Бен Форта



Освой самостоятельно SQL за 10 минут

ВИЛЬЯМС
SAMS

Освой самостоятельно

SQL

за 10 минут

4-е издание

Бен Форта



Москва ◆ Санкт-Петербург ◆ Киев
2014

ББК 32.973.26-018.2.75
Ф80
УДК 681.3.07

Издательский дом "Вильямс"
Главный редактор *С.Н. Тригуб*
Перевод с английского и редакция *В.Р. Гинзбурга*

По общим вопросам обращайтесь
в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Форта, Бен.
Ф80 SQL за 10 минут, 4-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2014. — 288 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1858-1 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Sams Publishing.

Authorized Russian translation of the English edition of Sams Teach Yourself SQL in 10 Minutes, © 2013 by Pearson Education, Inc. (ISBN 978-0-672-33607-2).

This translation is published and sold by permission of Pearson Education, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание
Бен Форта
SQL за 10 минут, 4-е издание

Литературный редактор *И.А. Попова*
Верстка *Л.В. Чернокозинская*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 08.11.2013. Формат 84x108/32.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 28,73. Уч.-изд. л. 8,82.
Тираж 2000 экз. Заказ № 1273

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1858-1 (рус.)
© Издательский дом "Вильямс", 2014
ISBN 978-0-672-33607-2 (англ.)
© Pearson Education, Inc., 2013

Оглавление

Введение 15
УРОК 1. Основы SQL 19
УРОК 2. Извлечение данных из таблиц 27
УРОК 3. Сортировка полученных данных 39
УРОК 4. Фильтрация данных 47
УРОК 5. Расширенная фильтрация данных 55
УРОК 6. Фильтрация с использованием метасимволов 65
УРОК 7. Создание вычисляемых полей 73
УРОК 8. Использование функций обработки данных 83
УРОК 9. Итоговые вычисления 93
УРОК 10. Группировка данных 105
УРОК 11. Подзапросы 115
УРОК 12. Объединение таблиц 123
УРОК 13. Создание расширенных объединений 135
УРОК 14. Комбинированные запросы 147
УРОК 15. Добавление данных 155
УРОК 16. Обновление и удаление данных 165
УРОК 17. Создание таблиц и работа с ними 173
УРОК 18. Представления 185
УРОК 19. Хранимые процедуры 197
УРОК 20. Обработка транзакций 207
УРОК 21. Курсоры 215
УРОК 22. Расширенные возможности SQL 223
ПРИЛОЖЕНИЕ А. Сценарии демонстрационных таблиц 237
ПРИЛОЖЕНИЕ Б. Работа с популярными программами 245
ПРИЛОЖЕНИЕ В. Синтаксис инструкций SQL 259
ПРИЛОЖЕНИЕ Г. Типы данных в SQL 265
ПРИЛОЖЕНИЕ Д. Зарезервированные слова SQL 273
Предметный указатель 277

Содержание

Об авторе 13

Введение 15

Для кого предназначена эта книга 16

СУБД, рассмотренные в книге 16

Условные обозначения 17

Ждем ваших отзывов! 18

Урок 1. Основы SQL 19

Терминология баз данных 19

Базы данных 20

Таблицы 20

Столбцы и типы данных 21

Строки 23

Первичные ключи 24

Что такое SQL 25

Попробуйте сами 26

Резюме 26

Урок 2. Извлечение данных из таблиц 27

Инструкция `SELECT` 27

Извлечение отдельных столбцов 28

Извлечение нескольких столбцов 30

Извлечение всех столбцов 31

Извлечение уникальных строк 32

Ограничение результатов запроса 34

Использование комментариев 37

Резюме 38

Урок 3. Сортировка полученных данных 39

Сортировка записей 39

Сортировка по нескольким столбцам 41

Сортировка по положению столбца 42

Указание направления сортировки 43
Резюме 46

Урок 4. Фильтрация данных 47
Использование предложения WHERE 47
Операторы в предложении WHERE 49
Сравнение с одиночным значением 49
Проверка на неравенство 50
Сравнение с диапазоном значений 51
Проверка на отсутствие значения 52
Резюме 54

Урок 5. Расширенная фильтрация данных 55
Комбинирование условий WHERE 55
Оператор AND 55
Оператор OR 57
Порядок обработки операторов 58
Оператор IN 60
Оператор NOT 61
Резюме 63

Урок 6. Фильтрация с использованием метасимволов 65
Использование оператора LIKE 65
Метасимвол "знак процента" (%) 66
Метасимвол "знак подчеркивания" (_) 69
Метасимвол "квадратные скобки" ([]) 70
Советы по использованию метасимволов 72
Резюме 72

Урок 7. Создание вычисляемых полей 73
Что такое вычисляемые поля 73
Конкатенация полей 74
Использование псевдонимов 78
Выполнение математических вычислений 80
Резюме 82

Урок 8. Использование функций обработки данных 83
Что такое функция 83
Проблемы с функциями 83

Применение функций 85
Функции для работы с текстом 85
Функции для работы с датой и временем 88
Функции для работы с числами 91
Резюме 92

Урок 9. Итоговые вычисления 93
Использование итоговых функций 93
Функция AVG() 94
Функция COUNT() 96
Функция MAX() 97
Функция MIN() 98
Функция SUM() 99
Итоговые вычисления для уникальных значений 101
Комбинирование итоговых функций 103
Резюме 103

Урок 10. Группировка данных 105
Принципы группировки данных 105
Создание групп 106
Фильтрация по группам 108
Группировка и сортировка 111
Порядок предложений в инструкции SELECT 114
Резюме 114

Урок 11. Подзапросы 115
Что такое подзапросы 115
Фильтрация с помощью подзапросов 115
Использование подзапросов в качестве вычисляемых полей 119
Резюме 122

Урок 12. Объединение таблиц 123
Что такое объединение 123
Что такое реляционные таблицы 123
Зачем нужны объединения 125
Создание объединения 126
Важность предложения WHERE 127
Внутренние объединения 130
Объединение нескольких таблиц 131
Резюме 134

Урок 13. Создание расширенных объединений 135
Использование псевдонимов таблиц 135
Объединения других типов 136
Самообъединения 137
Естественные объединения 139
Внешние объединения 140
Использование объединений совместно с итоговыми функциями 143
Правила создания объединений 145
Резюме 145

Урок 14. Комбинированные запросы 147
Что такое комбинированные запросы 147
Создание комбинированных запросов 148
Использование оператора UNION 148
Правила применения оператора UNION 151
Включение или исключение повторяющихся строк 151
Сортировка результатов комбинированных запросов 153
Резюме 154

Урок 15. Добавление данных 155
Способы добавления данных 155
Добавление полных строк 155
Добавление части строки 159
Добавление результатов запроса 160
Копирование данных из одной таблицы в другую 162
Резюме 164

