, встречающиеся в строке *from_text*, заменены соответствующими символами из строки *to_text*. Например:

```
SELECT TRANSLATE('foobar', 'fa', 'bu') FROM DUAL -> 'boobur'
```

TRANSLATE(text USING [CHAR_CS | NCHAR_CS])

Преобразует строку *text* в указанную кодировку. Если задан параметр *CHAR_CS*, то *text* преобразуется в тип данных *CHAR*, а если *CHAR_CS* – то в тип данных *NCHAR*. Например:

```
SELECT TRANSLATE(N'foobar' USING CHAR_CS) FROM DUAL
'foobar'
```

TREAT(expr AS [REF] [schema.]type)

Преобразует выражение *expr* из объявленного типа в тип, заданный параметром *type*. Дополнительную информацию об использовании этой функции см. в документе Oracle SQL Reference.

TRUNC(base[, number])

Возвращает число *base*, усеченное до *number* десятичных знаков. Если параметр *number* опущен, производится усечение до целого. Значение *number* может быть отрицательным, в этом случае обнуляются цифры слева от запятой. Например:

```
SELECT TRUNC('123.456', 2) FROM DUAL -> 123.45
```

TRUNC(date[, format])

Возвращает дату *date*, усеченную до единицы измерения времени, заданной параметром *format*. Если параметр *format* опущен, производится усечение до ближайшего дня. (Информацию о допустимых спецификаторах формата см. в описании функции *TO_CHAR*.) Например:

```
SELECT TRUNC(TO_DATE('15/04/1999', 'MM/DD/YYYY'), 'YYYY') FROM DUAL
1999
```

TZ_OFFSET({expr | SESSIONTIMEZONE | DBTIMEZONE})

Возвращает смещение часового пояса, указанного в аргументе. Символьное выражение *expr* может содержать название или смещение часового пояса. Значения *SESSIONTIMEZONE* и *DBTIMEZONE* задают часовой пояс, используемый в текущем сеансе или сконфигурированный для базы данных соответственно. Например:

```
SELECT TZ_OFFSET('+08:00'), TZ_OFFSET(SESSIONTIMEZONE) FROM DUAL
+08:00 -07:00
```

UID

Возвращает целое число, уникально идентифицирующее пользователя, владеющего текущим сеансом. Параметры отсутствуют. Например:

```
SELECT UID FROM DUAL -> 47
```

UNISTR(string)

Преобразует строку *string* в тип *NCHAR*, подставляя вместо закодированных символов UNICODE их текстовые значения. Например:

```
SELECT UNISTR('El Ni\00F1o') FROM DUAL -> 'El Nico'
```

UPDATEXML(instance, xpath, expr[, namespace])

Обновляет значения, хранящиеся в узлах экземпляра *instance*, записывая в них новые значения из *expr*. Обновляются только узлы, которые возвращает XPath-запрос *xpath*. Необязательный параметр *namespace* задает пространство имен XML в запросе. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference. Например:

```
SELECT UPDATEXML(XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),
                 '/foo/bar', '<bar>Bye, World!</bar>') from DUAL
Bye, World!
```

USERENV(option)

Возвращает информацию о текущем сеансе в виде строки типа *VARCHAR2*. Эта функция устарела и сохранена только ради совместимости с предыдущими версиями. *USERENV* является синонимом *SYS_CONTEXT('USER_ENV', option)*.

См. описание текущей функциональности в документации по функции *SYS_CONTEXT* с пространством имен *USERENV*. Например:

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL
'AMERICAN_AMERICA.AL32UTF8'
```

VALUE(table_alias)

Принимает псевдоним таблицы *table_alias*, ассоциированной со строкой в объектной таблице, и возвращает экземпляр объекта, хранящийся в объектной таблице для этой строки.

VAR_POP(expression) [OVER (window_clause)]

Вычисляет дисперсию генеральной совокупности, состоящей из набора чисел, представленных выражением *expression*. Значения NULL предварительно отбрасываются. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT VAR_POP(col1) FROM NUMS -> 24.5
```

VAR_SAMP(expression) [OVER (window_clause)]

Возвращает выборочную дисперсию набора чисел, представленных выражением *expression*. Значения NULL предварительно отбрасываются. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT VAR_SAMP(col1) FROM NUMS -> 32.6666667
```

VARIANCE([DISTINCT] expression) [OVER (window_clause)]

Возвращает дисперсию набора чисел, представленного выражением *expression*. Результат вычисляется следующим образом: если количество строк в наборе *expression* равно 1, то возвращается 0, а если больше 1, то возвращается значение функции *VAR_SAMP*. О том, что означает фраза *window_clause*, см. раздел «Оконные функции в ANSI SQL» выше в этой главе. Например:

```
SELECT VARIANCE(col1) FROM NUMS -> 32.6666667
```

VSIZE(expression)

Возвращает количество байтов во внутреннем представлении *expression*. Если *expression* равно NULL, то возвращает NULL. Например:

```
SELECT vsize(1) FROM DUAL -> 2
```

XMLAGG(instance[, order_by])

Агрегатная функция, которая возвращает XML-документ, полученный из таблицы фрагментов XML, заданной параметром *instance*. Необязательная фраза *order_by* позволяет упорядочить фрагменты XML. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLCAST(value AS datatype)

Приводит значение *value* к типу *datatype* и возвращает результат. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLCDATA(value)

Возвращает значение *value* в виде секции XML CDATA. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLCOLATTVAL(expr [AS alias][, ...])

Возвращает фрагмент XML, составленный из аргументов *expr*. Необязательная фраза AS позволяет изменить значение атрибута *name*. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLCONCAT(instance[, ...])

Возвращает экземпляр XML, являющийся объединением параметров *instance*. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLDIFF(xml1, xml2[, hashlevel, flags])

Возвращает Xdiff-документ, описывающий различия между *xml1* и *xml2*. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLELEMENT([NAME] name[, XMLATTRIBUTES(expr [AS alias][, ...])][, value[, ...]])

Возвращает элемент *XMLELEMENT*, имя которого задано параметром *name*, а атрибуты – в необязательной фразе *XMLATTRIBUTES*. В параметрах *value* задаются значения результирующего элемента. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLEXISTS(xquery[, passing_clause])

Возвращает *TRUE*, если XQuery-запрос *xquery* возвращает непустой результат, и *FALSE* – в противном случае. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLFOREST(value [AS alias][, ...])

Возвращает фрагмент XML, составленный из значений, переданных в списке параметров *value*. Необязательная фраза AS позволяет изменить имя объемлющего тега. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLPARSE([DOCUMENT|CONTENT] value [WELLFORMED])

Возвращает экземпляр XML, сконструированный путем разбора XML-документа, переданного в параметре *value*. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLPATCH(xml, xdiff)

Возвращает экземпляр XML, сконструированный в результате применения изменений, описанных в Xdiff-документе, переданном в параметре *xdiff*. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLPI( )

Генерирует инструкцию по обработке XML. Обычно применяется для передачи приложению инструкций, ассоциированных со всем XML-документом или его частью. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLQUERY(query)

Возвращает результат выполнения XMLQuery-запроса *query*. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLROOT(value, VERSION [version | NO VALUE][, STANDALONE {YES | NO | NO VALUE})

Возвращает новый XML-документ, в котором телом является значение *value*, а пролог содержит информацию о версии. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLSEQUENCE(instance)

Возвращает массив фрагментов XML, сконструированный из узлов верхнего уровня экземпляра XML, переданного в параметре *instance*. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLSERIALIZE(...)

Возвращает сериализованное представление XML-выражения в виде строки. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLTABLE(...)

Отображает результат выполнения XQuery-запроса в виде двумерного реляционного представления. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

XMLTRANSFORM(instance, stylesheet)

Возвращает результат примерения XSL-таблицы стилей *stylesheet* к XML-документу *instance*. Дополнительную информацию об XML-запросах см. в документе Oracle SQL Reference.

Функции, поддерживаемые PostgreSQL

В этом разделе приведен алфавитный перечень функций, специфичных для СУБД PostgreSQL, – с примерами.

ABSTIME(timestamp)

Преобразует временную метку *timestamp* к типу *ABSTIME*. Эта функция сохранена только ради обратной совместимости и может быть исключена в будущих версиях. Например:

```
SELECT ABSTIME(CURRENT_TIMESTAMP) -> 2003-06-24 00:19:17-07
```

ACOS(number)

Возвращает арккосинус числа *number*, находящегося в диапазоне от -1 до 1. Результат принимает значения от 0 до π и выражен в радианах. Например:

```
SELECT ACOS( 0 ) -> 1.570796
```

AGE(timestamp)

То же самое, что *AGE(CURRENT_DATE, timestamp)*.

AGE(timestamp, timestamp)

Возвращает интервал времени между двумя временными метками. Например:

```
SELECT AGE( '2003-12-31', CURRENT_TIMESTAMP )
6 mons 7 days 00:34:41.658325
```

AREA(object)

Возвращает площадь геометрического объекта. Например:

```
SELECT AREA( BOX '((0,0),(1,1))' ) -> 1
```

ARRAY_APPEND(array, element)

Возвращает результат добавления элемента *element* в массив *array*. Например:

```
SELECT ARRAY_APPEND(ARRAY[1,2], 3) -> {1,2,3}
```

ARRAY_CAT(array1, array2)

Возвращает результат добавления массива *array2* в конец массива *array1*. Например:

```
SELECT ARRAY_CAT(ARRAY[1,2], ARRAY[3,4]) -> {1,2,3,4}
```

ARRAY_DIMS(array)

Возвращает размерности массива *array*. Например:

```
SELECT ARRAY_DIMS(ARRAY[1,2]) -> '[1:2]'
```

ARRAY_LOWER(array, i)

Возвращает нижнюю границу размерности *i* массива *array*. Например:

```
SELECT ARRAY_LOWER(ARRAY[1,2], 1) -> 1
```

ARRAY_PREPEND(i, array)

Возвращает результат добавления элемента *i* в начало массива *array*. Например:

```
SELECT ARRAY_PREPEND(0, ARRAY[1,2]) -> {0,1,2}
```

ARRAY_TO_STRING(array, delimiter)

Возвращает строку, полученную путем конкатенации всех элементов массива *array*. Параметр *delimiter* задает разделитель элементов. Например:

```
SELECT ARRAY_TO_STRING(ARRAY[1,2,3], ';') -> '1;2;3'
```

ARRAY_UPPER(array, i)

Возвращает верхнюю границу размерности *i* массива *array*. Например:

```
SELECT ARRAY_UPPER(ARRAY[1,2], 1) -> 2
```

ASCII(text)

Возвращает ASCII-код первого символа строки *text*. Например:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Возвращает арксинус числа *number*, находящегося в диапазоне от -1 до 1. Результат принимает значения от $-\pi/2$ до $\pi/2$ и выражен в радианах. Например:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(number)

Возвращает арктангенс произвольного числа *number*. Результат принимает значения от $-\pi/2$ до $\pi/2$ и выражен в радианах. Например:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATAN2(float1, float2)

Возвращает арктангенс числа. Функция *ATAN2(x, y)* аналогична *ATAN(y/x)* с тем отличием, что для определения квадранта, в котором находится результат, используются знаки аргументов *x* и *y*. Например:

```
SELECT ATAN2( 3.1415926, 0 ) -> 1.5707963267949
```

BIT_AND(expression)
BIT_OR(expression)

Возвращает результат применения поразрядной операции И или ИЛИ ко всем отличным от NULL значениям выражения *expression*. Например:

```
SELECT BIT_AND( column1 ) FROM table
0
```

BOOL_AND(expression)
BOOL_OR(expression)

Возвращает результат применения логической операции И или ИЛИ ко всем отличным от NULL значениям выражения *expression*. Например:

```
SELECT BOOL_OR( column1 ) FROM table
true
```

BOX(box, box)

Возвращает прямоугольник *BOX*, являющийся пересечением двух прямоугольников. Если прямоугольники не пересекаются, функция возвращает NULL. Например:

```
SELECT BOX( BOX '((-1,-1),(1,1))', BOX '((0,0),(1,1))' )
(1,1),(0,0)
```

BOX(circle)

Возвращает прямоугольник *BOX* максимального размера, который можно целиком разместить внутри заданной окружности *circle*. Например:

```
SELECT BOX(CIRCLE '((0,0),2.0)')
(1.41421356237309,1.41421356237309),
(-1.41421356237309,-1.41421356237309)
```

BOX(point, point)

Возвращает прямоугольник *BOX*, противоположные вершины которого находятся в заданных точках. Например:

```
SELECT BOX(POINT(0,0), POINT(1,1)) -> (1,1),(0,0)
```

BOX(polygon)

Преобразует многоугольник в прямоугольник *BOX*. Например:

```
SELECT BOX(POLYGON '((0,0),(1,1),(2,0))') -> (2,1),(0,0)
```

BROADCAST(inet)

Конструирует широковещательный адрес в виде символьной строки. Например:

```
SELECT BROADCAST('192.168.1.5/24') -> '192.168.1.255/24'
```

BTRIM(s, c)

Возвращает строку *s*, из которой удалены все символы, встречающиеся в строке *c*. Например:

```
SELECT BTRIM('<<<trim_me>>>', '><') -> 'trim_me'
```

CBRT(float8)

Возвращает кубический корень из числа *float8*. Например:

```
SELECT CBRT(8) -> 2
```

CENTER(object)

Возвращает объект *POINT*, представляющий центр заданного объекта. Например:

```
SELECT CENTER( CIRCLE '((0,0), 2.0)' ) -> (0,0)
```

CHAR(text)

Преобразует *text* в тип *CHAR*.

CHAR_LENGTH(string) или CHARACTER_LENGTH(string)

Возвращает длину строки *string* в символах.

CIRCLE(box)

Возвращает окружность *CIRCLE*, содержащуюся внутри прямоугольника *box*. Например:

```
SELECT CIRCLE(BOX '((0,0),(1,1))') -> <(0.5,0.5),0.707106781186548>
```

CIRCLE(point, float8)

Возвращает окружность *CIRCLE* с центром в точке *point* и радиусом *float8*. Например:

```
SELECT CIRCLE(POINT '(0,0)', 2.0) -> <(0,0),2>
```

CLOCK_TIMESTAMP( )

Возвращает текущую дату и время, которые могут измениться в процессе выполнения одной команды SQL.

COALESCE(list)

Возвращает первый отличный от NULL элемент списка *list*. Например:

```
SELECT COALESCE(NULL,1,2,3,NULL) -> 1
```

COS(number)

Возвращает косинус числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT COS(0) -> 1.000000
```

COT(number)

Возвращает котангенс числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT COT( 3.1415 ) -> -10792.88993953
```

CURRVAL(s)

Возвращает текущее значение последовательности с именем *s*. Например:

```
SELECT CURRVAL( 'myseq' ) -> 99
```

DATE_PART(text, value)

То же самое, что *EXTRACT(text, value)*; дополнительную информацию см. в описании функции *EXTRACT* в разделе «Скалярные функции ANSI SQL» выше.

DATE_TRUNC(precision, timestamp)

Усекает временную метку *timestamp* до указанной точности *precision*. Например:

```
SELECT DATE_TRUNC('hour', TIMESTAMP '2003-04-15 23:58:30')
2003-04-15 23:00:00
```

DECODE(s, type)

Декодирует закодированную строку *s*. Параметр *type* может принимать значения '*base64*', '*hex*' или '*escape*'. Например:

```
SELECT DECODE( ENCODE('darjeeling', 'base64'), 'base64' ) -> 'darjeeling'
```

DEGREES(float8)

Переводит радианы в градусы. Например:

```
SELECT DEGREES( 3.1415926 ) -> 179.999996929531
```

DIAMETER(circle)

Возвращает диаметр окружности *circle*. Например:

```
SELECT DIAMETER(CIRCLE(POINT '(0,0)', 2.0)) -> 4
```

ENCODE(s, type)

Кодирует строку *s*. Параметр *type* может принимать значения *'base64'*, *'hex'* или *'escape'*. Например:

```
SELECT DECODE( ENCODE('darjeeling', 'base64'), 'base64' ) -> 'darjeeling'
```

EVERY(expression)

Синоним *BOOL_AND(expression)*.

FLOAT(int)

Преобразует целое число *int* в число с плавающей точкой.

FLOAT4(int)

Преобразует целое число *int* в число с плавающей точкой.

HEIGHT(box)

Возвращает длину вертикальной стороны прямоугольника *box*. Например:

```
SELECT HEIGHT(BOX '((0,0),(1,1))') -> 1
```

HOST(inet)

Возвращает адрес хоста в виде текста. Например:

```
SELECT HOST('192.168.1.5/24') -> '192.168.1.5'
```

INITCAP(text)

Переводит первую букву каждого слова в верхний регистр. Например:

```
SELECT INITCAP( 'my name is inigo montoya.' )
'My Name Is Inigo Montoya.'
```

INTEGER(float)

Преобразует число с плавающей точкой в целое.

INTERVAL(reltime)

Преобразует относительное время *reltime* в тип *INTERVAL*.

ISCLOSED(path)

Возвращает 't', если путь *path* замкнут, иначе 'f'. Например:

```
SELECT ISCLOSED(PATH '((0,0),(1,1),(2,0))') -> 't'
SELECT ISCLOSED(PATH '[(0,0),(1,1),(2,0)]') -> 'f'
```

ISFINITE(interval)

Возвращает 'f', если интервал *interval* открыт, иначе 't'. Например:

```
SELECT ISFINITE(INTERVAL '4 hours') -> 't'
```

ISFINITE(timestamp)

Возвращает 'f', если временная метка *timestamp* некорректна или представляет бесконечность, иначе 't'. Например:

```
SELECT ISFINITE(TIMESTAMP '2001-02-16 21:28:30') -> 't'
```

ISOPEN(path)

Возвращает 't', если путь *path* не замкнут, иначе 'f'. Например:

```
SELECT ISOPEN(PATH '((0,0),(1,1),(2,0))') -> 'f'
SELECT ISOPEN(PATH '[(0,0),(1,1),(2,0)]') -> 't'
```

JUSTIFY_DAYS(interval)

Позволяет исключить переполнение количества дней (переполнение дней происходит, если их количество больше 30). В примере функция обрабатывает «нетипичную» ситуацию с количеством дней и преобразует представление к нормализованному виду: количество дней не более 30. Например:

```
select justify_days(interval ' 1 year 9 months 61 days 25 hours ') -> 1 year 11 mons
1 day 25:00:00
```

JUSTIFY_HOURS(interval)

Работает аналогично *JUSTIFY_DAYS*, но нормализует часы, а не дни. При этом ненормализованные дни не затрагиваются. Например:

```
SELECT JUSTIFY_HOURS(INTERVAL ' 1 year 9 months 61 days 25 hours') -> 1 year 9 mons
62 days 01:00:00
```

JUSTIFY_INTERVAL(interval)

Совмещает в себе возможности функций *JUSTIFY_DAYS* и *JUSTIFY_HOURS*, т. е. нормализует как дни, так и часы. Например:

```
SELECT JUSTIFY_INTERVAL (INTERVAL ' 1 year 9 months 61 days 25 hours') -> 1 year 11
mons 2 days 01:00:00
```

LENGTH(object)

Возвращает длину объекта. Например:

```
SELECT LENGTH('Howdy!') -> 6
SELECT LENGTH(PATH '((-1,0),(1,0))') -> 4
```

LOG(float8[, b])

Возвращает логарифм числа *float8* по основанию *b*. Если параметр *b* не задан, возвращает десятичный логарифм. Например:

```
SELECT LOG( 100 ) -> 2
```

LPAD(exp1, int, exp2)

Возвращает строку *exp1*, дополненную слева строкой *exp2* до длины *int*. Например:

```
SELECT LPAD('Duck', 10, 's') -> 'ssssssDuck'
```

LSEG(box)

Возвращает диагональ прямоугольника *box* в виде отрезка прямой. Например:

```
SELECT LSEG(BOX '((-1,0),(1,0))') -> [(1,0),(-1,0)]
```

LSEG(point, point)

Возвращает отрезок прямой с концами в указанных точках. Например:

```
SELECT LSEG(POINT '(-1,0)', POINT '(1,0)') -> [(-1,0),(1,0)]
```

LTRIM(text)

Возвращает строку *text*, из которой удалены начальные пробелы. Например:

```
SELECT LTRIM(' Howdy! ') -> 'Howdy! '
```

MASKLEN(cidr)

Возвращает длину сетевой маски *cidr*. Например:

```
SELECT MASKLEN('192.168.1.5/24') -> 24
```

MD5(s)

Возвращает MD5-свертку *s*.

NETMASK(inet)

Возвращает сетевую маску для IP-адреса *inet*. Например:

```
SELECT NETMASK('192.168.1.5/24') -> '255.255.255.0'
```

NETWORK(inet)

Возвращает сетевую часть IP-адреса *inet*. Например:

```
SELECT NETWORK('192.168.1.5/24') -> '192.168.1.0/24'
```

NEXTVAL(s)

Возвращает следующее число в последовательности с именем *s*. Например:

```
SELECT NEXTVAL('myseq') -> 100
```

NPOINTS(object)

Возвращает количество точек в описании объекта *object*. Например:

```
SELECT NPOINTS(POLYGON '((1,1),(0,0))') -> 2
```

NULLIF(input, value)

Возвращает NULL, если *input* = *value*, в противном случае *input*. Например:

```
SELECT NULLIF( 5, 6 ), NULLIF( 5, 5 )
5 NULL
```

PATH(polygon)

Преобразует многоугольник *polygon* в путь. Например:

```
SELECT PATH( '((0,0),(1,1),(2,0))' )
((0,0),(1,1),(2,0))
```

PCLOSE(path)

Преобразует путь *path* в замкнутый путь. Например:

```
SELECT PCLOSE(PATH '[(0,0),(1,1),(2,0)]')
((0,0),(1,1),(2,0))
```

PI( )

Возвращает значение числа π.

POINT(circle)

Возвращает центр окружности *circle*. Например:

```
SELECT POINT( CIRCLE '((0,0), 2.0)' ) -> (0,0)
```

POINT(lseg1, lseg2)

Возвращает точку пересечения двух отрезков прямой. Например:

```
SELECT POINT(LSEG '((-1,0),(1,0))', LSEG '((-2,-2),(2,2))')
(0,0)
```

POINT(polygon)

Возвращает центр многоугольника *polygon*. Например:

```
SELECT POINT(POLYGON '((0,0),(1,1),(2,0))')
(1,0.333333333333333)
```

POLYGON(path)

Преобразует путь *path* в многоугольник. Например:

```
SELECT POLYGON(PATH '((0,0),(1,1),(2,0))')
((0,0),(1,1),(2,0))
```

POLYGON(box)

Преобразует прямоугольник *box* в многоугольник. Например:

```
SELECT POLYGON(BOX '((0,0),(1,1))')
((0,0),(0,1),(1,1),(1,0))
```

POLYGON(circle)

Синоним *POLYGON(12, circle)*.

POLYGON(npts, circle)

Возвращает аппроксимацию окружности *circle* многоугольником с *npts* вершинами. Например:

```
SELECT POLYGON(6, CIRCLE '((0,0),2.0)')
((-2,0),
(-0.999999999994107,1.73205080757228),
(1.00000000001179,1.73205080756207),
(2,-2.04136478690279e-11),
(0.999999999976428,-1.73205080758249),
(-1.00000000002946,-1.73205080755187))
```

POPEN(path)

Преобразует путь *path* в открытый путь. Например:

```
SELECT POPEN(PATH '((0,0),(1,1),(2,0))')
[(0,0),(1,1),(2,0)]
```

POW(number, exponent)

Возводит число *number* в степень *exponent*. Например:

```
SELECT POW( 2, 3 ) -> 8
```

QUOTE_IDENT(s)

Возвращает строку *s*, в которой специальные символы экранированы так, чтобы значение можно было безопасно использовать в качестве идентификатора в командах SQL. Например:

```
SELECT QUOTE_IDENT( 'tea' ) -> '"tea"'
```

QUOTE_LITERAL(s)

Возвращает строку *s*, в которой специальные символы экранированы так, чтобы значение можно было безопасно использовать в качестве строкового литерала в командах SQL. Например:

```
SELECT QUOTE_LITERAL( 'you\'re here' ) -> 'you''re here'
```

RADIANS(float8)

Преобразует градусы в радианы. Например:

```
SELECT RADIANS( 180 ) -> 3.14159265358979
```

RADIUS(circle)

Возвращает радиус окружности *circle*. Например:

```
SELECT RADIUS( CIRCLE '((0,0), 2.0)' ) -> 2.0
```

RANDOM()

Возвращает случайное число в диапазоне от 0.0 до 1.0. Например:

```
SELECT RANDOM( ) -> 0.785398163397448
```

REGEXP_MATCHES(s, pattern[, flags])

Возвращает подстроки строки *s*, соответствующие регулярному выражению *pattern*. Необязательный аргумент *flags* задает параметры алгоритма сопоставления. Чаще всего употребляются флаги 'i' (сравнение без учета регистра), 'n' (сопоставление с учетом символа новой строки) и 'g' (глобальный поиск). Например:

```
SELECT REGEXP_MATCHES('catfish, cowfish', 'c..') -> 'cat'
```

REGEXP_REPLACE(s, pattern, replacement[, flags])

Возвращает строку *s*, в которой все подстроки, соответствующие регулярному выражению *pattern*, заменены строкой *replacement*. Необязательный аргумент *flags* задает параметры алгоритма сопоставления. Чаще всего употребляются флаги 'i' (сравнение без учета регистра), 'n' (сопоставление с учетом символа новой строки) и 'g' (глобальный поиск). Например:

```
SELECT REGEXP_REPLACE('abcabcabc', 'bc', 'nt') -> 'antabcabc'
SELECT REGEXP_REPLACE('abcabcabc', 'bc', 'nt', 'g') -> 'antantant'
```

REGEXP_SPLIT_TO_ARRAY(s, pattern[, flags])

Разбивает строку *s*, используя регулярное выражение *pattern* в качестве разделителя, и возвращает массив полученных подстрок. Необязательный аргумент *flags* задает параметры алгоритма сопоставления. Чаще всего употребляются флаги 'i' (сравнение без учета регистра), 'n' (сопоставление с учетом символа новой строки) и 'g' (глобальный поиск). Например:

```
SELECT REGEXP_SPLIT_TO_ARRAY('a b c', E'\\s+')
{a,b,c}
```

REGEXP_SPLIT_TO_TABLE(s, pattern[, flags])

Разбивает строку *s*, используя регулярное выражение *pattern* в качестве разделителя, и возвращает таблицу, содержащую полученные подстроки. Необязательный аргумент *flags* задает параметры алгоритма сопоставления. Чаще всего употребляются флаги 'i' (сравнение без учета регистра), 'n' (сопоставление с учетом символа новой строки) и 'g' (глобальный поиск). Например:

```
SELECT REGEXP_SPLIT_TO_TABLE('a b c', E'\\s+')
'a'
'b'
'c'
```

RELTIME(interval)

Преобразует интервал *interval* в объект типа *RELTIME*. Сохранена только ради обратной совместимости и может быть исключена в будущих версиях.

ROUND(number[, p])

Округляет число *number* до *p* знаков после запятой. Необязательный аргумент *p* по умолчанию равен 0, что соответствует классическому округлению до целого. Например:

```
SELECT ROUND( 5.5 ) -> 6
SELECT ROUND( 5.5555, 2 ) -> 5.56
```

RPAD(text, length, char)

Возвращает строку *text*, дополненную слева символом *char* до длины *length*. Например:

```
SELECT RPAD('Duck', 10, 's') -> 'Ducksssssss'
```

RTRIM(text)

Возвращает строку *text*, из которой удалены конечные пробелы. Например:

```
SELECT RTRIM(' St. Lucia ') -> ' St. Lucia'
```

SET_MASKLEN(inet, size)

Устанавливает длину сетевой маски для IP-адреса *inet* равной *size*. Например:

```
SELECT SET_MASKLEN('192.168.1.5/24',16)
'192.168.1.5/16'
```

SETSEED(i)

Инициализирует генератор случайных чисел числом *i*.

SETVAL(s, i)

Устанавливает следующее число в последовательности с именем *s* равным *i*. Например:

```
SETVAL('myseq', 0)
```

SIGN(number)

Возвращает знак числа *number*. Например:

```
SELECT SIGN( -69 ), SIGN(69)
-1,1
```

SIN(number)

Возвращает синус числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT SIN( 0 ) -> 0.000000
```

STATEMENT_TIMESTAMP( )

Возвращает дату и время начала выполнения SQL-команды.

STRING_TO_ARRAY(str1, delimiter)

Разбивает строку *str1* по разделителю *delimiter* и возвращает массив, составленный из образовавшихся подстрок. Например:

```
SELECT STRING_TO_ARRAY('1;2;3', ';') -> {1,2,3}
```

SUBSTRING(string [FROM start] [FOR bytes]), SUBSTR(string, start[, bytes])

Возвращает подстроку строки *string* длиной *bytes* байтов, начинающуюся в позиции *start*. Если параметр *bytes* опущен, то подстрока продолжается до конца строки. Например:

```
SELECT SUBSTRING( 'Inigo Montoya' FROM 7 FOR 4 ) -> 'Mont'
```

TAN(number)

Возвращает тангенс числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT TAN( 3.1415 ) -> -0.000093
```

TEXT(char)

Преобразует *char* к типу *TEXT*.

TIMEOFDAY( )

Возвращает текущую временную метку (полученную с помощью функции *CLOCK_TIMESTAMP*) в виде строки.

TIMESTAMP(date[, time])

Преобразует дату во временную метку.

TO_CHAR(expression, text)

Преобразует выражение *expression* в строку. Например:

```
SELECT TO_CHAR(NUMERIC '-125.8', '999D99S') -> 125.80-
SELECT TO_CHAR (interval '15h 2m 12s','HH24:MI:SS') -> 15:02:12
```

TO_DATE(string, format)

Преобразует строку *string* в дату. Второй аргумент определяет формат входной строки. Например:

```
SELECT TO_DATE('05 Dec 2000', 'DD Mon YYYY') -> 2000-12-05
```

В табл. 4.12 перечислены допустимые спецификаторы формата.

Таблица 4.12. Спецификаторы формата в PostgreSQL

Спецификатор формата	Назначение
AD или *A.D.*	Индикатор новой эры
AM или *A.M.*	Индикатор полудня
BC или *B.C.*	Индикатор старой эры (до Рождества Христова)
CC	Год, записанный двумя цифрами
D	День недели (1–7)
DAY	Полное название дня недели, записанное заглавными буквами
Day	Полное название дня недели с заглавной буквы
day	Полное название дня недели, записанное строчными буквами
DD	День месяца (01–31)
DDD	Порядковый номер дня в году (001–366)
DY	Сокращенное название дня с заглавной буквы
Dy	Сокращенное название дня, записанное заглавными буквами
dy	Сокращенное название дня, записанное строчными буквами
HH или *HH12*	Час (01–12)
HH24	Час (0–23)
I	Последние цифры года согласно стандарту ISO
IW	Порядковый номер недели в году согласно стандарту ISO
IY, IYY, IYYY	Последние две, три или все четыре цифры года согласно стандарту ISO
J	Юлианский день; количество дней, прошедших с 1 января 4713 года до н.э.
MI	Минута (00–59)
MM	Месяц (01–12)
MON	Сокращенное название месяца, записанное заглавными буквами

Спецификатор формата	Назначение
Mon	Сокращенное название месяца с заглавной буквы
mon	Сокращенное название месяца, записанное строчными буквами
MONTH	Полное название месяца, записанное заглавными буквами
Month	Полное название месяца с заглавной буквы
month	Полное название месяца, записанное строчными буквами
MS	Миллисекунды (0–999)
PM или *P.M.*	Индикатор полудня
Q	Квартал года
RM	Номер месяца, записанный римскими цифрами (I–XII)
rm	Номер месяца, записанный римскими цифрами (I–xii)
SS	Секунда (00–59)
SSSS	Количество секунд, прошедших с полуночи (0–86399)
SYYY	Год, записанный четырьмя цифрами; годам до нашей эры предшествует знак минус
TS	Время в коротком формате
TZ	Название часового пояса, записанное заглавными буквами
tz	Название часового пояса, записанное строчными буквами
US	Микросекунды (000000–999999)
W	Неделя месяца (1–5)
WW	Неделя года (1–53)
Y, YY, YYY или *YYYY*	Год, записанный одной, двумя, тремя или четырьмя цифрами
Y,YYY	Год, в записи которого присутствует запятая

TO_NUMBER(`string`, `format`)

Преобразует строку *`string`* в числовое значение. Второй аргумент задает формат входной строки. Например:

```
SELECT TO_NUMBER('12,454.8-', '99G999D9S') -> -12454.8
```

TO_TIMESTAMP(`text`, `format`)

Преобразует строку *`text`* во временную метку. Второй аргумент задает формат входной строки. (О допустимых спецификаторах формата см. описание функции *TO_DATE*.) Например:

```
SELECT TO_TIMESTAMP('05 Dec 2000', 'DD Mon YYYY') ->
2000-12-05 00:00:00-08
```

TRANSACTION_TIMESTAMP( )

Возвращает дату и время начала выполнения текущей транзакции.

TRANSLATE(text, from, to)

Преобразует символы текста *text*, которые встречаются в строке *from*, в соответствующие символы из строки *to*. Например:

```
SELECT TRANSLATE('foo', 'fo', 'ab') -> 'abb'
```

TRUNC(float8)

Отбрасывает все знаки после запятой. Например:

```
SELECT TRUNC( PI( ) ) -> 3
```

VARCHAR(string)

Преобразует строку *string* в тип *VARCHAR*.

WIDTH(box)

Возвращает ширину прямоугольника *box*. Например:

```
SELECT WIDTH( BOX '((0,0),(3,1))' ) -> 3
```

Функции, поддерживаемые SQL Server

В этом разделе приведен алфавитный перечень функций, поддерживаемых на платформе SQL Server, – с примерами.

ACOS(number)

Возвращает арккосинус числа *number*, находящегося в диапазоне от -1 до 1. Результат принимает значения от 0 до π и выражен в радианах. Например:

```
SELECT ACOS( 0 ) -> 1.570796
```

APP_NAME( )

Возвращает имя приложения, открывшего текущий сеанс. Например:

```
SELECT APP_NAME( ) -> 'SQL Enterprise Manager'
```

ASCII(text)

Возвращает ASCII-код первого символа строки *text*. Например:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Возвращает арксинус числа *number*, находящегося в диапазоне от -1 до 1. Результат принимает значения от -π/2 до π/2 и выражен в радианах. Например:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(number)

Возвращает арктангенс произвольного числа *number*. Результат принимает значения от -π/2 до π/2 и выражен в радианах. Например:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATN2(float1, float2)

Возвращает угол (в радианах), тангенс которого равен *float1*/*float2*. Например:

```
SELECT ATN2( 35.175643, 129.44 ) -> 0.265345
```

BINARY_CHECKSUM(|expression[, ...n])*

Возвращает двоичное значение контрольной суммы, вычисленное для списка выражений или строки таблицы. В следующем примере возвращается список идентификаторов пользователей, для которых сохраненная контрольная сумма пароля не совпадает с текущей:

```
SELECT userid AS 'Changed' FROM users
WHERE NOT password_chksum = BINARY_CHECKSUM( password )
```

CHAR(integer_expression)

Преобразует ASCII-код в символ. Например:

```
SELECT CHAR( 78 ) -> 'N'
```

CHARINDEX(substring, string[, start_location])

Возвращает начальную позицию первого вхождения подстроки *substring* в строку *string*. Если задан необязательный аргумент *start_location*, то поиск начинается с указанной в нем позиции. Например:

```
SELECT CHARINDEX( 'he', 'Howdy, there!' ) -> 9
```

CHECKSUM(|expression[, ...n])*

Возвращает значение контрольной суммы, вычисленное для строки таблицы или списка выражений. В следующем примере возвращается список идентификаторов пользователей, для которых сохраненная контрольная сумма пароля не совпадает с текущей:

```
SELECT userid AS 'Changed' FROM users
WHERE NOT password_chksum = CHECKSUM( password )
```

CHECKSUM_AGG([ALL|DISTINCT] expression)

Возвращает контрольную сумму значений в группе. Например:

```
SELECT CHECKSUM_AGG( BINARY_CHECKSUM(*) ) FROM authors -> 67
```

COALESCE(expression[, ...n])

Возвращает первое отличное от NULL выражение в списке аргументов. Например:

```
SELECT COALESCE( NULL, 1, 3, 5, 7 ) -> 1
```

COL_LENGTH(table, column)

Возвращает длину столбца *column* в байтах. Например:

```
SELECT COL_LENGTH('authors', 'au_fname') -> 50
```

COL_NAME(table_id, column_id)

Возвращает имя столбца по идентификатору содержащей его таблицы *table_id* и идентификатору самого столбца *column_id*. Например:

```
SELECT COL_NAME( OBJECT_ID('authors'), 1 )
```

CONTAINS({column|}, contains_search_condition)*

Ищет в столбце *column* точное или «нечеткое» совпадение с условием *contains_search_condition*. Функция *CONTAINS* применяется для выполнения полнотекстового поиска; дополнительную информацию см. в официальной документации. В следующем примере возвращаются идентификаторы всех товаров в таблице **products**, в названиях которых слова «peanut» и «butter» расположены недалеко друг от друга.

```
SELECT productid FROM products
WHERE CONTAINS(productname, ' "peanut" NEAR "butter" ' )
```

CONTAINSTABLE(table, column, contains_search_condition)

Возвращает таблицу, содержащую данные, которые точно или «нечетко» соответствуют условию *contains_search_condition*. Функция *CONTAINSTABLE* применяется для выполнения полнотекстового поиска.

В следующем примере запрос осуществляет поиск вакансий, в которых встречаются слова computer и California, при этом для каждого слова задается определенный вес. Для каждой строки набора результатов, удовлетворяющей условию поиска, отображается относительная «близость» к совпадению (ранг). Кроме того, строки с более высоким рангом возвращаются первыми.

```
SELECT job_title, job_description, KEY_TBL.RANK
  FROM jobs
  INNER JOIN
  CONTAINSTABLE(jobs, *, 'ISABOUT(computer weight (.8),
    California weight (.4))', 10) AS KEY_TBL
  ON jobs.job_id = KEY_TBL.[KEY]
ORDER BY KEY_TBL.RANK DESC
```

CONVERT(data_type[(length)], expression[, style])

Преобразует выражение *expression* из одного типа данных в другой. Например:

```
SELECT CONVERT( VARCHAR(50), CURRENT_TIMESTAMP, 1 ) -> '06/29/03'
```

COS(number)

Возвращает косинус числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT COS(0) -> 1.000000
```

COT(number)

Возвращает котангенс числа *number*. Например:

```
SELECT COT( 3.1415 ) -> -10792.88993953
```

COUNT_BIG({[ALL|DISTINCT] expression}|)*

Аналогична функции *COUNT*, но возвращает результат типа *BIGINT*. Например:

```
SELECT COUNT_BIG( names ) FROM people -> 26743
```

DATABASEPROPERTYEX(database, property)

Возвращает текущее значение заданного параметра или свойства базы данных. Например:

```
SELECT DATABASEPROPERTYEX('pubs', 'Version') -> 539
```

DATALENGTH(expression)

Возвращает количество байтов в символьной или двоичной строке. Например:

```
SELECT MAX( DATALENGTH( au_fname ) ) FROM authors -> 11
```

DATEADD(datepart, number, date)

Прибавляет к дате *date number* единицы времени *datepart* (например, дней). Например:

```
SELECT DATEADD( Year, 10, CURRENT_TIMESTAMP ) -> 2013-06-29 19:47:15.270
```

DATEDIFF(datepart, startdate, enddate)

Вычисляет разность между двумя выражениями типа *DATETIME*, выраженную в единицах времени *datepart*. Например:

```
SELECT DATEDIFF( Day, CURRENT_TIMESTAMP,
DATEADD( Year, 1, CURRENT_TIMESTAMP ))
366
```

DATENAME(datepart, date)

Возвращает название единицы времени *datepart* (например, месяца) для даты *date*. Например:

```
SELECT DATENAME(month, GETDATE( )) -> 'June'
```

DATEPART(datepart, date)

Возвращает значение единицы времени *datepart* (например, года) для даты *date*. Например:

```
SELECT DATEPART(year, GETDATE( )) -> 2003
```

DAY(date)

Возвращает целое число, представляющее день указанной даты *date*. Например:

```
SELECT DAY('04/15/2004') -> 15
```

DB_ID([database_name])

Возвращает идентификатор базы данных с именем *database_name*. Например:

```
SELECT DB_ID( ) -> 5
```

DB_NAME(database_id)

Возвращает имя базы данных с идентификатором *database_id*. Например:

```
SELECT DB_NAME( 5 ) -> 'pubs'
```

DEGREES(numeric_expression)

Преобразует радианы в градусы. Например:

```
SELECT DEGREES( PI( ) ) -> 180
```

DIFFERENCE(character_expression, character_expression)

Возвращает разницу между значениями SOUNDEX двух символьных выражений. Результатом является целое число от 0 до 4; чем оно больше, тем лучше фонетическое сходство. Например:

```
SELECT DIFFERENCE( 'moe', 'low' ) -> 3
```

FILE_ID(file_name)

Возвращает идентификатор файла с логическим именем *file_name*. Например:

```
SELECT FILE_ID( 'master' ) -> 1
```

FILE_NAME(file_id)

Возвращает логическое имя файла с идентификатором *file_id*. Например:

```
SELECT FILE_NAME( 1 ) -> 'master'
```

FILEGROUP_ID(filegroup_name)

Возвращает идентификатор группы файлов с логическим именем *filegroup_name*. Например:

```
SELECT FILEGROUP_ID( 'PRIMARY' ) -> 1
```

FILEGROUP_NAME(filegroup_id)

Возвращает логическое имя группы файлов с идентификатором *filegroup_id*. Например:

```
SELECT FILEGROUP_NAME( 1 ) -> 'PRIMARY'
```

FILEGROUPPROPERTY(filegroup_name, property)

Возвращает значение свойства *property* указанной группы файлов. Например:

```
SELECT FILEGROUPPROPERTY( 'PRIMARY', 'IsReadOnly' ) -> 0
```

FILEPROPERTY(file, property)

Возвращает значение свойства *property* указанного файла. Например:

```
SELECT FILEPROPERTY( 'pubs', 'SpaceUsed' ) -> 160
```

FORMATMESSAGE(msg_number, param_value[, ... n])

Конструирует сообщение на основе сообщения, уже имеющегося в таблице *SYS.MESSAGES* (аналогично *RAISEERROR*). Например:

```
sp_addmessage 50001, 1, 'Table %s has %s rows.'
SELECT FORMATMESSAGE(50001, 'AUTHORS', (SELECT COUNT(*) FROM AUTHORS) )
'Table AUTHORS has 23 rows.'
```

FREETEXT({column|}, freetext_string)*

Применяется для полнотекстового поиска. Возвращает строки, для которых значение в столбце *column* соответствует строке *freetext_string* по смыслу, но не обязательно дословно.

FREETEXTTABLE(table, {column|}, freetext_string[, top_n_by_rank])*

Применяется для полнотекстового поиска. Возвращает таблицу, включающую те строки из таблицы *table*, для которых значение в столбце *column* соответствует строке *freetext_string* по смыслу, но не обязательно дословно. Например:

```
SELECT * from FREETEXTTABLE (authors, *, 'kev')
```

FULLTEXTCATALOGPROPERTY(catalog_name, property)

Возвращает сведения о свойствах полнотекстового каталога. Например:

```
SELECT FULLTEXTCATALOGPROPERTY( 'Cat_Desc', 'LogSize' )
```

FULLTEXTSERVICEPROPERTY(property)

Возвращает информацию, связанную со свойствами механизма полнотекстового поиска. Например:

```
SELECT FULLTEXTSERVICEPROPERTY('IsFulltextInstalled') -> 1
```

GETANSINULL([database])

Возвращает параметр допустимости значений NULL, используемый по умолчанию для базы данных, при создании новых столбцов в текущем сеансе. Например:

```
SELECT GETANSINULL( ) -> 1
```

GETDATE( )

Возвращает текущую дату и время. Например:

```
SELECT GETDATE( ) -> 2003-06-27 19:26:59.893
```

GETUTCDATE( )

Возвращает текущую дату и время в формате UTC (универсальное скоординированное время). Например:

```
SELECT GETUTCDATE( ) -> 2003-06-28 02:26:46.720
```

GROUPING(column_name)

Возвращает 1, если строка была добавлена в результате выполнения операции *CUBE* или *ROLLUP*, иначе 0. Например:

```
SELECT royalty, SUM(advance) 'total advance', GROUPING(royalty) 'grp'
FROM titles GROUP BY royalty WITH ROLLUP
royalty   total advance         grp
--------- --------------------- ---
NULL      NULL                  0
10        57000.0000            0
12        2275.0000             0
14        4000.0000             0
16        7000.0000             0
24        25125.0000            0
NULL      95400.0000            1
```

HOST_ID( )

Возвращает идентификатор рабочей станции. Например:

```
SELECT HOST_ID( ) -> 216
```

HOST_NAME( )

Возвращает имя рабочей станции. Например:

```
SELECT HOST_NAME( ) -> 'PLATO'
```

IDENT_CURRENT(table_name)

Возвращает последнее значение в столбце идентификаторов, сгенерированное для указанной таблицы. Например:

```
SELECT IDENT_CURRENT('jobs') -> 876
```

IDENT_INCR(table_or_view)

Возвращает значение приращения для столбца идентификаторов. Например:

```
SELECT IDENT_INCR('jobs') -> 1
```

IDENT_SEED(table_or_view)

Возвращает начальное значение для столбца идентификаторов. Например:

```
SELECT IDENT_SEED('jobs') -> 1
```

IDENTITY(data_type[, seed, increment]) AS column_name

Используется только в команде *SELECT INTO* для вставки столбца идентификаторов в новую таблицу. Например:

```
SELECT IDENTITY(int, 1,1) AS IDINTO NewTableFROM OldTable
```

INDEX_COL(table, index_id, key_id)

Возвращает имя индексированного столбца, зная имя таблицы, идентификатор индекса и порядковый номер ключевого столбца индекса. Например:

```
SELECT INDEX_COL(OBJECT_ID('authors'), 1, 1) -> NULL
```

INDEXPROPERTY(table_id, index, property)

Возвращает указанное свойство индекса (например, *FILLFACTOR*). Например:

```
SELECT INDEXPROPERTY(OBJECT_ID('authors'), 'UPKCL_auidind', 'IsPadIndex')
0
```

IS_MEMBER({group|role})

Возвращает true или false (1 или 0) в зависимости от того, является пользователь членом указанной группы *group* Windows NT или роли *role* базы данных SQL Server. Например:

```
SELECT IS_MEMBER( 'db_owner' ) -> 0
```

IS_SRVROLEMEMBER(role[, login])

Возвращает true или false (1 или 0) в зависимости от того, является пользователь членом указанной роли *role* сервера. Например:

```
SELECT IS_SRVROLEMEMBER( 'sysadmin' ) -> 0
```

ISDATE(expression)

Проверяет, можно ли преобразовать указанную символьную строку в тип *DATETIME*. Например:

```
SELECT ISDATE(NULL), ISDATE(GETDATE( ))
0      1
```

ISNULL(check_expression, replacement_value)

Возвращает первый аргумент, если он отличен от NULL, иначе второй аргумент. Например:

```
SELECT ISNULL( NULL, 'NULL' ) -> 'NULL'
```

ISNUMERIC(expression)

Проверяет, можно ли преобразовать указанную символьную строку в тип *NUMERIC*. Например:

```
SELECT ISNUMERIC('3.1415'), ISNUMERIC('IRK')
1      0
```

LEFT(character_expression, integer_expression)

Возвращает первые слева *integer_expression* символов символьного выражения *character_expression*. Например:

```
SELECT LEFT( 'Wet Paint', 3 ) -> 'Wet'
```

LEN(string_expression)

Возвращает количество символов в указанном строковом выражении. Например:

```
SELECT LEN( 'Wet Paint' ) -> 9
```

LOG(float_expression)

Возвращает натуральный логарифм. Например:

```
SELECT LOG( PI( ) ) -> 1.1447298858494002
```

LOG10(float_expression)

Возвращает десятичный логарифм. Например:

```
SELECT LOG10( PI( ) ) -> 0.49714987269413385
```

LTRIM(character_expression)

Удаляет начальные пробелы. Например:

```
SELECT LTRIM(' beaucoup ') -> 'beaucoup '
```

MONTH(date)

Возвращает целое число, представляющее месяц указанной даты *date*. Например:

```
SELECT MONTH( GETDATE( ) ) -> 6
```

NCHAR(integer_expression)

Возвращает символ UNICODE с указанным целочисленным кодом. Например:

```
SELECT NCHAR(120) -> 'x'
```

NEWID( )

Генерирует новый уникальный идентификатор типа *UNIQUEIDENTIFIER*. Например:

```
SELECT NEWID( ) -> '32B35185-F55E-4FE0-B2C8-B57B35815C12'
```

NULLIF(expression, expression)

Возвращает NULL, если оба указанных выражения равны. Например:

```
SELECT NULLIF( 5, 5 ) -> NULL
```

OBJECT_ID(object)

Возвращает идентификатор объекта *object*. Например:

```
SELECT OBJECT_ID('authors')  -> 8
```

OBJECT_NAME(object_id)

Возвращает имя объекта с указанным идентификатором. Например:

```
SELECT OBJECT_NAME ( OBJECT_ID('authors') ) -> 'authors'
```

OBJECTPROPERTY(id, property)

Возвращает свойства объектов в текущей базе данных. Например:

```
SELECT OBJECTPROPERTY ( object_id('authors'),'ISTABLE') -> 1
```

OPEN {{[GLOBAL] cursor_name}|cursor_variable_name}

Открывает локальный или глобальный курсор.

OPENDATASOURCE(provider_name, init_string)

Открывает соединение с источником данных, не используя имя связанного сервера (linked server). Примеры см. в разделе «Загрузчики» руководства пользователя по SQL Server.

OPENQUERY(linked_server, query)

Передает указанный запрос источнику данных, который ранее был сконфигурирован как связанный сервер. Примеры см. в разделе «Загрузчики» руководства пользователя по SQL Server.