Урок 16. Обновление и удаление данных 165
Обновление данных 165
Удаление данных 168
Советы по обновлению и удалению данных 170
Резюме 171

Урок 17. Создание таблиц и работа с ними 173
Создание таблиц 173
Создание простой таблицы 174
Работа со значениями NULL 176
Определение значений по умолчанию 178
Обновление таблиц 179

Удаление таблиц 182
Переименование таблиц 183
Резюме 183

Урок 18. Представления 185
Что такое представления 185
Зачем нужны представления 186
Правила и ограничения представлений 187
Создание представлений 189
Использование представлений для упрощения сложных объединений 189
Использование представлений для переформатирования извлекаемых данных 191
Использование представлений для фильтрации нежелательных данных 193
Использование представлений с вычисляемыми полями 194
Резюме 196

Урок 19. Хранимые процедуры 197
Что такое хранимые процедуры 197
Зачем нужны хранимые процедуры 198
Выполнение хранимых процедур 200
Создание хранимых процедур 201
Резюме 206

Урок 20. Обработка транзакций 207
Что такое транзакции 207
Управление транзакциями 209
Инструкция ROLLBACK 211
Инструкция COMMIT 211
Точки сохранения 212
Резюме 214

Урок 21. Курсоры 215
Что такое курсоры 215
Работа с курсорами 217
Создание курсоров 217
Управление курсорами 218
Закрытие курсоров 221
Резюме 221

Урок 22. Расширенные возможности SQL 223
Что такое ограничения 223
Первичные ключи 224
Внешние ключи 226
Ограничения уникальности 228
Ограничения на значения столбца 229
Что такое индексы 230
Что такое триггеры 233
Безопасность баз данных 235
Резюме 236

Приложение А. Сценарии демонстрационных таблиц 237
Демонстрационные таблицы 237
Описания таблиц 238
Получение демонстрационных таблиц 242
Загрузка готовых баз данных 242
Загрузка SQL-сценариев для различных СУБД 243

Приложение Б. Работа с популярными программами 245
Apache OpenOffice Base 245
Adobe ColdFusion 246
IBM DB2 246
MariaDB 247
Microsoft Access 247
Microsoft ASP 248
Microsoft ASP.NET 249
Microsoft Query 249
Microsoft SQL Server (включая Microsoft SQL Server Express) 250
MySQL 251
Oracle 252
Oracle Express 253
PHP 254
PostgreSQL 254
SQLite 255
Конфигурирование источников данных ODBC 256

Приложение В. Синтаксис инструкций SQL 259
ALTER TABLE 259
COMMIT 260
CREATE INDEX 260
CREATE PROCEDURE 260
CREATE TABLE 261
CREATE VIEW 261
DELETE 261
DROP 262
INSERT 262
INSERT SELECT 262
ROLLBACK 262
SELECT 263
UPDATE 263

Приложение Г. Типы данных в SQL 265
Строковые типы данных 266
Числовые типы данных 268
Типы данных даты и времени 269
Бинарные типы данных 270

Приложение Д. Зарезервированные слова SQL 273

Предметный указатель 277

Об авторе

Бен Форта — директор департамента разработки в компании Adobe Systems. За его плечами более 20 лет работы в компьютерной индустрии, включая разработку продуктов, их поддержку и распространение, а также обучение пользователей. Бен — автор множества бестселлеров, включая книги по MySQL, ColdFusion, Flash, Java, Windows и другим компьютерным технологиям. Имеет большой опыт проектирования баз данных, часто читает лекции и пишет статьи, посвященные технологиям баз данных. Живет в г. Оук-Парк, штат Мичиган, со своей женой Марси и семью детьми. Можете написать Бену письмо по адресу `ben@forta.com` или же посетить его сайт `http://forta.com`.

Введение

SQL является самым популярным языком баз данных. Не важно, кто вы — разработчик приложений, администратор баз данных, веб-дизайнер или пользователь пакета Microsoft Office. Хорошее практическое знание SQL в любом случае поможет вам взаимодействовать с базами данных.

Эта книга была написана по необходимости. Несколько лет я читал курс лекций по разработке веб-приложений, и студенты постоянно просили порекомендовать им книгу по SQL. Существовало много книг, посвященных данной теме, и некоторые из них действительно были очень хороши. Но всем им была присуща одна общая черта: в них было слишком много информации с точки зрения рядовых пользователей. Вместо того чтобы фокусироваться непосредственно на SQL, в большинстве книг излагалось все: от проектирования и нормализации баз данных до теории реляционных баз данных и вопросов администрирования. И хотя это очень важные темы, они не интересны большинству людей, которые просто хотят изучить SQL.

Итак, не найдя ни одной книги, которую я мог бы порекомендовать, я вложил весь опыт преподавания в книгу, которую вы держите в руках. Она поможет вам быстро освоить SQL. Мы начнем с простой выборки данных, постепенно переходя к более сложным темам, таким как объединения, подзапросы, хранимые процедуры, курсоры, триггеры и ограничения. Обучение будет проходить методично, систематично и просто — на каждый урок вам потребуется не более 10 минут.

Книга уже помогла изучить SQL сотням тысяч пользователей, теперь пришел ваш черед. Переходите к первому уроку и приступайте к работе. Вы быстро научитесь писать эффективные SQL-запросы.

Для кого предназначена эта книга

Эта книга предназначена для вас, если вы:

- новичок в SQL;
- хотите быстро научиться использовать SQL;
- хотите научиться применять SQL в разрабатываемых вами приложениях;
- хотите самостоятельно составлять запросы к базам данных на SQL без чьей-либо помощи.

СУБД, рассмотренные в книге

В большинстве случаев SQL-запросы, рассматриваемые в данной книге, можно выполнять в любой системе управления базами данных (СУБД). Но поскольку не все реализации SQL идентичны, особое внимание в книге будет уделено следующим СУБД (по мере необходимости будут даваться детальные инструкции или примечания):

- Apache OpenOffice Base;
- IBM DB2;
- Microsoft Access;
- Microsoft SQL Server (включая Microsoft SQL Server Express);
- MariaDB;
- MySQL;
- Oracle (включая Oracle Express);
- PostgreSQL;
- SQLite.

Для каждой из перечисленных СУБД на сайте книги доступны демонстрационные базы данных (или SQL-сценарии для их создания):

```
http://forta.com/books/0672336073/
http://www.williamspublishing.com/
       Books/978-5-8459-1858-1.html
```

Условные обозначения

В книге используются различные шрифтовые выделения — во-первых, для того чтобы можно было отличить программный код от обычного текста, а во-вторых, чтобы вы не пропустили важные термины.

Текст, который вы вводите, и текст, который должен появиться на экране, будут представлены моноширинным шрифтом.

```
Он выглядит примерно так, как на экране.
```

Переменные и аргументы функций приводятся *моноширинным курсивным* шрифтом. Их необходимо заменять конкретными значениями, в зависимости от логики запроса.

Стрелка (↳) в начале строки кода означает, что эта строка слишком длинная и не поместилась в одну строку книги. Продолжайте вводить все символы после символа ↳ так, как если бы они были частью предыдущей строки.

> ПРИМЕЧАНИЕ
>
> В примечаниях приводится интересная информация, относящаяся к обсуждаемой теме.

> СОВЕТ
>
> В советах даются полезные подсказки или же объясняются более быстрые способы выполнения требуемых действий.

> ПРЕДУПРЕЖДЕНИЕ
>
> В предупреждениях говорится о возможных проблемах и объясняется, как избегать неприятных ситуаций.

> **Новый термин**
>
> В подобных врезках также даются определения новых базовых понятий.

Ввод ▼

Заголовком "Ввод" обозначен код, который можно ввести самостоятельно.

Вывод ▼

Заголовком "Вывод" помечены результаты выполнения запросов.

Анализ ▼

Заголовок "Анализ" предшествует подробному анализу примера.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`
WWW: `http://williamspublishing.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
в Украине: 03150, Киев, а/я 152

УРОК 1

Основы SQL

На этом уроке вы узнаете, что такое SQL и что можно сделать с его помощью.

Терминология баз данных

Тот факт, что вы читаете книгу по SQL, говорит о том, что вам так или иначе необходимо работать с базами данных. SQL предназначен именно для этого, поэтому, прежде чем перейти к рассмотрению языка, очень важно ознакомиться с основными понятиями баз данных.

Хотите вы того или нет, но вы постоянно пользуетесь базами данных. Каждый раз, когда вы выбираете имя в адресной книге электронной почты, вы обращаетесь к базе данных. Когда вы что-то ищете с помощью поискового сайта в Интернете, то посылаете запросы к базе данных. Когда вы регистрируетесь на офисном компьютере, то вводите свое имя и пароль, которые затем сравниваются со значениями, хранящимися в базе данных. И даже когда вы вставляете свою пластиковую карту в банкомат, проверка PIN-кода и остатка на счете идет через базу данных.

Но, несмотря на то что мы постоянно имеем дело с базами данных, для многих остается загадкой, что же это такое. Отчасти это связано с тем, что разные люди вкладывают разный смысл в одни и те же термины. Поэтому мы начнем с определения наиболее важных терминов, относящихся к базам данных.

> СОВЕТ: **вспомните основные концепции**
>
> Ниже даны очень краткие определения основных понятий баз данных. Они предназначены либо для того, чтобы освежить ваши знания, либо чтобы дать вам самые общие представления, если вы новичок. Понимание основ баз данных необходимо для изучения SQL, поэтому рекомендую найти хорошую книгу по базам данных и пополнить свой багаж знаний.

Базы данных

Термин *база данных* используется в самых разных областях, но мы (применительно к SQL) будем считать базу данных набором сведений, хранящихся неким упорядоченным способом. Проще всего рассматривать базу данных как шкаф для хранения документов. Шкаф — это просто физический объект для хранения данных, независимо от того, что это за данные и как они упорядочены.

База данных

Контейнер (обычно файл или группа файлов) для хранения упорядоченных данных.

ПРЕДУПРЕЖДЕНИЕ: **неправильное употребление термина приводит к путанице**

Люди часто используют термин *база данных* для обозначения программного обеспечения, управляющего базой данных. Это ведет к путанице. На самом деле такое программное обеспечение называется *СУБД* (система управления базами данных). База данных — это хранилище, создаваемое и управляемое посредством СУБД. Что именно представляет собой такое хранилище, зависит от конкретной СУБД.

Таблицы

Когда вы храните документы в шкафу, вы стараетесь не перемешивать их. Вместо этого все документы раскладываются по соответствующим папкам.

В среде баз данных такая папка называется таблицей. *Таблица* — это структурированный файл, в котором могут храниться данные определенного типа. В таблице может находиться список клиентов, каталог продукции и любая другая информация.

Таблица

Структурированный набор данных определенного типа.

Ключевой момент заключается в том, что данные, хранимые в таблице, должны быть одного типа или взяты из одного списка. Никогда не храните список клиентов и список заказов в одной таблице базы данных. Это затрудняет поиск и получение информации. Лучше создать две таблицы для каждого из списков.

Каждая таблица базы данных обладает уникальным именем, идентифицирующим ее, и никакая другая таблица в базе данных не может носить то же самое имя.

> ПРИМЕЧАНИЕ: **имена таблиц**
>
> Уникальность имени таблицы достигается комбинацией нескольких компонентов, включая имя базы данных и самой таблицы. В качестве части уникального имени в некоторых базах данных применяется также имя владельца. Это означает, что нельзя использовать два одинаковых имени таблицы в одной базе, но в разных базах данных имена таблиц могут повторяться.

Таблицы имеют определенные характеристики и свойства, определяющие, каким образом в них хранятся данные. Сюда входит информация о том, какие данные могут храниться в таблицах, как они распределены по таблицам, как называются отдельные информационные блоки, и многое другое. Подобный набор информации, описывающей таблицу, называется *схемой*. Схемы служат для описания как отдельных таблиц в базе данных, так и базы данных в целом (а также для описания связей между таблицами, если таковые имеются).

> **Схема**
>
> Информация о базе данных, а также о структуре и свойствах ее таблиц.

Столбцы и типы данных

Таблицы состоят из столбцов, в которых хранятся отдельные фрагменты информации.

Столбец

Одиночное поле таблицы. Все таблицы состоят из одного или нескольких столбцов.

Чтобы лучше понять это, представьте себе таблицу базы данных в виде сетки наподобие электронной таблицы. В каждом столбце сетки находится определенная часть информации. Например, в таблице клиентов в одном столбце находится номер клиента, в другом — его имя. Адрес, город, область, почтовый индекс — все это расположено в отдельных столбцах.

СОВЕТ: **распределение данных**

Очень важно правильно распределить данные по нескольким столбцам. Например, название города, области (штата) и почтовый индекс всегда должны находиться в отдельных столбцах. Это позволяет сортировать или фильтровать данные по столбцам (например, для поиска всех клиентов из определенной области или города). Если названия города и области хранятся в одном столбце, будет очень сложно отсортировать или отфильтровать данные по области.

Когда вы распределяете данные по столбцам, уровень дробления определяется вами и требованиями конкретной базы данных. Например, адреса обычно хранятся в виде названия улицы и номера дома. Это удобно, если только однажды вы не решите отсортировать таблицу по названиям улиц. В таком случае предпочтительно отделять номера домов от названий улиц.

С каждым столбцом базы данных связан определенный тип данных, который указывает, какие данные могут храниться в этом столбце. Например, если в столбце должны содержаться числа (скажем, соответствующие количеству товаров в заказах), то тип данных будет числовым. Если в столбце необходимо хранить даты, текст, заметки, денежные суммы и тому подобное, то для всех этих данных существуют определенные типы.