OPENROWSET(provider_name, {datasource, user_id, password | provider_string}, {[catalog.][schema.]object|query})

Передает указанный запрос источнику данных, не требуя, чтобы он был сконфигурирован как связанный сервер.

PARSENAME(object_name, object_piece)

Возвращает указанную информацию об объекте: имя объекта (1), имя владельца (2), имя базы данных (3) и имя сервера (4). Параметр *object_piece* должен быть целым числом от 1 до 4. Например:

```
SELECT PARSENAME('pubs..authors', 1) -> 'authors'
SELECT PARSENAME('pubs..authors', 2) -> NULL
SELECT PARSENAME('pubs..authors', 3) -> 'pubs'
SELECT PARSENAME('pubs..authors', 4) -> NULL
```

PATINDEX('%pattern%', expression)

Возвращает начальную позицию первого вхождения образца в строку. Например:

```
SELECT PATINDEX('%Du%', 'Donald Duck') -> 8
```

PERMISSIONS([object_id[, column]])

Возвращает битовую карту, указывающую разрешения на доступ к указанному объекту или столбцу для текущего пользователя. Например:

```
SELECT PERMISSIONS(OBJECT_ID('authors'))&8 -> 8
```

PI( )

Возвращает значение числа π. Например:

```
SELECT 2*PI( ) -> 6.2831853071795862
```

RADIANS(numeric_expression)

Переводит градусы в радианы. Например:

```
SELECT RADIANS( 90.0 ) -> 1.570796326794896600
```

RAND([seed])

Возвращает псевдослучайное число типа *FLOAT* между 0 и 1, seed – это число, задающее начальное значение. Например:

```
SELECT RAND(PI( )) -> 0.71362925915543995
```

REPLACE(string_expression1, string_expression2, string_expression3)

Заменяет все вхождения строки *string_expression2* в строке *string_expression1* строкой *string_expression3*. Например:

```
SELECT REPLACE('Donald Duck', 'Duck', 'Trump') -> 'Donald Trump'
```

REPLICATE(character_expression, integer_expression)

Повторяет строку указанное число раз. Например:

```
SELECT REPLICATE( 'FOOBAR', 3 ) -> 'FOOBARFOOBARFOOBAR'
```

REVERSE(character_expression)

Возвращает обращенную строку. Например:

```
SELECT REVERSE( 'Donald Duck' ) -> 'kcuD dlanoD'
```

RIGHT(character_expression, integer_expression)

Возвращает крайние правые *integer_expression* символов символьного выражения *character_expression*. Например:

```
SELECT RIGHT( 'Donald Duck', 4 ) -> 'Duck'
```

ROUND(number, decimal[, function])

Округляет число *number* до *decimal* знаков после запятой. Отметим, что целое число *decimal* может быть отрицательным, тогда число округляется слева от запятой. Если значение *function* отлично от нуля, то возвращаемое значение усекается, в противном случае округляется. Например:

```
SELECT ROUND(PI( ), 2) -> 3.1400000000000001
```

ROWCOUNT_BIG( )

Возвращает число строк, затронутых при выполнении последней команды. (То же, что *@@ROWCOUNT*, но возвращает значение типа *BIGINT*.) Например:

```
SELECT ROWCOUNT_BIG( ) -> 1
```

RTRIM(character_expression)

Удаляет конечные пробелы. Например:

```
SELECT RTRIM(' beaucoup ') -> ' beaucoup'
```

SIGN(numeric_expression)

Если аргумент отрицателен, возвращает –1; если равен 0, то 0; а если положителен, то 1. Например:

```
SELECT SIGN(-PI( )) -> -1.0
```

SIN(number)

Возвращает синус числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT SIN( 0 ) -> 0.000000
```

SOUNDEX(character_expression)

Возвращает четырехсимвольный код для оценки степени фонетического сходства двух строк. Например:

```
SELECT SOUNDEX('char') -> 'C600'
```

SPACE(integer_expression)

Возвращает строку, состоящую из заданного количества пробелов. Например:

```
SELECT SPACE(5) -> '  '
```

STATS_DATE(table_id, index_id)

Возвращает дату и время последнего обновления статистики индекса. Например:

```
SELECT i.name, STATS_DATE(i.id, i.indid)FROM sysobjects o, sysindexes i
WHERE o.name = 'authors' AND o.id = i.id
UPKCL_auidind 2000-08-06 01:34:00.153
aunmind 2000-08-06 01:34:00.170
```

STDEV(expression)

Возвращает стандартное отклонение всех значений в указанном выражении. Например:

```
SELECT STDEV( qty ) FROM sales -> 16.409201831957116
```

STDEVP(expression)

Возвращает стандартное отклонение генеральной совокупности всех значений в указанном выражении. Например:

```
SELECT STDEVP( qty ) FROM sales -> 16.013741264834152
```

STR(number[,length[,decimal]])

Преобразует число *number* в символьную строку длины *length* с *decimal* знаками после запятой.

STUFF(string1,start,length,string2)

Удаляет *length* символов в строке *string1*, начиная с позиции *start*, и вставляет вместо них строку *string2*. Например:

```
SELECT STUFF( 'Donald Duck', 8, 4, 'Trump' ) -> 'Donald Trump'
```

SUBSTRING(string,start,length)

Возвращает *length* символов строки *string1*, начиная с позиции *start*. Например:

```
SELECT SUBSTRING( 'Donald Duck', 8, 4 ) -> 'Duck'
```

SUSER_ID([login])

Возвращает системный идентификатор пользователя (ID) с заданным именем. Начиная с версии SQL Server 2000, эта функция всегда возвращает NULL, поэтому старайтесь ею не пользоваться.

SUSER_SID([login])

Возвращает системный идентификатор безопасности (SID) текущего пользователя или пользователя с заданным именем. SID возвращается в двоичном формате. Например:

```
SELECT SUSER_SID('montoyai')
0x68FC17A71010DE40B005BCF2E443B377
```

SUSER_SNAME([server_user_sid])

Возвращает имя текущего пользователя или пользователя с заданным идентификатором безопасности SID. Например:

```
SELECT SUSER_SNAME( ) -> 'montoyai'
```

TAN(number)

Возвращает тангенс числа *number*, представляющего собой угол, выраженный в радианах. Например:

```
SELECT TAN( 3.1415 ) -> -0.000093
```

TEXTPTR(column)

Возвращает указатель на столбец типа *TEXT*, *NTEXT* или *IMAGE* в формате *VARBINARY*. Например:

```
SELECT TEXTPTR(pr_info)FROM pub_info WHERE pub_id ='0736'
0xFEFF6F00000000005C00000001000100
```

TEXTVALID(table.column, text_ptr)

Возвращает true или false (1 или 0) в зависимости от того, действителен ли указатель на столбец типа *TEXT*, *NTEXT* или *IMAGE*. Например:

```
SELECT pub_id, 'Valid (if 1) Text data'
= TEXTVALID('pub_info.logo', TEXTPTR(logo)) FROM pub_info ORDER BY pub_id
0736       1
0877       1
1389       1
1622       1
1756       1
9901       1
9952       1
9999       1
```

TYPEPROPERTY(datatype, property)

Возвращает информацию о свойствах указанного типа данных. Параметр *datatype* может содержать имя произвольного типа данных, а параметр *property* может принимать следующие значения:

Precision

Точность *datatype*, то есть количество цифр или символов, которое этот тип способен хранить.

Scale

Масштаб *datatype*, то есть количество десятичных знаков для числового типа данных. Если *datatype* не числовой тип, возвращается NULL.

Например:

```
SELECT TYPEPROPERTY('decimal', 'PRECISION') -> 38
```

UNICODE(ncharacter_expression)

Возвращает кодовую позицию *UNICODE* для первого символа указанного выражения. Например:

```
SELECT UNICODE('Hello!') -> 72
```

USER_ID([user])

Возвращает идентификатор (ID) пользователя *user* в текущей базе данных. Если параметр *user* опущен, то возвращает идентификатор текущего пользователя. Например:

```
SELECT USER_ID( ) -> 2
```

USER_NAME([id])

Возвращает имя пользователя с идентификатором *id*. Если параметр *id* опущен, то возвращает имя текущего пользователя. Например:

```
SELECT USER_NAME( ) -> 'montoyai'
```

VAR(expression)

Возвращает дисперсию всех значений в указанном выражении. Например:

```
SELECT VAR(qty) FROM sales -> 269.26190476190476
```

VARP(expression)

Возвращает дисперсию генеральной совокупности всех значений в указанном выражении. *VARP* – агрегатная функция. Например:

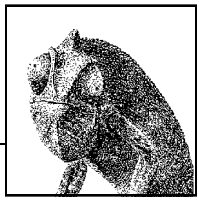
```
SELECT VARP(qty) FROM sales -> 256.43990929705217
```

YEAR(date)

Возвращает целое число, представляющее год указанной даты *date*. Например:

```
SELECT YEAR( CURRENT_TIMESTAMP ) -> 2003
```

A



Ключевые слова: общие и платформо-зависимые

В следующих таблицах приведены ключевые слова, определенные в стандарте ANSI, а также в четырех рассматриваемых в настоящей книге реализациях SQL. Таблицы расположены в следующем порядке:

1. Общие ключевые слова
2. Ключевые слова в стандарте SQL2003
3. Ключевые слова MySQL
4. Ключевые слова Oracle
5. Ключевые слова PostgreSQL
6. Ключевые слова SQL Server

Таблица A.1. Общие ключевые слова

ADD	*ALL*	*ALTER*	*AND*
AS	*ASC*	*BY*	*CHECK*
COLUMN	*CREATE*	*DATE*	*DEFAULT*
DELETE	*DESC*	*DROP*	*FOR*
FROM	*IN*	*INTO*	*IS*
LIKE	*NOT*	*NULL*	*ON*
OR	*ORDER*	*REVOKE*	*SELECT*
SET	*TABLE*	*THEN*	*TO*
UNIQUE	*UPDATE*	*WITH*	

Таблица A.2. Ключевые слова в стандарте SQL2003

ABSOLUTE	*ACTION*	*ADD*	*ADMIN*
AFTER	*AGGREGATE*	*ALIAS*	*ALL*
ALLOCATE	*ALTER*	*AND*	*ANY*

ARE	*ARRAY*	*AS*	*ASC*
ASSERTION	*AT*	*ATOMIC*	*AUTHORIZATION*
BEFORE	*BEGIN*	*BIGINT*	*BINARY*
BIT	*BLOB*	*BOOLEAN*	*BOTH*
BREADTH	*BY*	*CALL*	*CASCADE*
CASCADED	*CASE*	*CAST*	*CATALOG*
CHAR	*CHARACTER*	*CHECK*	*CLASS*
CLOB	*CLOSE*	*COLLATE*	*COLLATION*
COLLECT	*COLUMN*	*COMMIT*	*COMPLETION*
CONDITION	*CONNECT*	*CONNECTION*	*CONSTRAINT*
CONSTRAINTS	*CONSTRUCTOR*	*CONTAINS*	*CONTINUE*
CORRESPONDING	*CREATE*	*CROSS*	*CUBE*
CURRENT	*CURRENT_DATE*	*CURRENT_PATH*	*CURRENT_ROLE*
CURRENT_TIME	*CURRENT_ TIMESTAMP*	*CURRENT_USER*	*CURSOR*
CYCLE	*DATA*	*DATALINK*	*DATE*
DAY	*DEALLOCATE*	*DEC*	*DECIMAL*
DECLARE	*DEFAULT*	*DEFERRABLE*	*DELETE*
DEPTH	*DEREF*	*DESC*	*DESCRIPTOR*
DESTRUCTOR	*DIAGNOSTICS*	*DICTIONARY*	*DISCONNECT*
DO	*DOMAIN*	*DOUBLE*	*DROP*
ELEMENT	*END-EXEC*	*EQUALS*	*ESCAPE*
EXCEPT	*EXCEPTION*	*EXECUTE*	*EXIT*
EXPAND	*EXPANDING*	*FALSE*	*FIRST*
FLOAT	*FOR*	*FOREIGN*	*FREE*
FROM	*FUNCTION*	*FUSION*	*GENERAL*
GET	*GLOBAL*	*GOTO*	*GROUP*
GROUPING	*HANDLER*	*HASH*	*HOUR*
IDENTITY	*IF*	*IGNORE*	*IMMEDIATE*
IN	*INDICATOR*	*INITIALIZE*	*INITIALLY*
INNER	*INOUT*	*INPUT*	*INSERT*
INT	*INTEGER*	*INTERSECT*	*INTERSECTION*
INTERVAL	*INTO*	*IS*	*ISOLATION*
ITERATE	*JOIN*	*KEY*	*LANGUAGE*
LARGE	*LAST*	*LATERAL*	*LEADING*
LEAVE	*LEFT*	*LESS*	*LEVEL*

Таблица А.2 (продолжение)

LIKE	*LIMIT*	*LOCAL*	*LOCALTIME*
LOCALTIMESTAMP	*LOCATOR*	*LOOP*	*MATCH*
MEMBER	*MEETS*	*MERGE*	*MINUTE*
MODIFIES	*MODIFY*	*MODULE*	*MONTH*
MULTISET	*NAMES*	*NATIONAL*	*NATURAL*
NCHAR	*NCLOB*	*NEW*	*NEXT*
NO	*NONE*	*NORMALIZE*	*NOT*
NULL	*NUMERIC*	*OBJECT*	*OF*
OFF	*OLD*	*ON*	*ONLY*
OPEN	*OPERATION*	*OPTION*	*OR*
ORDER	*ORDINALITY*	*OUT*	*OUTER*
OUTPUT	*PAD*	*PARAMETER*	*PARAMETERS*
PARTIAL	*PATH*	*PERIOD*	*POSTFIX*
PRECEDES	*PRECISION*	*PREFIX*	*PREORDER*
PREPARE	*PRESERVE*	*PRIMARY*	*PRIOR*
PRIVILEGES	*PROCEDURE*	*PUBLIC*	*READ*
READS	*REAL*	*RECURSIVE*	*REDO*
REF	*REFERENCES*	*REFERENCING*	*RELATIVE*
REPEAT	*RESIGNAL*	*RESTRICT*	*RESULT*
RETURN	*RETURNS*	*REVOKE*	*RIGHT*
ROLE	*ROLLBACK*	*ROLLUP*	*ROUTINE*
ROW	*ROWS*	*SAVEPOINT*	*SCHEMA*
SCROLL	*SEARCH*	*SECOND*	*SECTION*
SELECT	*SEQUENCE*	*SESSION*	*SESSION_USER*
SET	*SETS*	*SIGNAL*	*SIZE*
SMALLINT	*SPECIFIC*	*SPECIFICTYPE*	*SQL*
SQLEXCEPTION	*SQLSTATE*	*SQLWARNING*	*START*
STATE	*STATIC*	*STRUCTURE*	*SUBMULTISET*
SUCCEEDS	*SUM*	*SYSTEM_USER*	*TABLE*
TABLESAMPLE	*TEMPORARY*	*TERMINATE*	*THAN*
THEN	*TIME*	*TIMESTAMP*	*TIMEZONE_HOUR*
TIMEZONE_MINUTE	*TO*	*TRAILING*	*TRANSACTION*
TRANSLATION	*TREAT*	*TRIGGER*	*TRUE*
UESCAPE	*UNDER*	*UNDO*	*UNION*

UNIQUE	*UNKNOWN*	*UNTIL*	*UPDATE*
USAGE	*USER*	*USING*	*VALUE*
VALUES	*VARCHAR*	*VARIABLE*	*VARYING*
VIEW	*WHEN*	*WHENEVER*	*WHERE*
WHILE	*WITH*	*WRITE*	*YEAR*

Таблица А.3. Ключевые слова в MySQL

ACCESSIBLE	*ADD*	*ALL*	*ALTER*
ANALYZE	*AND*	*AS*	*ASC*
ASENSITIVE	*BEFORE*	*BETWEEN*	*BIGINT*
BINARY	*BLOB*	*BOTH*	*BY*
CALL	*CASCADE*	*CASE*	*CHANGE*
CHAR	*CHARACTER*	*CHECK*	*COLLATE*
COLUMN	*CONDITION*	*CONSTRAINT*	*CONTINUE*
CONVERT	*CREATE*	*CROSS*	*CURRENT_DATE*
CURRENT_TIME	*CURRENT_ TIMESTAMP*	*CURRENT_USER*	*CURSOR*
DATABASE	*DATABASES*	*DAY_HOUR*	*DAY_ MICROSECOND*
DAY_MINUTE	*DAY_SECOND*	*DEC*	*DECIMAL*
DECLARE	*DEFAULT*	*DELAYED*	*DELETE*
DESC	*DESCRIBE*	*DETERMINISTIC*	*DISTINCT*
DISTINCTROW	*DIV*	*DOUBLE*	*DROP*
DUAL	*EACH*	*ELSE*	*ELSEIF*
ENCLOSED	*ESCAPED*	*EXISTS*	*EXIT*
EXPLAIN	*FALSE*	*FETCH*	*FLOAT*
FLOAT4	*FLOAT8*	*FOR*	*FORCE*
FOREIGN	*FROM*	*FULLTEXT*	*GRANT*
GROUP	*HAVING*	*HIGH_PRIORITY*	*HOUR_ MICROSECOND*
HOUR_MINUTE	*HOUR_SECOND*	*IF*	*IGNORE*
IN	*INDEX*	*INFILE*	*INNER*
INOUT	*INSENSITIVE*	*INSERT*	*INT*
INT1	*INT2*	*INT3*	*INT4*
INT8	*INTEGER*	*INTERVAL*	*INTO*
IS	*ITERATE*	*JOIN*	*KEY*

Таблица А.3 (продолжение)

KEYS	*KILL*	*LEADING*	*LEAVE*
LEFT	*LIKE*	*LIMIT*	*LINEAR*
LINES	*LOAD*	*LOCALTIME*	*LOCALTIME-STAMP*
LOCK	*LONG*	*LONGBLOB*	*LONGTEXT*
LOOP	*LOW_PRIORITY*	*MASTER_SSL_VERI-FY_SERVER_CERT*	*MATCH*
MEDIUMBLOB	*MEDIUMINT*	*MEDIUMTEXT*	*MIDDLEINT*
MINUTE_ MICROSECOND	*MINUTE_SECOND*	*MOD*	*MODIFIES*
NATURAL	*NO_WRITE_TO_ BINLOG*	*NOT*	*NULL*
NUMERIC	*ON*	*OPTIMIZE*	*OPTION*
OPTIONALLY	*OR*	*ORDER*	*OUT*
OUTER	*OUTFILE*	*PRECISION*	*PRIMARY*
PROCEDURE	*PURGE*	*RANGE*	*READ*
READ_ONLY	*READ_WRITE*	*READS*	*REAL*
REFERENCES	*REGEXP*	*RELEASE*	*RENAME*
REPEAT	*REPLACE*	*REQUIRE*	*RESTRICT*
RETURN	*REVOKE*	*RIGHT*	*RLIKE*
SCHEMA	*SCHEMAS*	*SECOND_MICRO-SECOND*	*SELECT*
SENSITIVE	*SEPARATOR*	*SET*	*SHOW*
SMALLINT	*SPATIAL*	*SPECIFIC*	*SQL*
SQL_BIG_RESULT	*SQL_CALC_ FOUND_ROWS*	*SQL_SMALL_ RESULT*	*SQLEXCEPTION*
SQLSTATE	*SQLWARNING*	*SSL*	*STARTING*
STRAIGHT_JOIN	*TABLE*	*TERMINATED*	*THEN*
TINYBLOB	*TINYINT*	*TINYTEXT*	*TO*
TRAILING	*TRIGGER*	*TRUE*	*UNDO*
UNION	*UNIQUE*	*UNLOCK*	*UNSIGNED*
UPDATE	*USAGE*	*USE*	*USING*
UTC_DATE	*UTC_TIME*	*UTC_TIMESTAMP*	*VALUES*
VARBINARY	*VARCHAR*	*VARCHARACTER*	*VARYING*
WHEN	*WHERE*	*WHILE*	*WITH*
WRITE	*XOR*	*YEAR_MONTH*	*ZEROFILL*

Таблица А.4. Ключевые слова в Oracle

ACCESS	*ADD*	*ALL*	*ALTER*
AND	*ANY*	*AS*	*ASC*
AUDIT	*BETWEEN*	*BY*	*CHAR*
CHECK	*CLUSTER*	*COLUMN*	*COMMENT*
COMPRESS	*CONNECT*	*CREATE*	*CURRENT*
DATE	*DECIMAL*	*DEFAULT*	*DELETE*
DESC	*DISTINCT*	*DROP*	*ELSE*
EXCLUSIVE	*EXISTS*	*FILE*	*FLOAT*
FOR	*FROM*	*GRANT*	*GROUP*
HAVING	*IDENTIFIED*	*IMMEDIATE*	*IN*
INCREMENT	*INDEX*	*INITIAL*	*INSERT*
INTEGER	*INTERSECT*	*INTO*	*IS*
LEVEL	*LIKE*	*LOCK*	*LONG*
MAXEXTENTS	*MINUS*	*MLSLABEL*	*MODE*
MODIFY	*NOAUDIT*	*NOCOMPRESS*	*NOT*
NOWAIT	*NULL*	*NUMBER*	*OF*
OFFLINE	*ON*	*ONLINE*	*OPTION*
OR	*ORDER*	*PCTFREE*	*PRIOR*
PRIVILEGES	*PUBLIC*	*RAW*	*RENAME*
RESOURCE	*REVOKE*	*ROW*	*ROWID*
ROWNUM	*ROWS*	*SELECT*	*SESSION*
SET	*SHARE*	*SIZE*	*SMALLINT*
START	*SUCCESSFUL*	*SYNONYM*	*SYSDATE*
TABLE	*THEN*	*TO*	*TRIGGER*
UID	*UNION*	*UNIQUE*	*UPDATE*
USER	*VALIDATE*	*VALUES*	*VARCHAR*

Таблица А.5. Ключевые слова в PostgreSQL

ABORT	*ADD*	*ALL*	*ALLOCATE*
ALTER	*ANALYZE*	*AND*	*ANY*
ARE	*AS*	*ASC*	*ASSERTION*
AT	*AUTHORIZATION*	*AVG*	*BEGIN*
BETWEEN	*BINARY*	*BIT*	*BIT_LENGTH*
BOTH	*BY*	*CASCADE*	*CASCADED*

Таблица А.5 (продолжение)

CASE	*CAST*	*CATALOG*	*CHAR*
CHAR_LENGTH	*CHARACTER*	*CHARACTER_LENGTH*	*CHECK*
CLOSE	*CLUSTER*	*COALESCE*	*COLLATE*
COLLATION	*COLUMN*	*COMMIT*	*CONNECT*
CONNECTION	*CONSTRAINT*	*CONTINUE*	*CONVERT*
COPY	*CORRESPONDING*	*COUNT*	*CREATE*
CROSS	*CURRENT*	*CURRENT_DATE*	*CURRENT_ SESSION*
CURRENT_TIME	*CURRENT_ TIMESTAMP*	*CURRENT_USER*	*CURSOR*
DATE	*DEALLOCATE*	*DEC*	*DECIMAL*
DECLARE	*DEFAULT*	*DELETE*	*DESC*
DESCRIBE	*DESCRIPTOR*	*DIAGNOSTICS*	*DISCONNECT*
DISTINCT	*DO*	*DOMAIN*	*DROP*
ELSE	*END*	*ESCAPE*	*EXCEPT*
EXCEPTION	*EXEC*	*EXECUTE*	*EXISTS*
EXPLAIN	*EXTEND*	*EXTERNAL*	*EXTRACT*
FALSE	*FETCH*	*FIRST*	*FLOAT*
FOR	*FOREIGN*	*FOUND*	*FROM*
FULL	*GET*	*GLOBAL*	*GO*
GOTO	*GRANT*	*GROUP*	*HAVING*
IDENTITY	*IN*	*INDICATOR*	*INNER*
INPUT	*INSERT*	*INTERSECT*	*INTERVAL*
INTO	*IS*	*JOIN*	*LAST*
LEADING	*LEFT*	*LIKE*	*LISTEN*
LOAD	*LOCAL*	*LOCK*	*LOWER*
MAX	*MIN*	*MODULE*	*MOVE*
NAMES	*NATIONAL*	*NATURAL*	*NCHAR*
NEW	*NO*	*NONE*	*NOT*
NOTIFY	*NULL*	*NULLIF*	*NUMERIC*
OCTET_LENGTH	*OFFSET*	*ON*	*OPEN*
OR	*ORDER*	*OUTER*	*OUTPUT*
OVERLAPS	*PARTIAL*	*POSITION*	*PRECISION*
PREPARE	*PRESERVE*	*PRIMARY*	*PRIVILEGES*
PROCEDURE	*PUBLIC*	*REFERENCES*	*RESET*

REVOKE	*RIGHT*	*ROLLBACK*	*ROWS*
SCHEMA	*SECTION*	*SELECT*	*SESSION*
SESSION_USER	*SET*	*SETOF*	*SHOW*
SIZE	*SOME*	*SQL*	*SQLCODE*
SQLERROR	*SQLSTATE*	*SUBSTRING*	*SUM*
SYSTEM_USER	*TABLE*	*TEMPORARY*	*THEN*
TO	*TRAILING*	*TRANSACTION*	*TRANSLATE*
TRANSLATION	*TRIM*	*TRUE*	*UNION*
UNIQUE	*UNKNOWN*	*UNLISTEN*	*UNTIL*
UPDATE	*UPPER*	*USAGE*	*USER*
USING	*VACUUM*	*VALUE*	*VALUES*
VARCHAR	*VARYING*	*VERBOSE*	*VIEW*
WHEN	*WHENEVER*	*WHERE*	*WITH*
WORK	*WRITE*		

Таблица А.6. Ключевые слова в SQL Server

ADD	*ALL*	*ALTER*	*AND*
ANY	*AS*	*ASC*	*AUTHORIZATION*
BACKUP	*BEGIN*	*BETWEEN*	*BREAK*
BROWSE	*BULK*	*BY*	*CASCADE*
CASE	*CHECK*	*CHECKPOINT*	*CLOSE*
CLUSTERED	*COALESCE*	*COLLATE*	*COLUMN*
COMMIT	*COMPUTE*	*CONSTRAINT*	*CONTAINS*
CONTAINSTABLE	*CONTINUE*	*CONVERT*	*CREATE*
CROSS	*CURRENT*	*CURRENT_DATE*	*CURRENT_TIME*
CURRENT_ TIMESTAMP	*CURRENT_USER*	*CURSOR*	*DATABASE*
DBCC	*DEALLOCATE*	*DECLARE*	*DEFAULT*
DELETE	*DENY*	*DESC*	*DISK*
DISTINCT	*DISTRIBUTED*	*DOUBLE*	*DROP*
DUMP	*ELSE*	*END*	*ERRLVL*
ESCAPE	*EXCEPT*	*EXEC*	*EXECUTE*
EXISTS	*EXIT*	*EXTERNAL*	*FETCH*
FILE	*FILLFACTOR*	*FOR*	*FOREIGN*
FREETEXT	*FREETEXTTABLE*	*FROM*	*FULL*

Таблица А.6 (продолжение)

FUNCTION	*GOTO*	*GRANT*	*GROUP*
HAVING	*HOLDLOCK*	*IDENTITY*	*IDENTITY_INSERT*
IDENTITYCOL	*IF*	*IN*	*INDEX*
INNER	*INSERT*	*INTERSECT*	*INTO*
IS	*JOIN*	*KEY*	*KILL*
LEFT	*LIKE*	*LINENO*	*LOAD*
NATIONAL	*NOCHECK*	*NONCLUSTERED*	*NOT*
NULL	*NULLIF*	*OF*	*OFF*
OFFSETS	*ON*	*OPEN*	*OPENDATASOURCE*
OPENQUERY	*OPENROWSET*	*OPENXML*	*OPTION*
OR	*ORDER*	*OUTER*	*OVER*
PERCENT	*PIVOT*	*PLAN*	*PRECISION*
PRIMARY	*PRINT*	*PROC*	*PROCEDURE*
PUBLIC	*RAISERROR*	*READ*	*READTEXT*
RECONFIGURE	*REFERENCES*	*REPLICATION*	*RESTORE*
RESTRICT	*RETURN*	*REVERT*	*REVOKE*
RIGHT	*ROLLBACK*	*ROWCOUNT*	*ROWGUIDCOL*
RULE	*SAVE*	*SCHEMA*	*SECURITYAUDIT*
SELECT	*SESSION_USER*	*SET*	*SETUSER*
SHUTDOWN	*SOME*	*STATISTICS*	*SYSTEM_USER*
TABLE	*TABLESAMPLE*	*TEXTSIZE*	*THEN*
TO	*TOP*	*TRAN*	*TRANSACTION*
TRIGGER	*TRUNCATE*	*TSEQUAL*	*UNION*
UNIQUE	*UNPIVOT*	*UPDATE*	*UPDATETEXT*
USE	*USER*	*VALUES*	*VARYING*
VIEW	*WAITFOR*	*WHEN*	*WHERE*
WHILE	*WITH*	*WRITETEXT*	

Алфавитный указатель

Специальные символы

!< (оператор «не меньше»), 40
!= (оператор «не равно»), 40, 42
!> (оператор «не больше»), 40
"…" (двойные кавычки), разделитель идентификаторов, 33, 43
% (арифметический оператор взятия остатка при делении), 39
% (метасимвол, соответствующий произвольной строке), 43
& (битовый оператор И), 39
'…' (одинарные кавычки)
ограничитель символьных литералов, 37, 43
разделитель идентификаторов, 35
(…) (скобки), оператор приоритета, 41, 43
* (арифметический оператор умножения), 39, 42
+ (арифметический оператор сложения), 39, 42
+ (оператор конкатенации, SQL Server), 42
+ (унарный оператор положительного значения), 41
, (разделитель списка элементов), 43
- (арифметический оператор вычитания), 39, 42
- (оператор диапазон в ограничениях CHECK), 42
- (унарный оператор отрицательного значения), 41
-- (однострочный комментарий), 43
. (разделитель в идентификаторе), 43
/ (арифметический оператор деления), 39
/*…*/ (многострочный комментарий), 43
< (оператор «меньше»), 40, 42
<= (оператор «меньше или равно»), 40, 42
<> (оператор «не равно»), 40, 42
= (оператор присваивания), 39
:= (оператор присваивания, Oracle), 39
= (оператор сравнения на равенство), 40, 42
[…] (квадратные скобки), разделитель идентификатора, 35
[] (квадратные скобки), 33
^ (битовый оператор исключающего ИЛИ), 39
_ (подчеркивание), в идентификаторах, 32
| (битовый оператор ИЛИ), 39
|| (оператор конкатенации), 545
~ (битовый оператор отрицания), 41
> (оператор «больше»), 40, 42
>= (оператор «больше или равно»), 40, 42

A

ABS, функция, 533
ABSTIME, функция, PostgreSQL, 597
ACOS, функция
MySQL, 551
PostgreSQL, 597
ACOS, функция,
Oracle, 570
SQL Server, 610
ADD_MONTHS, функция, Oracle, 570
ADDDATE, функция, MySQL, 551
AES_DECRYPT, функция, Oracle, 551
AES_ENCRYPT, функция, Oracle, 551
AGE, функция, PostgreSQL, 597
ALL, оператор, 40, 79
ALTER DATABASE, оператор, 92

MySQL, 93
Oracle, 94
PostgreSQL, 112
SQL Server, 113
ALTER FUNCTION, оператор, 123
MySQL, 129