Тип данных

Тип разрешенных для хранения данных. Каждому столбцу таблицы присваивается тип, который разрешает хранить в нем только определенную информацию.

Типы данных ограничивают характер информации, которую можно хранить в столбце (например, предотвращают ввод алфавитных символов в числовое поле). Типы данных также помогают корректно отсортировать информацию и играют важную роль в оптимизации использования места на диске. Таким образом, выбору типов данных для столбцов создаваемой таблицы необходимо уделить особое внимание.

ПРЕДУПРЕЖДЕНИЕ: **совместимость типов данных**

Типы данных и их названия являются одним из основных источников несовместимости в SQL. Базовые типы данных обычно поддерживаются всеми СУБД примерно одинаково, в отличие от некоторых сложных типов. Более того, иногда вы будете сталкиваться с тем, что один и тот же тип данных в разных СУБД называется по-разному. К сожалению, с этим ничего нельзя поделать, просто следует помнить о таких вещах при создании схем таблиц.

Строки

Данные в таблице хранятся в строках, и каждая запись хранится в своей строке. Возвращаясь к сравнению с сеткой электронной таблицы, можно сказать, что ее вертикальные ряды являются столбцами таблицы, а горизонтальные — строками.

Например, в таблице клиентов информация о каждом клиенте хранится в отдельной строке. Число строк в таблице равно числу записей о клиентах.

Строка

Отдельная запись в таблице.

ПРИМЕЧАНИЕ: **записи или строки?**

Часто пользователи баз данных говорят о *записях*, имея в виду *строки*. Обычно эти два термина взаимозаменяемы, но термин *строка* технически более правилен.

Первичные ключи

В каждой строке таблицы должно быть несколько столбцов, которые уникальным образом идентифицируют ее. В таблице клиентов это могут быть номера клиентов, тогда как в таблице заказов таким столбцом может служить идентификатор заказа. В таблице со списком сотрудников можно использовать столбец с номерами сотрудников или столбец с номерами карточек социального страхования.

Первичный ключ

Столбец (или набор столбцов), значения которого уникально идентифицируют каждую строку таблицы.

Столбец (или набор столбцов), уникально идентифицирующий каждую строку таблицы, называется *первичным ключом*. Первичный ключ нужен для обращения к конкретной строке. Без него выполнять обновление или удаление строк таблицы было бы очень затруднительно, так как не было бы никакой гарантии, что изменяются нужные строки.

СОВЕТ: **всегда определяйте первичные ключи**

Несмотря на то что первичные ключи не являются обязательными, большинство разработчиков баз данных создают их для каждой таблицы, чтобы в будущем иметь возможность выполнять любые манипуляции с данными.

Любой столбец таблицы может быть выбран в качестве первичного ключа, если соблюдаются следующие условия.

- Две разные строки не могут иметь одно и то же значение первичного ключа.

- Каждая строка должна иметь определенное значение первичного ключа (столбцы первичного ключа не могут иметь значения NULL).
- Значения в столбце первичного ключа не могут быть изменены.
- Значения первичного ключа нельзя использовать дважды. (Если строка удалена из таблицы, ее первичный ключ нельзя в дальнейшем назначать другим строкам.)

В качестве первичного ключа обычно выбирается только один столбец таблицы. Но данное требование не обязательно, и первичным ключом могут служить несколько столбцов. При этом правила, приведенные выше, должны соблюдаться для всех столбцов первичного ключа, а все комбинации их значений должны быть уникальными (в отдельных столбцах значения могут повторяться).

Существует еще один важный тип ключа, который называется "внешний ключ", но его мы рассмотрим на уроке 22.

Что такое SQL

SQL (Structured Query Language) — это язык структурированных запросов, который был специально разработан для взаимодействия с базами данных.

В отличие от других языков (разговорных, вроде английского, или языков программирования, таких как Java, C# или PHP), SQL состоит всего из нескольких слов. И сделано это намеренно. SQL был создан для решения одной задачи, с которой он вполне справляется, — предоставлять простой и эффективный способ чтения и записи информации из баз данных.

Каковы же преимущества SQL?

- SQL не относится к числу патентованных языков, используемых поставщиками определенных СУБД. Почти все ведущие СУБД поддерживают SQL, поэтому знание данного языка позволит вам взаимодействовать практически с любой базой данных.
- SQL легко изучить. Его немногочисленные инструкции состоят из простых английских слов.
- Несмотря на кажущуюся простоту, SQL является очень мощным языком. Разумно пользуясь его инструкциями, можно выполнять очень сложные операции с базами данных.

Вот почему стоит изучить SQL.

ПРИМЕЧАНИЕ: **расширения SQL**

Многие поставщики СУБД расширили возможности SQL, введя в язык дополнительные операторы или инструкции. Эти расширения необходимы для обеспечения дополнительной функциональности или для упрощения определенных операций. И хотя часто они бывают очень полезны, подобные расширения специфичны для конкретной СУБД и редко поддерживаются более чем одним поставщиком.

Стандарт SQL контролируется комитетом ANSI и соответственно называется ANSI SQL. Все крупные СУБД, даже те, у которых есть собственные расширения, поддерживают ANSI SQL. Отдельные же реализации носят собственные имена (PL-SQL, Transact-SQL и др.).

Чаще всего в книге рассматривается именно ANSI SQL. В тех редких случаях, когда используется разновидность SQL, относящаяся к определенной СУБД, об этом упоминается отдельно.

Попробуйте сами

Подобно изучению любого другого языка, лучше всего попробовать применить SQL на практике. Для этого вам понадобится база данных и СУБД, в которой можно выполнять SQL-запросы.

Во всех упражнениях книги используются настоящие инструкции SQL и настоящие таблицы базы данных. В приложении А описываются все демонстрационные таблицы и рассказывается о том, как их получить (или создать самостоятельно), чтобы можно было выполнять инструкции каждого урока. В приложении Б объясняется, как выполнять SQL-запросы в различных СУБД. Прежде чем перейти к следующему уроку, прочитайте эти два приложения, чтобы подготовиться к выполнению примеров.

Резюме

На первом уроке вы узнали, что такое SQL и для чего он нужен. В связи с тем что SQL применяется для взаимодействия с базами данных, была также рассмотрена основная терминология баз данных.

УРОК 2

Извлечение данных из таблиц

На этом уроке вы узнаете, как применять инструкцию SELECT для извлечения одного или нескольких столбцов из таблицы.

Инструкция SELECT

Как уже говорилось на уроке 1, инструкции SQL состоят из обычных английских терминов. Эти термины называются *ключевыми словами*, и каждая инструкция SQL содержит одно или несколько ключевых слов. Чаще всего вы будете сталкиваться с инструкцией SELECT, которая предназначена для извлечения информации из одной или нескольких таблиц.