Oracle, 130
PostgreSQL, 134
SQL Server, 136
ALTER INDEX, оператор, 141
Oracle, 145
PostgreSQL, 154
SQL Server, 156
ALTER METHOD, оператор, 159
ALTER PRODCEDURE, оператор, 123
MySQL, 129
Oracle, 130
PostgreSQL, 134
SQL Server, 136
ALTER ROLE, оператор
Oracle, 163
PostgreSQL, 164
SQL Server, 166
ALTER SCHEMA, оператор
PostgreSQL, 169
SQL Server, 170
ALTER TABLE, оператор, 170
MySQL, 177
Oracle, 188
PostgreSQL, 220
SQL Server, 223
ALTER TRIGGER, оператор, 227
Oarcle, 232
PostgreSQL, 236
SQL Server, 237
ALTER VIEW, оператор, 255
MySQL, 259
Oracle, 260
SQL Server, 263
AND, оператор, 40
Any Type, типы данных, Oracle, 55
ANY, оператор, 40, 79
APP_NAME, функция, SQL Server, 610
APPENDCHILDXML, функция, Oracle, 570
AREA, функция, PostgreSQL, 597
ARRAY_APPEND, функция, PostgreSQL, 597
ARRAY_CAT, функция, PostgreSQL, 597
ARRAY_DIMS, функция, PostgreSQL, 597
ARRAY_LOWER, функция, PostgreSQL, 597
ARRAY_PREPEND, функция, PostgreSQL, 597
ARRAY_TO_STRING, функция, PostgreSQL, 598
ARRAY_UPPER, функция, PostgreSQL, 598
ASCII, функция
MySQL, 551
PostgreSQL, 598
SQL Server, 610
ASCII, функция, Oracle, 571
ASIN, функция
MySQL, 551
PostrgreSQL, 598
SQL Server, 610
ASIN, функция, Oracle, 571
ATAN, функция
MySQL, 551
PostgreSQL, 598
SQL Server, 610
ATAN, функция, Oracle, 571
ATAN2, функция
MySQL, 551
ATAN2, функция, Oracle, 571
ATN2, функция, SQL Server, 611
AuthorizationID, 19
AVG,, функция, 504

B

BENCHMARK, функция, MySQL, 552
BETWEEN, оператор, 40, 81
BFILE, тип данных, Oracle, 55
BFILENAME, функция, Oracle, 571
BIGINT, тип данных, 47
MySQL, 51
PostgreSQL, 58
SQL Server, 63
BIGSERIAL, тип данных, PostgreSQL, 59, 61
BIN, функция, MySQL, 552
BIN_TO_NUM, функция, Oracle, 571
BINARY, тип данных, 47
MySQL, 51
SQL Server, 63
BINARY, функция, MySQL, 552

BINARY_CHECKSUM, функция, SQL Server, 611
BINARY_DOUBLE, тип данных, Oracle, 55
BINARY_FLOAT, тип данных, Oracle, 55
Bindings:1999, раздел SQL99, 24
BIT VARYING, тип данных
PostgreSQL, 59
удален в SQL99, 25
BIT, тип данных, 47
MySQL, 51
PostgreSQL, 59
SQL Server, 63
удален из SQL99, 25
BIT_AND, функция
MySQL, 552
PostgreSQL, 598
BIT_COUNT, функция, MySQL, 552
BIT_LENGTH,, функция, 534
BIT_OR, функция
MySQL, 552
PostgreSQL, 598
BIT_XOR, функция, MySQL, 552
BITAND, функция, Oracle, 571
BLOB, тип данных, 47
MySQL, 51
Oracle, 55
BOOL, тип данных, 47
MySQL, 51
PostgreSQL, 59
BOOL_AND, функция, PostgreSQL, 598
BOOL_OR, функция, PostgreSQL, 598
BOOLEAN тип данных, 47, 51, 59
BOX, тип данных, PostgreSQL, 59
BOX, функция, PostgreSQL, 598
BROADCAST, функция, PostgreSQL, 599
BTRIM, функция, PostgreSQL, 599
BYTEA, тип данных, PostgreSQL, 59

C

Call Level Interface (CLI), 26
CALL, оператор, 83
CARDINALITY, функция, Oracle, 571
CASE, функция, 530
CAST, функция, 532
CBRT, функция, PostgreSQL, 599
CEIL, функция, 535
CEILING, функция, 535
CENTER, функция, PostgreSQL, 599
CHAR FOR BIT DATA, тип данных, 47
CHAR VARYING, тип данных, 50
Oracle, 58
SQL Server, 66
CHAR, тип данных, 47
MySQL, 52
Oracle, 56
PostgreSQL, 59
SQL Server, 63
CHAR, функция
MySQL, 552
PostgreSQL, 599
SQL Server, 611
CHAR_LENGTH, функция, 599
CHAR_LENGTH, функция, 534
CHARACTER VARYING, тип данных, 50
Oracle, 58
PostgreSQL, 62
SQL Server, 66
CHARACTER, тип данных, 47
MySQL, 52
Oracle, 56
PostgreSQL, 59
SQL Server, 63
CHARACTER_LENGTH, функция, PostgreSQL, 599
CHARINDEX, функция, SQL Server, 611
CHARSET, функция, MySQL, 552
CHARTOROWID, функция, Oracle, 572
CHECK, ограничения целостности, 73
CHECKSUM, функция, SQL Server, 611
CHECKSUM_AGG, функция, SQL Server, 611
CHR, функция, Oracle, 572
CIDR, тип данных, PostgreSQL, 59
CIRCLE, тип данных, PostgreSQL, 59
CIRCLE, функция, PostgreSQL, 599
CLI(Call Level Interface), 26
CLOB, тип данных, Oracle, 56
CLOCK_TIMESTAMP, функция, PostgreSQL, 600
CLOSE CURSOR, оператор, 85
CLUSTER_ID, функция, Oracle, 572
CLUSTER_PROBABILITY, функция, Oracle, 572
CLUSTER_SET, функция, Oracle, 572
COALESCE, функция
MySQL, 552
PostgreSQL, 600
SQL Server, 611

COALESCE, функция, Oracle, 572
COERCIBLITY, функция, MySQL, 553
COL_LENGTH, функция, SQL Server, 611
COL_NAME, функция, SQL Server, 611
COLLATION, функция, MySQL, 553
COLLECT, функция, Oracle, 572
COMMIT, оператор, 86
COMPOSE, функция, Oracle, 572
COMPRESS, функция, MySQL, 553
CONCAT, функция, Oracle, 572
CONCAT_WS, функция, MySQL, 553
CONNECT, оператор, 90
CONNECTION_ID, функция, MySQL, 553
CONTAINS, функция, SQL Server, 612
CONTAINSTABLE, функция, SQL Server, 612
CONV, функция, MySQL, 553
CONVERT, функция, 546
  Oracle, 572
  SQL Server, 612
CORR, функция, 505
CORR_K, функция, Oracle, 572
CORR_S, функция, Oracle, 572
COS, функция
  MySQL, 553
  Oracle, 572
  PostgreSQL, 600
  SQL Server, 612
COSH, функция, Oracle, 573
COT, функция
  MySQL, 553
  PostgreSQL, 600
  SQL Server, 612
COUNT, функция, 506
COUNT_BIG, функция, SQL Server, 612
COVAR_POP, функция, 507
COVAR_SAMP, функция, 508
CRC32, функция, MySQL, 553
CREATE DATABASE, оператор, 92
  MySQL, 93
  Oracle, 94
  PostgreSQL, 112
  SQL Server, 113
CREATE FUNCTION, оператор, 123
  MySQL, 129
  Oracle, 130
  PostgreSQL, 134
  SQL Server, 136
CREATE INDEX оператор, 141
  MySQL, 144
  Oracle, 145
  PostgreSQL, 154
  SQL Server, 156
CREATE METHOD, оператор, 159
CREATE PROCEDURE, оператор, 123
  MySQL, 129
  Oracle, 130
  PostgreSQL, 134
  SQL Server, 136
CREATE ROLE, оператор, 162
  Oracle, 163
  PostgreSQL, 164
  SQL Server, 166
CREATE SCHEMA, оператор, 166
  Oracle, 168
  PostgreSQL, 169
  SQL Server, 169
CREATE TABLE, оператор, 170
  MySQL, 176
  Oracle, 187
  PostgreSQL, 219
  SQL Server, 222
CREATE TRIGGER, оператор, 227
  MySQL, 231
  Oarcle, 231
  PostgreSQL, 236
  SQL Server, 237
CREATE VIEW, оператор, 255
  MySQL, 259
  Oracle, 260
  PostgreSQL, 263
  SQL Server, 263
CUBE_TABLE, функция, Oracle, 573
CUME_DIST, функция, 508, 524
CURDATE, функция, MySQL, 554
CURRENT_DATE,, функция, 529
CURRENT_TIME,, функция, 529
CURRENT_TIMESTAMP, функция, 529
CURRENT_USER, функция, 529
CURRVAL, функция, PostgreSQL, 600
CURSOR, тип данных, SQL Server, 63
CURTIME, функция, MySQL, 554
CV, функция, Oracle, 573

D

Data Control Language (DCL), 28
Data Definition Language (DDL), 28
Data Manipulation Language (DML), 28
DATABASE, функция, MySQL, 554

DATABASEPROPERTYEX, функция, SQL Server, 613
DATALENGTH, функция, SQL Server, 613
DATALINK, тип данных, 47
DATAOBJ_TO_PARTITION, функция, Oracle, 573
DATE, тип данных, 47
  MySQL, 52
  Oracle, 56
  PostgreSQL, 59
  SQL Server, 63
DATE_ADD, функция, MySQL, 554
DATE_FORMAT, функция, MySQL, 554
DATE_PART, функция, PostgreSQL, 600
DATE_SUB, функция, MySQL, 554
DATE_TRUNC, функция, PostgreSQL, 600
DATEADD, функция, SQL Server, 613
DATEDIFF, функция, SQL Server, 613
DATENAME, функция, SQL Server, 613
DATEPART, функция, SQL Server, 613
DATETIME, тип данных, 47
  MySQL, 52
  SQL Server, 63
DATETIME2, тип данных, SQL Server, 63
DATETIMEOFFSET, тип данных, SQL Server, 63
DAY, функция, SQL Server, 613
DAYNAME, функция, MySQL, 555
DAYOFMONTH, функция, MySQL, 555
DAYOFWEEK, функция, MySQL, 556
DAYOFYEAR, функция, MySQL, 556
DB_ID, функция, SQL Server, 613
DB_NAME, функция, SQL Server, 613
DBCLOB, тип данных, 47
DBTIMEZONE, функция, Oracle, 573
DCL (Data Control Language), 28
DCL (Data Definition Language), 28
DEC, тип данных, 47
DECIMAL, тип данных, 47
  MySQL, 52
  Oracle, 56
  PostgreSQL, 60
  SQL Server, 64
DECLARE CURSOR, оператор, 85, 266
  MySQL, 270
  Oracle, 270
  PostgreSQL, 271
  SQL Server, 272
DECODE, функция
  MySQL, 556
  PostgreSQL, 600
DECODE, функция, Oracle, 573
DECOMPOSE, функция, Oracle, 573
DEFAULT, функция, MySQL, 556
DEGREES, функция
  MySQL, 556
  PostgreSQL, 600
  SQL Server, 614
DELETE, оператор, 276
  MySQL, 277
  Oracle, 279
  PostgreSQL, 281
  SQL Server, 282
DELETEXML, функция, Oracle, 573
DENSE_RANK, функция, 510, 525
DEPTH, функция, Oracle, 573
DEREF, функция, Oracle, 574
DES_DECRYPT, функция, MySQL, 556
DES_ENCRYPT, функция, MySQL, 556
DIAMETER, функция, PostgreSQL, 601
DIFFERENCE, функция, SQL Server, 614
DISCONNECT оператор, 285
  Oracle, 286
  SQL Server, 287
DML(Data Manipulation Language), 28
DOUBLE PRECISION тип данных, 47
  MySQL, 52
  Oracle, 56
  PostgreSQL, 60
  SQL Server, 64
DOUBLE, тип данных, 47, 52
DROP DATABASE оператор
  MySQL, 289
  Oracle, 291
  PostgreSQL, 293
  SQL Server, 295
DROP DOMAIN, оператор
  PostgreSQL, 293
DROP FUNCTION, оператор
  MySQL, 289
  Oracle, 291
  PostgreSQL, 294
  SQL Server, 295
DROP INDEX, оператор
  MySQL, 289
  Oracle, 291
  PostgreSQL, 294

SQL Server, 296
DROP PROCEDURE, оператор
MySQL, 290
Oracle, 291
SQL Server, 296
DROP ROLE, оператор
Oracle, 291
PostgreSQL, 294
SQL Server, 296
DROP SCHEMA, оператор
MySQL, 289
PostgreSQL, 294
SQL Server, 296
DROP TABLE, оператор
MySQL, 290
Oracle, 292
PostgreSQL, 294
SQL Server, 296
DROP TRIGGER, оператор
MySQL, 290
Oracle, 292
PostgreSQL, 294
SQL Server, 297
DROP TYPE, оператор
Oracle, 292
PostgreSQL, 294
SQL Server, 297
DROP VIEW, оператор
MySQL, 290
Oracle, 292
PostgreSQL, 295
SQL Server, 297
DROP, операторы, 287
DUMP, функция, Oracle, 574

E

ELT, функция, MySQL, 556
EMPTY_BLOB, функция, Oracle, 574
EMPTY_CLOB, функция, Oracle, 574
ENCODE, функция
MySQL, 556
PostgreSQL, 601
ENCRYPT, функция, MySQL, 557
ENUM, тип данных, 47, 52
EVERY, функция, PostgreSQL, 601
EXCEPT, оператор, 298
EXISTS, оператор, 80, 302
EXISTSNODE, функция, Oracle, 574
EXP, функция, 536
EXPORT_SET, функция, MySQL, 557
eXtensible Markup Language (XML)
в SQL2006, 24
в SQL3, 27
EXTRACT, функция, 536, 574
EXTRACTVALUE, функция
Oracle, 574
MySQL, 557

F

FEATURE_ID, функция, Oracle, 574
FEATURE_SET, функция, Oracle, 574
FEATURE_VALUE, функция, Oracle, 574
FETCH, оператор, 303
FIELD, функция, MySQL, 557
FILE_ID, функция, SQL Server, 614
FILE_NAME, функция, SQL Server, 614
FILEGROUP_ID, функция, SQL Server, 614
FILEGROUP_NAME, функция, SQL Server, 614
FILEGROUPPROPERTY, функция, SQL Server, 614
FILEPROPERTY, функция, SQL Server, 614
FIND_IN_SET, функция, MySQL, 557
FIRST, функция, Oracle, 574
FIRST_VALUE, функция, Oracle, 575
FLOAT, тип данных, 47
MySQL, 52
Oracle, 56
SQL Server, 64
FLOAT, функция, PostgreSQL, 601
FLOAT4, тип данных, PostgreSQL, 60
FLOAT4, функция, PostgreSQL, 601
FLOAT8, тип данных, PostgreSQL, 60
FLOOR, функция, 540
FOREIGN KEY, ограничение, 69
FORMAT, функция, MySQL, 557
FORMATMESSAGE, функция, SQL Server, 614
FOUND_ROWS, функция, MySQL, 557
Foundation:1999, раздел SQL99, 24
FREETEXT, функция, SQL Server, 614
FREETEXTTABLE, функция, SQL Server, 615
FROM_DAYS, функция, MySQL, 558
FROM_TZ, функция, Oracle, 575
FROM_UNIXTIME, функция, MySQL, 558

FULLTEXTCATALOGPROPERTY, функция, SQL Server, 615
FULLTEXTSERVICEPROPERTY, функция, SQL Server, 615

G

GEOGRAPHY, тип данных, 48
GEOMETRY, тип данных, 48
GETANSINULL, функция, PostgreSQL, 615
GETDATE, функция, PostgreSQL, 615
GETLOCK, функция, MySQL, 558
GETUTCDATE, функция, PostgreSQL, 615
GRANT, оператор, 310
  MySQL, 314
  Oracle, 319
  PostgreSQL, 330
  SQL Server, 332
GRAPHIC, тип данных, 48
GREATEST, функция
  Oracle, 575
  MySQL, 558
GROUP_CONCAT, функция, MySQL, 558
GROUP_ID, функция, Oracle, 575
GROUPING, функция
  Oracle, 575
  SQL Server, 615
GROUPING_ID, функция, Oracle, 576

H

HEIGHT, функция, PostgreSQL, 601
HEX, функция, MySQL, 558
HEXTORAW, функция, Oracle, 576
HIERARCHYID, тип данных, SQL Server, 64
HOST, функция, PostgreSQL, 601
HOST_ID, функция, SQL Server, 616
HOST_NAME, функция, SQL Server, 616
HOUR, функция, MySQL, 559

I

IDENT_CURRENT, функция, SQL Server, 616
IDENT_INCR, функция, SQL Server, 616
IDENT_SEED, функция, SQL Server, 616
IDENTITY, функция, SQL Server, 616
IF, функция, MySQL, 559
IFNULL, функция, MySQL, 559
IN, оператор, 40, 80, 339
INDEX_COL, функция, SQL Server, 616
INDEXPROPERTY, функция, SQL Server, 616
INET, тип данных, PostgreSQL, 60
INET_ATON, функция, MySQL, 559
INET_NTOA, функция, MySQL, 559
INITCAP, функция
  Oracle, 576
  PostgreSQL, 601
INSERT, оператор, 341
  MySQL, 345
  Oracle, 346
  PostgreSQL, 351
  SQL Server, 351
INSERT, функция, MySQL, 559
INSERTCHILDXML, функция, Oracle, 576
INSERTXMLBEFORE, функция, Oracle, 576
INSTR, функция
  MySQL, 559
  Oracle, 576
INT, тип данных, 48
  MySQL, 53
  PostgreSQL, 60
  SQL Server, 64
INT2, тип данных, 48
INT4, тип данных, PostgreSQL, 60