> **Ключевое слово**
>
> Зарезервированное слово, являющееся частью SQL. Никогда не называйте таблицу или столбец таким словом. В приложении Д перечислены наиболее часто используемые ключевые слова.

Чтобы при помощи инструкции SELECT извлечь данные из таблицы, нужно указать как минимум две вещи: что именно вы хотите извлечь и откуда.

ПРИМЕЧАНИЕ: **выполняйте примеры по ходу чтения книги**

Во всех примерах книги используются файлы данных, описанные в приложении А. Если вы хотите самостоятельно выполнять примеры (очень желательно поступать именно так), обратитесь к приложению А, в котором даны инструкции о том, как загрузить эти файлы и создать с их помощью готовые таблицы.

Очень важно понимать, что SQL — это язык, а не приложение. Способ ввода инструкций SQL и отображения результатов их выполнения зависит от конкретного приложения. Чтобы помочь вам приспособить рассматриваемые примеры к своей СУБД, в приложении Б объясняется, как выполнять примеры книги в популярных СУБД и средах разработки.

Извлечение отдельных столбцов

Начнем с простой инструкции SELECT.

Ввод ▼

```
SELECT prod_name
FROM Products;
```

Анализ ▼

В этом примере инструкция SELECT извлекает один столбец под названием prod_name из таблицы Products. Имя столбца указывается сразу после ключевого слова SELECT, а ключевое слово FROM указывает на имя таблицы, из которой извлекаются данные. Результат выполнения инструкции будет таким.

Вывод ▼

```
prod_name
--------------------
Fish bean bag toy
Bird bean bag toy
Rabbit bean bag toy
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
```

```
Raggedy Ann
King doll
Queen doll
```

> ПРИМЕЧАНИЕ: **неотсортированные данные**
>
> Если вы попробуете выполнить этот запрос самостоятельно, то заметите, что данные часто отображаются в ином порядке. Волноваться не стоит — так и должно быть. Если результаты запроса не отсортированы явным образом (об этом будет говориться на следующем уроке), то данные будут возвращаться в произвольном порядке. Это может быть порядок, в котором строки заносились в таблицу, или какой-то другой порядок. Главное, чтобы запрос возвращал одно и то же число строк.

Простая инструкция SELECT, которая использовалась в предыдущем примере, возвращает все строки таблицы. Данные не фильтруются и не сортируются. Эти темы будут рассматриваться на следующих уроках.

> СОВЕТ: **завершение инструкций**
>
> Несколько инструкций SQL должны быть разделены точкой с запятой (символом ;). В большинстве СУБД не требуется вставлять точку с запятой после единственной инструкции, но если в вашем конкретном случае СУБД выдает ошибку, вам придется это делать. Точку с запятой всегда можно добавлять; она не помешает, даже если не является обязательной.

> СОВЕТ: **инструкции SQL и регистр символов**
>
> Важно подчеркнуть, что инструкции SQL нечувствительны к регистру символов, поэтому инструкции SELECT, select и Select эквивалентны. Многие разработчики применяют верхний регистр для всех ключевых слов SQL и нижний регистр — для имен столбцов и таблиц, чтобы код легче читался. Но будьте внимательны: инструкции SQL не зависят от регистра, в отличие от имен таблиц, столбцов и значений (это зависит от СУБД и ее конфигурации).

СОВЕТ: **использование пробелов**

Все лишние пробелы в инструкции SQL пропускаются при обработке запроса. Поэтому инструкция SQL может быть записана как в одной длинной строке, так и разбита на несколько строк. Следующие три инструкции функционально идентичны.

```
SELECT prod_name
FROM Products;

SELECT prod_name FROM Products;

SELECT
prod_name
FROM
Products;
```

Большинство разработчиков разбивают инструкции на несколько строк, чтобы их было легче читать и отлаживать.

Извлечение нескольких столбцов

Для извлечения нескольких столбцов из таблицы применяется та же самая инструкция SELECT. Отличие состоит в том, что после ключевого слова SELECT необходимо через запятую указать несколько имен столбцов.

СОВЕТ: **будьте внимательны с запятыми**

При перечислении нескольких столбцов вставляйте запятые между ними, но не после завершающего столбца в списке. Это приведет к ошибке.

Следующая инструкция SELECT извлекает из таблицы Products три столбца.

Ввод ▼

```
SELECT prod_id, prod_name, prod_price
FROM Products;
```

Анализ ▼

Как и в предыдущем примере, здесь для получения данных из таблицы `Products` применяется инструкция `SELECT`. В данном случае перечислены три имени столбца, разделенные запятыми. Результат выполнения инструкции показан ниже.

Вывод ▼

```
prod_id     prod_name             prod_price
---------   --------------------  -------------
BNBG01      Fish bean bag toy     3.4900
BNBG02      Bird bean bag toy     3.4900
BNBG03      Rabbit bean bag toy   3.4900
BR01        8 inch teddy bear     5.9900
BR02        12 inch teddy bear    8.9900
BR03        18 inch teddy bear    11.9900
RGAN01      Raggedy Ann           4.9900
RYL01       King doll             9.4900
RYL02       Queen doll            9.4900
```

> ПРИМЕЧАНИЕ: **представление данных**
>
> Как видно из результатов запроса, инструкции SQL обычно возвращают неотформатированные данные. Форматирование данных является проблемой представления, а не выборки. Поэтому представление (например, отображение приведенных выше цен в виде денежной суммы с правильным числом знаков после десятичной запятой) зависит от приложения, посредством которого отображаются данные. Неотформатированные данные применяются редко.

Извлечение всех столбцов

Помимо извлечения указанных столбцов (одного или нескольких), с помощью инструкции `SELECT` можно запросить все столбцы, не перечисляя каждый из них. Для этого вместо имен столбцов указывается групповой символ "звездочка" (`*`). Это делается следующим образом.

Ввод ▼

```
SELECT *
FROM Products;
```

Анализ ▼

При наличии группового символа (*) возвращаются все столбцы. Обычно (но не всегда) столбцы возвращаются в том порядке, в котором они указаны в определении таблицы. Впрочем, табличные данные редко выводятся в том виде, в котором они хранятся в базе данных. (Обычно они возвращаются в приложение, которое должным образом форматирует их.)

> ПРЕДУПРЕЖДЕНИЕ: **использование групповых символов**
>
> Лучше не использовать групповой символ * (кроме тех случаев, когда вам действительно необходимы все столбцы таблицы). Несмотря на то что групповые символы помогут сэкономить время и усилия, необходимые для перечисления всех необходимых столбцов, извлечение ненужных столбцов обычно приводит к снижению производительности запроса и приложения в целом.

> СОВЕТ: **извлечение неизвестных столбцов**
>
> Есть одно большое преимущество в использовании групповых символов. Поскольку вы не указываете точные имена столбцов (при наличии символа * возвращаются все столбцы), появляется возможность извлечь столбцы, имена которых неизвестны.