INT8, тип данных, PostgreSQL, 58
INTEGER, тип данных, 48
  MySQL, 53
  Oracle, 56
  PostgreSQL, 60
INTEGER, функция, PostgreSQL, 601
INTERSECT, оператор, 354
INTERVAL DAY TO SECOND, тип данных, 48, 56
INTERVAL YEAR TO MONTH, тип данных, 48, 56
INTERVAL, тип данных, PostgreSQL, 60
INTERVAL, функция
  MySQL, 559
  PostgreSQL, 601
IS, оператор, 358
IS_FREE_LOCK, функция, MySQL, 559
IS_MEMBER, функция, SQL Server, 616
IS_SRVROLEMEMBER, функция, SQL Server, 616
IS_USED_LOCK, функция, MySQL, 560

ISCLOSED, функция, PostgreSQL, 601
ISDATE, функция, SQL Server, 617
ISFINITE, функция, PostgreSQL, 601
ISNULL, функция
MySQL, 560
SQL Server, 617
ISNUNERIC, функция, SQL Server, 617
ISOPEN, функция, PostgreSQL, 602
ITERATION_NUMBER, функция, Oracle, 577

J

Java Routines and Types (JRT), 27
JOIN, подфраза, 359
MySQL, 366
Oracle, 366
PostgreSQL, 367
SQL Server, 368
JOIN, фраза, 23
JRT (Java Routines and Types), 27
JUSTIFY_DAYS, функция, PostgreSQL, 602
JUSTIFY_HOURS, функция, PostgreSQL, 602
JUSTIFY_INTERVAL, функция, PostgreSQL, 602

L

LAG, функция, Oracle, 577
LAST, функция, Oracle, 577
LAST_DAY, функция
MySQL, 560
Oracle, 577
LAST_INSERT_ID, функция, MySQL, 560
LAST_VALUE, функция, Oracle, 577
LCASE, функция, MySQL, 560
LEAD, функция, Oracle, 578
LEAST, функция
MySQL, 560
Oracle, 578
LEFT, функция
MySQL, 560
SQL Server, 617
LEN, функция, SQL Server, 617
LENGTH, функция
MySQL, 560
Oracle, 578
PostgreSQL, 602
LENGTHB, функция, Oracle, 578
LIKE, оператор, 40, 368
LINE, тип данных, PostgreSQL, 60
LN, функция, 540
LNNVL, функция, Oracle, 578
LOAD_FILE, функция, MySQL, 560
LOCALTIMESTAMP, функция, Oracle, 578
LOCATE, функция, MySQL, 560
LOG, функция
MySQL, 561
Oracle, 579
PostgreSQL, 602
SQL Server, 617
LOG10, функция
MySQL, 561
SQL Server, 617
LOG2, функция, MySQL, 561
LONG RAW, тип данных, Oracle, 57
LONG VARCHAR, тип данных, 48
LONG VARGRAPHIC, тип данных, 48
LONG, тип данных, Oracle, 57
LONGBLOB, тип данных, MySQL, 53
LONGTEXT, тип данных, MySQL, 53
LOWER, функция, 547
LPAD, функция
MySQL, 561
Oracle, 579
PostgreSQL, 602
LSEG, тип данных, PostgreSQL, 60
LSEG, функция, PostgreSQL, 603
LTRIM, функция
MySQL, 561
Oracle, 579
PostgreSQL, 603
SQL Server, 617

M

MACADDR, тип данных, PostgreSQL, 61
MAKE_REF, функция, Oracle, 579
MAKE_SET, функция, MySQL, 561
MAKEDATE, функция, MySQL, 561
MAKETIME, функция, MySQL, 561
Management of External Data (MED), 27
MASKLEN, функция, PostgreSQL, 603
MATCH, функция, MySQL, 562
MAX, функция, 511
MD5, функция
MySQL, 562
PostgreSQL, 603
MED (Management of External Data), 27

MEDIAN, функция, Oracle, 579
MEDIUMBLOB, тип данных, MySQL, 53
MEDIUMINT, тип данных, MySQL, 53
MEDIUMTEXT, тип данных, MySQL, 53
MERGE, оператор, 372
  Oracle, 375
  SQL Server, 375
MICROSECOND, функция, MySQL, 562
MID, функция, MySQL, 562, 567
MIN, функция, 511
MINUTE, функция, MySQL, 562
MOD, функция, 541
MONEY, тип данных, 48
  PostgreSQL, 61
  SQL Server, 64
MONTH, функция
  MySQL, 562
  SQL Server, 617
MONTHNAME, функция, MySQL, 562
MONTHS_BETWEEN, функция, Oracle, 579
MySQL, 30
  ALTER TABLE, оператор, 177
  ALTER VIEW, оператор, 259
  CREATE SCHEMA, оператор, 168
  CREATE TABLE, оператор, 176
  CREATE TRIGGER, оператор, 231
  CREATE VIEW, оператор, 259
  DECLARE CURSOR, оператор, 270
  DELETE, оператор, 277
  DROP, операторы, 289
  FETCH, оператор, 306
  GRANT, оператор, 314
  INSERT, оператор, 345
  JOIN, подфраза, 366
  ORDER BY, фраза, 381
  ROLLBACK, оператор, 401
  SELECT, оператор, 422
  SET TRANSACTION, оператор, 468
  SET, оператор, 452
  START TRANSACTION, оператор, 473
  TRUNCATE TABLE, оператор, 483
  UPDATE, оператор, 492
  битовые операторы, 39
  идентификаторы, 34
  типы данных, 50
MySQLREVOKE, оператор, 391

N

NANVL, функция, Oracle, 579
NATIONAL CHAR VARYING, тип данных, 49
  Oracle, 57
  SQL Server, 65
NATIONAL CHAR, тип данных, 49
  Oracle, 57
  SQL Server, 65
NATIONAL CHARACTER VARYING, тип данных, 49
  Oracle, 57
  SQL Server, 65
NATIONAL CHARACTER, тип данных, 49
  Oracle, 57
  SQL Server, 65
NATIONAL TEXT, тип данных, SQL Server, 65
NCHAR VARYING, тип данных, 49
NCHAR, тип данных, 49
  MySQL, 53
  Oracle, 57
  SQL Server, 65
NCHAR, функция
  SQL Server, 618
NCLOB, тип данных, 49, 57
NETMASK, функция, PostgreSQL, 603
NETWORK, функция, PostgreSQL, 603
NEW_TIME, функция, Oracle, 579
NEWID, функция, SQL Server, 618
NEXT_DAY, функция, Oracle, 580
NEXTVAL, функция, PostgreSQL, 603
NLS_CHARSET_DECL_LEN, функция, Oracle, 580
NLS_CHARSET_ID, функция, Oracle, 580
NLS_CHARSET_NAME, функция, Oracle, 580
NLS_INITCAP, функция, Oracle, 581
NLS_LOWER, функция, Oracle, 581
NLS_UPPER, функция, Oracle, 581
NLSSORT, функция, Oracle, 581
NOT, оператор, 40
NOW, функция, 529, 530, 562
NPOINTS, функция, PostgreSQL, 603
NTEXT, тип данных, SQL Server, 49, 65
NTILE, функция, Oracle, 581
NULL, значение, 19

NULLIF, функция
MySQL, 562
Oracle, 581
PostgreSQL, 603
SQL Server, 618
NUMBER, тип данных, 49, 57
NUMERIC, тип данных, 49
MySQL, 53
PostgreSQL, 61
SQL Server, 65
NUMTODSINTERVAL, функция, Oracle, 581
NUMTOYMINTERVAL, функция, Oracle, 581
NVARCHAR, тип данных, 49
MySQL, 53
SQL Server, 65
NVARCHAR2, тип данных, Oracle, 57
NVL, функция, Oracle, 582
NVL2, функция, Oracle, 582

O

Object Language Binding (OBJ), 27
OBJECT_ID, функция, SQL Server, 618
OBJECT_NAME, функция, SQL Server, 618
OBJECTPROPERTY, функция, SQL Server, 618
OCT, функция, MySQL, 563
OCTET_LENGTH, функция, 534
OID, тип данных, PostgreSQL, 61
OLAP (Online Analytical Processing), 25
OLD_PASSWORD, функция, MySQL, 563
Online Analytical Processing (OLAP), 25
OPEN, оператор, 377
OPEN, функция, SQL Server, 618
OPENDATASOURCE, функция, SQL Server, 618
OPENQUERY, функция, SQL Server, 618
OPENROWSET, функция, SQL Server, 618
OR, оператор, 40
ORA_HASH, функция, Oracle, 582
Oracle, 30
ALTER ROLE, оператор, 163
ALTER TABLE, оператор, 188
ALTER TRIGGER, оператор, 232
ALTER VIEW, оператор, 260
CREATE ROLE, оператор, 163
CREATE SCHEMA, оператор, 168
CREATE TABLE, оператор, 187
CREATE TRIGGER, оператор, 231
CREATE VIEW, оператор, 260
DECLARE CURSOR, оператор, 270
DELETE, оператор, 279
DISCONNECT, оператор, 286
DROP, операторы, 290
EXCEPT, оператор, 300
FETCH, оператор, 306
GRANT, оператор, 319
INSERT, оператор, 346
JOIN, подфраза, 366
MERGE, оператор, 375
ORDER BY, фраза, 382
REVOKE, оператор, 393
ROLLBACK, оператор, 401
SELECT, оператор, 426
SET CONSTRAINT, оператор, 457
SET ROLE, оператор, 459
SET TIME ZONE, оператор, 464
SET TRANSACTION, оператор, 468
TRUNCATE TABLE, оператор, 484
UNION, оператор, 486
UPDATE, оператор, 492
идентификаторы, 34
пространственные данные, 55
типы данных, 55
ORD, функция, MySQL, 563
ORDER BY, фраза, 379
MySQL, 381
Oracle, 382
PostgreSQL, 382
SQL Server, 382
OVERLAY, функция, 548

P

PARSENAME, функция, SQL Server, 618
PASSWORD, функция, MySQL, 563
PATH, тип данных, PostgreSQL, 61
PATH, функция
Oracle, 582
PostgreSQL, 603
PATINDEX, функция, SQL Server, 619
PCLOSE, функция, PostgreSQL, 604
PERCENT_RANK, функция, 512, 526
PERCENTILE_CONT,, функция, 513
PERCENTILE_DISC, функция, 514
PERIOD_ADD, функция, MySQL, 563

PERIOD_DIFF, функция, MySQL, 563
PERMISSIONS, функция, SQL Server, 619
Persistent Stored Modules (PSM), 26
PI, функция
  MySQL, 563
  PostgreSQL, 604
  SQL Server, 619
PL/pgSQL, 29
PL/SQL, 29
POINT, тип данных, PostgreSQL, 61
POINT, функция, PostgreSQL, 604
POLYGON, функция, PostgreSQL, 604
POPEN, функция, PostgreSQL, 604
POSITION, функция, 541, 560
PostgreSQL, 30
  ALTER ROLE, оператор, 164
  ALTER SCHEMA, оператор, 169
  ALTER TABLE, оператор, 220
  ALTER TRIGGER, оператор, 236
  CREATE ROLE, оператор, 164
  CREATE SCHEMA, оператор, 169
  CREATE TABLE, оператор, 219
  CREATE TRIGGER, оператор, 236
  CREATE VIEW, оператор, 263
  DECLARE CURSOR, оператор, 271
  DELETE, оператор, 281
  DROP, операторы, 293
  EXCEPT, оператор, 301
  FETCH, оператор, 307
  GRANT, оператор, 330
  INSERT, оператор, 351
  JOIN, подфраза, 367
  ORDER BY фраза, 382
  ROLLBACK, оператор, 402
  SELECT, оператор, 440
  SET CONSTRAINT, оператор, 457
  SET SESSION AUTHORIZATION, оператор, 463
  SET TIME ZONE, оператор, 465
  SET TRANSACTION, оператор, 470
  SET, оператор, 453
  START TRANSACTION, оператор, 474
  TRUNCATE TABLE, оператор, 484
  UNION, оператор, 487
  UPDATE, оператор, 495
  идентификаторы, 34
  типы данных, 58
PostgreSQLREVOKE, оператор, 396
POW, функция
  MySQL, 563, 605
POWER, функция, 542, 563
POWERMULTISET, функция, Oracle, 582
POWERMULTISET_BY_CARDINALITY, функция, Oracle, 582
PREDICTION_, функции, Oracle, 583
PRESENTNNV, функция, Oracle, 583
PRESENTV, функция, Oracle, 583
PREVIOUS, функция, Oracle, 583
PRIMARY KEY ограничения, 68
PSM (Persistent Stored Modules), 26

Q

QUARTER, функция, MySQL, 563
QUOTE, функция, MySQL, 564
QUOTE_IDENT, функция, PostgreSQL, 605
QUOTE_LITERAL, функция, PostgreSQL, 605

R

RADIANS, функция
  MySQL, 564
  PostgreSQL, 605
  SQL Server, 619
RADIUS, функция, PostgreSQL, 605
RAND, функция
  MySQL, 564
  SQL Server, 619
RANDOM, функция, PostgreSQL, 605
RANK, функция, 515, 527
RATIO_TO_REPORT, функция, Oracle, 583
RAW, тип данных, Oracle, 57
RAWTOHEX, функция, Oracle, 583
RAWTONHEX, функция, Oracle, 583
REAL тип данных, 49
  MySQL, 54
  Oracle, 57
  PostgreSQL, 60
  SQL Server, 65
REF, функция, Oracle, 583
REFTOHEX, функция, Oracle, 584
REGEXP, функция, MySQL, 564
REGEXP_INSTR, функция, Oracle, 584
REGEXP_MATCHES, функция, PostgreSQL, 605

REGEXP_REPLACE, функция
Oracle, 584
PostgreSQL, 605
REGEXP_SPLIT_TO_ARRAY, функция, PostgreSQL, 606
REGEXP_SPLIT_TO_TABLE, функция, PostgreSQL, 606
REGEXP_SUBSTR, функция, Oracle, 584
REGR_, семейство функций, 516
RELEASE SAVEPOINT, оператор, 383
RELEASE_LOCK, функция, MySQL, 564
RELTIME, функция, PostgreSQL, 606
REMAINDER, функция, Oracle, 585
REPEAT, функция, MySQL, 564
REPLACE, функция
MySQL, 564
Oracle, 585
SQL Server, 619
REPLICATE, функция, SQL Server, 619
RETURN, оператор, 385
REVERSE, функция
MySQL, 564
SQL Server, 619
REVOKE, оператор, 387
MySQL, 391
Oracle, 393
PostgreSQL, 396
SQL Server, 397
RIGHT, функция
MySQL, 564
SQL Server, 619
ROLLBACK, оператор, 399
MySQL, 401
Oracle, 401
PostgreSQL, 402
SQL Server, 403
ROUND, функция
MySQL, 565
Oracle, 585
PostgreSQL, 606
SQL Server, 620
ROW_COUNT, функция, MySQL, 565
ROW_NUMBER,, функция, 528
ROWCOUNT_BIG, функция, SQL Server, 620
ROWID, тип данных, Oracle, 57
ROWIDTOCHAR, функция, Oracle, 585
ROWIDTONCHAR, функция, Oracle, 585
ROWVERSION, тип данных, SQL Server, 65
RPAD, функция
MySQL, 565
Oracle, 586
PostgreSQL, 606
RTRIM, функция
MySQL, 565
Oracle, 586
PostgreSQL, 606
SQL Server, 620

S

SAVEPOINT, оператор, 404
SCHEMA, функция, MySQL, 565
Schemata, 24
SCN_TO_TIMESTAMP, функция, Oracle, 586
SEC_TO_TIME, функция, MySQL, 565
SECOND, функция, MySQL, 565
SELECT, оператор, 406
JOIN фраза, 23
MySQL, 422
Oracle, 426
PostgreSQL, 440
SQL Server, 443
SELECT, фраза, 23
SEQUEL (Structured English Query Language), 15
SERIAL, тип данных, 49
MySQL, 54
PostgreSQL, 61
SERIAL4 тип данных PostgreSQL,, 61
SERIAL8 тип данных, PostgreSQL, 61
SESSION_USER, функция, 529, 565
MySQL, 569
SESSIONTIMEZONE, функция, Oracle, 586
SET CONNECTION, оператор, 454
SQL Server, 455
SET CONSTRAINT, оператор, 456
Oracle, 457
PostgreSQL, 457
SET PATH, оператор, 457
SET ROLE, оператор, 458
Oracle, 459
SET SCHEMA, оператор, 461
SET SESSION AUTHORIZATION, оператор, 462
PostgreSQL, 463

SET TIME ZONE, оператор, 463
Oracle, 464
PostgreSQL, 465
SET TRANSACTION, оператор, 466
MySQL, 468
Oracle, 468
PostgreSQL, 470
SQL Server, 470
SET, оператор, 451
MySQL, 452
PostgreSQL, 453
SQL Server, 454
SET, функция, Oracle, 586
SET_MASKLEN, функция, PostgreSQL, 606
SETSEED, функция, PostgreSQL, 607
SETVAL, функция, PostgreSQL, 607
SHA, функция, MySQL, 565
SHA1, функция, MySQL, 565
SIGN, функция
MySQL, 565
Oracle, 586
PostgreSQL, 607
SQL Server, 620
SIN, функция
MySQL, 566
Oracle, 586
PostgreSQL, 607
SQL Server, 620
SINH, функция, Oracle, 586
SLEEP, функция, MySQL, 566
SMALLDATETIME, тип данных, SQL Server, 65
SMALLINT, тип данных, 49
MySQL, 54
Oracle, 57
PostgreSQL, 62
SQL Server, 65
SMALLMONEY, тип данных, SQL Server, 65
SOME, оператор, 40, 79
SOUNDEX, функция
MySQL, 566
Oracle, 586
SQL Server, 620
SPACE, функция
MySQL, 566
SQL Server, 620