Извлечение уникальных строк

Как вы убедились, инструкция SELECT возвращает все строки, соответствующие критерию отбора. Но что если вам не нужны абсолютно все значения? Предположим, например, что необходимо узнать идентификаторы всех поставщиков из таблицы Products.

Ввод ▼

```
SELECT vend_id
FROM Products;
```

Вывод ▼

```
vend_id
----------
BRS01
BRS01
BRS01
DLL01
DLL01
DLL01
DLL01
FNG01
FNG01
```

Инструкция SELECT вернула 9 строк (хотя в списке всего четыре поставщика), потому что в таблице Products указаны 9 товаров. Как же получить список уникальных значений?

Решение заключается в применении ключевого слова DISTINCT, которое, как нетрудно предположить, заставляет СУБД вернуть только уникальные значения.

Ввод ▼

```
SELECT DISTINCT vend_id
FROM Products;
```

Анализ ▼

Предложение SELECT DISTINCT vend_id заставляет СУБД вернуть только записи с отличающимися значениями vend_id, и в результате мы получаем три строки, как показано ниже. Ключевое слово DISTINCT, если оно имеется, должно находиться непосредственно перед списком имен столбцов.

Вывод ▼

```
vend_id
----------
BRS01
DLL01
FNG01
```

ПРЕДУПРЕЖДЕНИЕ: **строки не могут быть частично уникальными**

Ключевое слово DISTINCT применяется ко всем столбцам, а не только к тому, перед которым оно стоит. Если задать выражение SELECT DISTINCT vend_id, prod_price, будут извлечены все строки, если только не окажется, что один и тот же продавец предлагает разные товары по одинаковым ценам.

Ограничение результатов запроса

Инструкция SELECT возвращает все строки, соответствующие критерию отбора. Зачастую это все строки таблицы. Но что если необходимо получить лишь первую строку или заданное число строк? Такое вполне возможно, однако приходится с сожалением констатировать, что это одна из тех редких ситуаций, когда разные реализации SQL ведут себя по-разному.

В Microsoft SQL Server и Microsoft Access можно воспользоваться ключевым словом TOP, чтобы извлечь лишь несколько первых записей, как показано ниже.

Ввод ▼

```
SELECT TOP 5 prod_name
FROM Products;
```

Вывод ▼

```
prod_name
----------------
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Fish bean bag toy
Bird bean bag toy
```

Анализ ▼

В этой инструкции предложение SELECT TOP 5 задает извлечение первых пяти строк.

В DB2 применяется собственный уникальный синтаксис.

Ввод ▼

```
SELECT prod_name
FROM Products
FETCH FIRST 5 ROWS ONLY;
```

Анализ ▼

Предложение FETCH FIRST 5 ROWS ONLY навряд ли требует каких-либо пояснений.

В Oracle необходимо организовать подсчет строк с помощью встроенного счетчика ROWNUM.

Ввод ▼

```
SELECT prod_name
FROM Products
WHERE ROWNUM <=5;
```

В MySQL, MariaDB, PostgreSQL и SQLite можно воспользоваться предложением LIMIT.

Ввод ▼

```
SELECT prod_name
FROM Products
LIMIT 5;
```

Анализ ▼

В этой инструкции извлекается единственный столбец. Предложение LIMIT 5 заставляет СУБД вернуть не более пяти строк. Если необходимо получить следующие пять строк, задайте начальную точку извлечения и требуемое количество строк, как показано ниже.

Ввод ▼

```
SELECT prod_name
FROM Products
LIMIT 5 OFFSET 5;
```

Анализ ▼

Предложение `LIMIT 5 OFFSET 5` заставляет СУБД вернуть пять строк, начиная со строки 5. Первое число — это количество строк для извлечения, а второе — начальная точка. Результат показан ниже.

Вывод ▼

```
prod_name
------------------
Rabbit bean bag toy
Raggedy Ann
King doll
Queen doll
```

> ПРЕДУПРЕЖДЕНИЕ: **строка 0**
>
> Первая извлекаемая строка имеет номер `0`, а не `1`. Таким образом, инструкция с предложением `LIMIT 1 OFFSET 1` вернет вторую строку, а не первую.

> СОВЕТ: **краткая форма записи для MySQL, MariaDB и SQLite**
>
> MySQL, MariaDB и SQLite поддерживают краткую форму предложения `LIMIT 3 OFFSET 4`, позволяя записать его как `LIMIT 3, 4`. В данной форме число перед запятой задает количество строк, а после запятой — начальную точку.

> ПРИМЕЧАНИЕ: **не все реализации SQL одинаковы**
>
> Данный раздел с описанием инструкций, позволяющих ограничивать результаты запроса, включен лишь для того, чтобы показать: нельзя слепо надеяться на то, что синтаксис инструкций SQL будет одинаковым и согласованным во всех СУБД. Простейшие инструкции обычно являются переносимыми, но более сложные всегда нужно проверять. Помните об этом при разработке приложений для баз данных.

Итак, предложение LIMIT задает число возвращаемых строк. Предложение LIMIT с ключевым словом OFFSET указывает, с какой строки таблицы необходимо начать отсчет. В нашем примере таблица Products содержит всего девять товаров, поэтому инструкция с предложением LIMIT 5 OFFSET 5 вернула четыре строки (пятой просто нет).

Использование комментариев

Инструкции SQL обрабатываются самой СУБД. Но что если в инструкцию необходимо включить текст, который не должен ни выполняться, ни обрабатываться? И зачем это вообще нужно? Вот несколько причин.

- Рассмотренные ранее инструкции были очень короткими и простыми. Но по мере увеличения сложности запросов в них придется добавлять описательные комментарии (в качестве напоминания самому себе на будущее или для других членов проекта). Такие комментарии вставляются в тексты запросов, но совершенно очевидно, что они не предназначены для обработки со стороны СУБД. В качестве примеров изучите файлы create.txt и populate.txt, использованные в приложении Б.
- То же самое справедливо для заголовков в начале SQL-файла, содержащих контактные данные программиста, а также различные описания и примечания. Примеры этого также можно увидеть в файлах, использованных в приложении Б.
- Другое важное применение комментариев — временный запрет выполнения фрагментов SQL-кода. Если вы создаете длинный запрос и хотите протестировать лишь его фрагмент, *закомментируйте* лишние фрагменты, чтобы СУБД проигнорировала их.

Большинство СУБД поддерживает несколько вариантов синтаксиса комментариев. Начнем с комментариев, встроенных в строку кода.

Ввод ▼

```
SELECT prod_name      -- это комментарий
FROM Products;
```

Анализ ▼

Для встраивания комментариев предназначено выражение -- (два дефиса). Все, что идет далее, считается текстом комментария.

Подобным образом удобно, к примеру, описывать назначение столбцов в инструкции CREATE TABLE.

Вот еще одна разновидность встроенных комментариев (она поддерживается реже).

Ввод ▼

```
# Это комментарий
SELECT prod_name
FROM Products;
```

Анализ ▼

Символ # в начале строки превращает ее в комментарий.

Можно также создавать многострочные комментарии, в том числе такие, которые предотвращают выполнение фрагментов запроса.

Ввод ▼

```
/* SELECT prod_name, vend_id
FROM Products; */
SELECT prod_name
FROM Products;
```

Анализ ▼

Выражение /* помечает начало комментария, а выражение */ — его конец. Все, что находится между ними, становится комментарием. Подобный тип комментариев часто применяется для блокирования фрагментов кода. В данном примере определены две инструкции SELECT, но первая из них не будет выполняться, потому что закомментирована.

Резюме

На этом уроке вы узнали, как применять инструкцию SELECT для извлечения одного, нескольких и всех столбцов таблицы. Было также показано, как извлекать уникальные значения и вставлять комментарии в код. К сожалению, вы убедились в том, что сложные SQL-запросы приходится проверять на совместимость с различными СУБД. Далее мы рассмотрим, как сортировать результаты запроса.

УРОК 3

Сортировка полученных данных

На этом уроке вы узнаете, как использовать предложение ORDER BY инструкции SELECT для сортировки результатов запроса.

Сортировка записей

Как объяснялось на предыдущем уроке, следующая инструкция SQL возвращает один столбец из таблицы. Но взгляните на результаты: данные выводятся в произвольном порядке.

Ввод ▼

```
SELECT prod_name
FROM Products;
```

Вывод ▼

```
prod_name
--------------------
Fish bean bag toy
Bird bean bag toy
Rabbit bean bag toy
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Raggedy Ann
King doll
Queen doll
```

Вообще-то, строки отображаются не в случайном порядке. При отсутствии сортировки строки обычно выводятся в том порядке, в котором они хранятся в таблице. Скорее всего, именно в этой последовательности они изначально добавлялись в таблицу. Но если

данные впоследствии обновлялись или удалялись, то порядок будет зависеть от того, как СУБД использует освободившееся место. Таким образом, вы не можете (и не должны) полагаться на порядок сортировки, если не задаете его в явном виде. В теории реляционных баз данных говорится, что порядок извлечения данных не имеет значения, если не указан порядок сортировки.

> **Предложение**
>
> Инструкции SQL состоят из предложений, одни из которых обязательны, другие — нет. Предложение обычно включает в себя ключевое слово и пользовательские данные. Примером может служить предложение FROM инструкции SELECT из предыдущего листинга.

Для сортировки извлекаемых инструкцией SELECT данных предназначено предложение ORDER BY. В нем указывается имя одного или нескольких столбцов, по которым и сортируются результаты запроса. Рассмотрим следующий пример.

Ввод ▼

```
SELECT prod_name
FROM Products
ORDER BY prod_name;
```

Анализ ▼

Эта инструкция идентична предыдущей, за исключением предложения ORDER BY, которое заставляет СУБД отсортировать данные в алфавитном порядке по столбцу prod_name. Результат показан ниже.

Вывод ▼

```
prod_name
--------------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
Bird bean bag toy
```

```
Fish bean bag toy
King doll
Queen doll
Rabbit bean bag toy
Raggedy Ann
```

> ПРЕДУПРЕЖДЕНИЕ: **местоположение предложения ORDER BY**
>
> При использовании предложения ORDER BY убедитесь в том, что оно является последним в инструкции SELECT. В противном случае будет выдано сообщение об ошибке.

> СОВЕТ: **сортировка по невыбранным столбцам**
>
> Чаще всего столбцы, указываемые в предложении ORDER BY, отображаются на экране, хотя это не обязательно. Данные могут сортироваться и по столбцу, который не извлекается в самом запросе.

Сортировка по нескольким столбцам

Часто бывает необходимо отсортировать данные по нескольким столбцам. Например, если вы выводите список сотрудников, вам может понадобиться отсортировать его сначала по фамилии сотрудника, а затем по имени. Это может быть полезным, если в компании есть несколько человек с одинаковыми фамилиями.

Чтобы выполнить сортировку по нескольким столбцам, укажите их имена через запятую. В следующем примере извлекаются три столбца, а результат сортируется по двум из них: сначала — по цене, потом — по названию.

Ввод ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price, prod_name;
```

Вывод ▼

```
prod_id     prod_price     prod_name
---------   ------------   ----------------
BNBG02      3.4900         Bird bean bag toy
BNBG01      3.4900         Fish bean bag toy
BNBG03      3.4900         Rabbit bean bag toy
RGAN01      4.9900         Raggedy Ann
BR01        5.9900         8 inch teddy bear
BR02        8.9900         12 inch teddy bear
RYL01       9.4900         King doll
RYL02       9.4900         Queen doll
BR03        11.9900        18 inch teddy bear
```

Важно понимать, что при сортировке по нескольким столбцам порядок сортировки будет таким, который указан в запросе. Другими словами, в приведенном выше примере товары сортируются по столбцу `prod_name`, только если существует несколько строк с одинаковыми значениями `prod_price`. Если никакие значения столбца `prod_price` не совпадают, данные по столбцу `prod_name` сортироваться не будут.

Сортировка по положению столбца

Порядок сортировки можно указать не только по именам столбцов, но и по относительному положению столбца (проще говоря — по номеру столбца). Чтобы лучше понять это, рассмотрим пример.

Ввод ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY 2, 3;
```

Вывод ▼

```
prod_id     prod_price     prod_name
--------    ------------   -------------------
BNBG02      3.4900         Bird bean bag toy
BNBG01      3.4900         Fish bean bag toy
BNBG03      3.4900         Rabbit bean bag toy
RGAN01      4.9900         Raggedy Ann
```

```
BR01        5.9900      8 inch teddy bear
BR02        8.9900      12 inch teddy bear
RYL01       9.4900      King doll
RYL02       9.4900      Queen doll
BR03        11.9900     18 inch teddy bear
```

Анализ ▼

Как видите, результат выполнения запроса идентичен предыдущему примеру. Разница только в предложении ORDER BY. Здесь не указаны имена столбцов, а лишь оговорено их относительное положение в списке SELECT. Выражение ORDER BY 2 означает сортировку по второму столбцу списка, а именно по столбцу prod_price. Предложение ORDER BY 2, 3 означает сортировку по столбцу prod_price, а затем — по столбцу prod_name.

Основное преимущество данного метода заключается в том, что не нужно несколько раз набирать в запросе имена столбцов. Но имеются и недостатки. Во-первых, отсутствие явных имен столбцов повышает вероятность того, что вы случайно укажете не тот столбец. Во-вторых, можно случайно сменить порядок столбцов при изменении списка SELECT (забыв при этом внести соответствующие изменения в предложение ORDER BY). И наконец, очевидно, что нельзя использовать этот метод для сортировки по столбцам, не указанным в списке SELECT.