SQL (Structured Query Language)
альтернативы, 21
история, 15
SQL Server, 31
ALTER ROLE, оператор, 166
ALTER SCHEMA, оператор, 170
ALTER TABLE, оператор, 223
ALTER TRIGGER, оператор, 237
ALTER VIEW, оператор, 263
BEGIN TRANSACTION, оператор, 475
CREATE ROLE, оператор, 166
CREATE SCHEMA, оператор, 169
CREATE TABLE, оператор, 222
CREATE TRIGGER, оператор, 237
CREATE VIEW, оператор, 263
DECLARE CURSOR, оператор, 272
DELETE, оператор, 282
DISCONNECT, оператор, 287
DROP, операторы, 295
EXCEPT, оператор, 301
FETCH, оператор, 309
GRANT, оператор, 332
INSERT, оператор, 351
JOIN подфраза, 368
MERGE, оператор, 375
ORDER BY фраза, 382
REVOKE, оператор, 397
ROLLBACK, оператор, 403
SELECT, оператор, 443
SET CONNECTION, оператор, 455
SET TRANSACTION, оператор, 470
SET, оператор, 454
TRUNCATE TABLE, оператор, 484
UNION, оператор, 487
UPDATE, оператор, 495
битовые операторы, 39
идентификаторы, 34
типы данных, 62
SQL/CLI, 26
SQL/Foundation, 26
SQL/Framework, 26
SQL/JRT (Java Routines and Types), 27
SQL/MED (Management of External Data), 27
SQL/OBJ (Object Language Binding), 27
SQL/PSM (Persistent Stored Modules), 26
SQL/Schemata, 27
SQL/XML, 27
SQL_VARIANT тип данных, SQL Server, 65
SQL2 (SQL92), 24, 25
SQL2006, 15, 24

SQL3(2003)
дополнительные пакеты возможностей, 26
классы операторов, 28
новые возможности, 25
соответствие, 24, 26
SQL99, 25
SQRT, функция, 543
START TRANSACTION, оператор, 471
MySQL, 473
STATEMENT_TIMESTAMP, функция, PostgreSQL, 607
STATS_, функции, Oracle, 587
STATS_DATE, функция, SQL Server, 620
STD, функция, MySQL, 566
STDDEV, функция
MySQL, 566
Oracle, 587
STDDEV_POP, функция, 517
STDDEV_SAMP, функция, 518
STDEV, функция, SQL Server, 620
STDEV_POP, функция, Oracle, 587
STDEV_SAMP, функция, Oracle, 587
STDEVP, функция, SQL Server, 621
STR, функция, SQL Server, 621
STR_TO_DATE, функция, MySQL, 566
STRCMP, функция, MySQL, 566
STRING_TO_ ARRAY, функция, PostgreSQL, 607
STUFF, функция, SQL Server, 621
SUBDATE, функция, MySQL, 554
SUBSTR, функция, Oracle, 587
SUBSTRB, функция, Oracle, 587
SUBSTRING, функция, 548
MySQL, 566
PostgreSQL, 607
SQL Server, 621
SUBSTRING_INDEX, функция, MySQL, 567
SUBTIME, функция, MySQL, 567
SUM, функция, 504
SUSER_ID, функция, SQL Server, 621
SUSER_SID, функция, SQL Server, 621
SUSER_SNAME, функция, SQL Server, 621
SYS_CONNECT_BY_PATH, функция, Oracle, 587
SYS_CONTEXT, функция, Oracle, 588
SYS_DBURIGEN, функция, Oracle, 588
SYS_EXTRACT_UTC, функция, Oracle, 588
SYS_GUID, функция, Oracle, 588
SYS_TYPEID, функция, Oracle, 588
SYS_XMLAGG, функция, Oracle, 588
SYS_XMLGEN, функция, Oracle, 588
SYSDATE, функция
MySQL, 562
Oracle, 588
SYSTEM_USER, функция, 529
MySQL, 569
SYSTIMESTAMP, функция, Oracle, 589

T

TABLE, тип данных, SQL Server, 66
TABLESAMPLE, фраза, 25
TAN, функция
MySQL, 567
Oracle, 589
PostgreSQL, 607
TAN, функция, SQL Server, 621
TANH, функция, Oracle, 589
TEXT, тип данных, 49
MySQL, 54
PostgreSQL, 62
SQL Server, 66
TEXT, функция, PostgreSQL, 607
TEXTRTR, функция, SQL Server, 621
TEXTVALID, функция, SQL Server, 622
TIME, тип данных, 49
MySQL, 54
PostgreSQL, 62
SQL Server, 66
TIME, функция, MySQL, 567
TIME_FORMAT, функция, MySQL, 567
TIME_TO_SEC, функция, MySQL, 567
TIMEDIFF, функция, MySQL, 567
TIMEOFDAY, функция, PostgreSQL, 607
TIMESPAN тип данных, 49
TIMESTAMP WITH TIMEZONE, тип данных, 50
TIMESTAMP, тип данных, 49
MySQL, 54
Oracle, 57
PostgreSQL, 62
SQL Server, 66
TIMESTAMP, функция,
MySQL, 567
PostgreSQL, 607

TIMESTAMP_TO_SCN, функция, Oracle, 589
TIMESTAMPADD, функция, MySQL, 568
TIMESTAMPDIFF, функция, MySQL, 568
TIMESTAMPZ, тип данных, 50
TIMETZ, тип данных, PostgreSQL, 50, 62
TINYBLOB, тип данных, 50
TINYINT, тип данных, 50
TINYTEXT, тип данных, 50
TO_BINARY_DOUBLE, функция, Oracle, 589
TO_BINARY_FLOAT, функция, Oracle, 589
TO_CHAR, функция
  Oracle, 589, 590, 591
  PostgreSQL, 608
TO_CLOB, функция, Oracle, 591
TO_DATE, функция
  Oracle, 591
  PostgreSQL, 608
TO_DAYS, функция, MySQL, 568
TO_DSINTERVAL, функция, Oracle, 591
TO_LOB, функция, Oracle, 591
TO_MULTIBYTE, функция, Oracle, 591
TO_NCHAR, функция, Oracle, 592
TO_NCLOB, функция, Oracle, 592
TO_NUMBER, функция,
  Oracle, 592
  PostgreSQL, 609
TO_SINGLE_BYTE, функция, Oracle, 592
TO_TIMESTAMP, функция,
  Oracle, 592
  PostgreSQL, 609
TO_TIMESTAMP_TZ, функция, Oracle, 592
TO_YMINTERVAL, функция, Oracle, 592
Transact-SQL, 29
TRANSACTION_TIMESTAMP, функция, PostgreSQL, 610
TRANSLATE, функция, 546
  Oracle, 592, 593
  PostgreSQL, 610
TREAT, функция, Oracle, 593
TRIM, функция, 550
TRUNC, функция
  Oracle, 593
TRUNC, функция, PostgreSQL, 610
TRUNCATE TABLE, оператор, 482
  MySQL, 483
  Oracle, 484
  PostgreSQL, 484
  SQL Server, 484
TRUNCATE, функция, MySQL, 568
TYPEPROPERTY, функция, SQL Server, 622
TZ_OFFSET, функция, Oracle, 593

U

UCASE, функция, MySQL, 568
UID, функция, Oracle, 593
UNCOMPRESS, функция, MySQL, 568
UNCOMPRESS_LENGTH, функция, MySQL, 568
UNHEX, функция, MySQL, 568
UNICODE, функция, SQL Server, 622
UNION JOIN, фраза, удалена в SQL99, 25
UNION, оператор, 485
  Oracle, 486
  PostgreSQL, 487
  SQL Server, 487
UNIQUE, ограничение, 72
UNIQUEIDENTIFIER, тип данных, SQL Server, 66
UNISTR, функция, Oracle, 593
UNIX_TIMESTAMP, функция, MySQL, 568
UNSIGNED, атрибут типов данных, MySQL, 50
UPDATE, оператор, 488
  MySQL, 492
  Oracle, 492
  PostgreSQL, 495
  SQL Server, 495
UPDATE...SET ROW, оператор, удален в SQL99, 25
UPDATEXML, функция, MySQL, 569
UPDATEXML, функция, Oracle, 594
UPPER, функция, 547
UROWID тип данных, Oracle, 58
user (AuthorizationID), 19
USER, функция, 529
  MySQL, 569
USER_ID, функция, SQL Server, 622
USER_NAME, функция, SQL Server, 622
USERENV, функция, Oracle, 594
UTC_DATE, функция, MySQL, 569

UTC_TIME, функция, MySQL, 569
UTC_TIMESTAMP, функция, MySQL, 569
UUID, функция, MySQL, 569

V

VALUE, функция, Oracle, 594
VAR, функция, SQL Server, 622
VAR_POP, функция, 519
  Oracle, 594
VAR_SAMP, функция, 519
  Oracle, 594
VARBINARY, тип данных, 50
  MySQL, 54
  SQL Server, 66
VARBIT, тип данных, PostgreSQL, 59
VARCHAR FOR BIT DATA, тип данных, 50
VARCHAR, тип данных, 50
  Oracle, 58
  PostgreSQL, 62
  SQL Server, 66
VARCHAR, функция, PostgreSQL, 610
VARCHAR2 тип данных, Oracle, 58
VARGRAPHIC, тип данных, 50
VARIANCE, функция
  MySQL, 569
VARIANCE, функция, Oracle, 594
VARP, функция, SQL Server, 623
VERSION, функция, MySQL, 569
VSIZE, функция, Oracle, 595

W

WEEK, функция, MySQL, 569
WEEKDAY, функция, MySQL, 569
WEEKOFYEAR, функция, MySQL, 570
WHERE, фраза, 23, 497
WIDTH, функция, PostgreSQL, 610
WIDTH_BUCKET, функция, 543

X

XML
  в SQL2006, 24
  в SQL3, 27
XML, тип данных, 50
XML, функция, Oracle, 595
XMLAGG, функция, Oracle, 595
XMLCAST, функция, Oracle, 595
XMLCOLATTVAL, функция, Oracle, 595
XMLCONCAT, функция, Oracle, 595
XMLDIFF, функция, Oracle, 595
XMLELEMENT, функция, Oracle, 595
XMLEXISTS, функция, Oracle, 595
XMLFOREST, функция, Oracle, 595
XMLPARSE, функция, Oracle, 596
XMLPATCH, функция, Oracle, 596
XMLPI, функция, Oracle, 596
XMLQUERY, функция, Oracle, 596
XMLROOT, функция, Oracle, 596
XMLSEQUENCE, функция, Oracle, 596
XMLSERIALIZE, функция, Oracle, 596
XMLTABLE, функция, Oracle, 596
XMLTRANSFORM, функция, Oracle, 596
XMLTYPE, тип данных, 50
XOR, функция, MySQL, 570

Y

YEAR, тип данных, MySQL, 50, 55
YEAR, функция, MySQL, 570
YEARWEEK, функция, MySQL, 570

Z

ZEROFILL атрибут типов данных, MySQL, 51

А

агрегатные функции, 502
  AVG, 504
  CORR, 505
  COUNT, 506
  COVAR_POP, 507
  COVAR_SAMP, 508
  CUME_DIST, 508, 524
  DENSE_RANK, 510, 525
  MAX, 511
  MIN, 511
  PERCENT_RANK, 512
  PERCENTILE_CONT, 513
  PERCENTILE_DISC, 514
  RANK, 515, 527
  REGR, 516
  STDDEV_POP, 517
  STDDEV_SAMP, 518
  SUM, 504
  VAR_POP, 519
  VAR_SAMP, 519
  как оконные функции, 524

арифметические операторы, 38
атомарность, 19

Б

бинарные типы данных, 45
битовые операторы, 39
    оператор И (&), 39
    оператор ИЛИ (|), 39
    оператор исключающее ИЛИ (^), 39
    оператор отрицания (~), 41

В

взятие остатка при делении, арифметический оператор (%), 39
внешние ключи, 69
встроенные скалярные функции, 528
выборка, 22
вычитание, арифметический оператор (-), 39

Д

декларативная обработка, 21
деление, арифметический оператор (/), 39
детерминированные функции, 501
диагностические операторы, 28
диалекты SQL, 29
Дьюи, Мелвил, 22

З

зарезервированные слова, 44

И

идентификаторы, 31
    ограничения, 33
    правила именования, 32
    разделители, 33
    уникальность, 36

К

Кантор Георг, основоположник теории множеств, 22
каталог, 18, 31
кластер, 18, 31
ключевые слова, 31, 44, 624
ключи внешние, 69
ключи первичные, 17, 68
Кодд, Эдгар
    автор теории реляционных баз данных, 16
    правила для реляционных баз данных, 16, 17, 19, 20, 21
    разработчик языка SEQUEL, 15
кодировка, 19
коллекции, 45
комментарии, 43
конкатенации оператор (+), SQL Server, 42
контроль доступа, 19
кортежи, 17

Л

литералы, 37
логические операторы, 40

М

метаданные, 20

Н

недетерминированная, функция, 501
неопределенное значение, 19
нормализация, 16

О

объекты, 18, 32
ограничения, 18
ограничения на уровне таблицы, 67
ограничения целостности, 67
    CHECK, 73
    FOREIGN KEY, 69
    PRIMARY KEY, 68
    UNIQUE, 72
    синтаксис, 67
оконные функции, 502, 520
    CUME_DIST, 508, 524
    DENSE_RANK, 510, 525
    PERCENT_RANK, 512, 526
    RANK, 515, 527
    ROW_NUMBER, 528
    синтаксис в Oracle, 521
    синтаксис в SQL Server, 521
    фраза framing, 523
    фраза ordering, 522
    фраза partitioning, 522

операторы
арифметические, 38
битовые, 39
категории, 38
конкатенации (||), 544
логические операторы, 40
приоритет, 41
присваивания, 39
сравнения, 39
унарные, 40
операции над множествами, 21

П

первичные ключи, 68
подзапросы, 476
подключения операторы, 28
построчная обработка, 21
правила именования идентификаторов, 32
приоритет операторов, 41
присваивания оператор (:=), 39
проверки значений, 19
проекция, 22
процедурное программирование, 21
псевдонимов присваивание, 39

Р

работы с данными операторы, 28
разделители, 42
идентификаторов, 33
регистр символов в идентификаторах, 32
реляционные системы управления базами данных (РСУБД), 15
история, 15
правила, 16

С

сессией управления операторы, 28
символьные литералы, 37
символьные типы данных, 45
скалярные функции, 502
ABS, 534
BIT_LENGTH, 534
CASE, 530
CAST, 532
CEIL, 535
CHAR_LENGTH, 534
CONVERT, 546
CURRENT_DATE, 529
CURRENT_TIME, 529
CURRENT_TIMESTAMP, 529
CURRENT_USER, 529
EXP, 536
EXTRACT, 536
FLOOR, 540
LN, 540
LOWER, 547
MOD, 541
NOW, 529, 530
OCTET_LENGTH, 534
OVERLAY, 548
POSITION, 541
POWER, 542
SESSION_USER, 529
SQRT, 543
SUBSTRING, 548
SYSTEM_USER, 529
TRANSLATE, 546
TRIM, 550
UPPER, 547
USER, 529
WIDTH_BUCKET, 543
встроенные, 528
строковые, 544
числовые, 533
сложение, арифметический оператор (+), 39, 42
соединения, 22, 23
union соединения, 360
внутренние соединения, 360, 362
кросс-соединения, 360, 362
левые внешние соединения, 360, 363
натуральные соединения, 359, 364
полные внешние соединения, 360, 364
правые внешние соединения, 360, 364
тэта-соединение, 23
сокращения в идентификаторах, 32
сравнения операторы, 39
стандарт SQL
история, 24
уровни соответствия, 25
столбцы, 16, 18
ограничения, 18
типы данных, 18
строки (записи), 16, 17
строковые функции, 544
структуры данных, 17
схема, 18, 31
схемой управления операторы, 28
схемы упорядочивания, 19

Т

таблицы, 16, 17
теория множеств, 22
типы данных, 44
    MySQL, 50
    Oracle, 55
    PostgreSQL, 58
    SQL Server, 62
    SQL2003, 45
    сравнение поддержки платформами, 46
транзакциями управления операторы, 28
триггер, 227
тэта-соединение, 23

У

умножение, арифметический оператор (*), 39
унарные операторы, 40
уникальный ключ, 72
управления выполнением операторы, 28
уровни соответствия стандарту, 25

Ф

Функции
    агрегатные, 502
    детерминированные, 501
    недетерминированные, 501
    оконные, 502, 520
    поддерживаемые MySQL, 551
    поддерживаемые Oracle, 570
    поддерживаемые PostgreSQL, 596
    поддерживаемые SQL Server, 610
    скалярные, 502, 528

Ч

числовые литералы, 37
числовые типы данных, 46
числовые функции, 25

Э

экземпляр, 31

Я

явные транзакции, 87

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-165-3, название «SQL. Справочник, 3е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.