> СОВЕТ: **сортировка по невыбранным столбцам**
>
> Описанный выше метод нельзя применять при сортировке по столбцам, отсутствующим в списке SELECT. Однако в случае необходимости можно в одной инструкции указывать реальные имена столбцов и их номера.

Указание направления сортировки

Данные можно сортировать не только по возрастанию (от А до Я). Такой порядок задан по умолчанию, но в предложении ORDER BY можно также указывать порядок по убыванию (от Я до А). Для этого предназначено ключевое слово DESC.

В следующем примере товары сортируются в порядке убывания цен (вначале идут самые дорогие товары).

Ввод ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price DESC;
```

Вывод ▼

```
prod_id     prod_price     prod_name
---------   ------------   --------------------
BR03        11.9900        18 inch teddy bear
RYL01       9.4900         King doll
RYL02       9.4900         Queen doll
BR02        8.9900         12 inch teddy bear
BR01        5.9900         8 inch teddy bear
RGAN01      4.9900         Raggedy Ann
BNBG01      3.4900         Fish bean bag toy
BNBG02      3.4900         Bird bean bag toy
BNBG03      3.4900         Rabbit bean bag toy
```

Но как быть в случае сортировки по нескольким столбцам? В следующем примере товары сортируются в порядке убывания цен (вначале самые дорогие), а также по названиям.

Ввод ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price DESC, prod_name;
```

Вывод ▼

```
prod_id     prod_price     prod_name
---------   -------        ------------------
BR03        11.9900        18 inch teddy bear
RYL01       9.4900         King doll
RYL02       9.4900         Queen doll
BR02        8.9900         12 inch teddy bear
BR01        5.9900         8 inch teddy bear
RGAN01      4.9900         Raggedy Ann
BNBG02      3.4900         Bird bean bag toy
BNBG01      3.4900         Fish bean bag toy
BNBG03      3.4900         Rabbit bean bag toy
```

Анализ ▼

Ключевое слово DESC применяется только к тому столбцу, после которого оно стоит. В предыдущем примере ключевое слово DESC было указано для столбца prod_price, но не для prod_name. Таким образом, столбец prod_price сортируется по убыванию, а столбец prod_name (в пределах каждого ценового блока) — как обычно, по возрастанию.

> СОВЕТ: **сортировка по убыванию по нескольким столбцам**
>
> Если необходимо отсортировать данные в порядке убывания по нескольким столбцам, укажите для каждого из них ключевое слово DESC.

Следует уточнить, что DESC — это сокращение от DESCENDING; можно использовать оба ключевых слова. Антонимом для DESC является ключевое слово ASC (ASCENDING), которое можно указывать для сортировки по возрастанию. Однако на практике слово ASC обычно не применяется, поскольку такой порядок используется по умолчанию (он предполагается, если не указано ни ASC, ни DESC).

> СОВЕТ: **чувствительность к регистру символов и порядок сортировки**
>
> При сортировке текстовых данных A — это то же самое, что и a? И a идет перед Б или после Я? Это не теоретические вопросы, и ответ на них зависит от настроек базы данных.
>
> При словарном порядке сортировки A считается идентичным a, и такое поведение является обычным для большинства СУБД. Однако в некоторых СУБД администратор может в случае необходимости изменить данную настройку. (Это может оказаться полезным, если в базе данных содержится много текстовых записей на иностранном языке.)
>
> Суть в том, что если вам понадобится альтернативный порядок сортировки, его нельзя будет достичь посредством обычного предложения ORDER BY. Вам придется обратиться к администратору базы данных.

Резюме

Этот урок был посвящен сортировке результатов запроса с помощью предложения ORDER BY инструкции SELECT. Данное предложение, которое должно быть последним в инструкции SELECT, можно применять для сортировки строк по одному или нескольким столбцам.

УРОК 4

Фильтрация данных

На этом уроке вы узнаете, как использовать предложение WHERE инструкции SELECT для задания условий фильтрации строк.

Использование предложения WHERE

В таблицах баз данных обычно содержится очень много информации, и довольно редко возникает необходимость извлекать все строки из таблицы. Гораздо чаще требуется извлечь какую-то часть данных для последующей обработки или составления отчетов. В этом случае необходимо указать *критерий отбора*, также называемый *условием фильтрации*.

В инструкции SELECT данные фильтруются путем указания критерия отбора в предложении WHERE. Это предложение указывается сразу после названия таблицы (в предложении FROM) следующим образом.

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price = 3.49;
```

Анализ ▼

Приведенная инструкция извлекает два столбца из таблицы товаров, но возвращает не все строки, а только те, значение в столбце `prod_price` которых равно `3.49`.

Вывод ▼

```
prod_name              prod_price
---------------        ------------
Fish bean bag toy      3.49
Bird bean bag toy      3.49
Rabbit bean bag toy    3.49
```

В данном примере используется простая проверка на равенство: сначала проверяется, содержится ли в столбце указанное значение, а затем данные фильтруются соответствующим образом. Однако SQL позволяет выполнять не только проверку на равенство.

> СОВЕТ: **сколько нулей?**
>
> Выполняя примеры данного урока, вы обнаружите, что числовые результаты отображаются в разных форматах: `3.49`, `3.490`, `3.4900` и т.п. Подобное поведение зависит от конкретной СУБД и от того, какие форматы данных в ней применяются. Так что, если ваши результаты чуть отличаются от приведенных в книге, не переживайте. В конце концов, `3.49` и `3.4900` — это одно и то же.

> СОВЕТ: **фильтрация в SQL и в приложении**
>
> Данные могут также фильтроваться на уровне приложения. Для этого инструкция `SELECT` первоначально извлекает больше данных, чем необходимо для клиентского приложения, а затем клиентский код обрабатывает полученные данные для отбора только нужных строк.
>
> Как правило, такой метод не приветствуется. Базы данных оптимизированы для быстрой и эффективной фильтрации. Заставляя клиентское приложение выполнять функции базы данных, вы значительно ухудшаете его производительность, а также затрудняете его корректное масштабирование. Кроме того, если данные фильтруются на стороне клиента, сервер отправляет ненужные данные по сети, тем самым увеличивая сетевой трафик.

> ПРЕДУПРЕЖДЕНИЕ: **положение предложения `WHERE`**
>
> При использовании обоих предложений, `ORDER BY` и `WHERE`, убедитесь, что предложение `ORDER BY` следует за предложением `WHERE`, иначе возникнет ошибка (см. урок 3).

Операторы в предложении WHERE

В первом предложении WHERE, которое мы рассмотрели, выполнялась проверка на равенство, т.е. определялось, содержится ли в столбце указанное значение. В SQL поддерживается целый набор условных (логических) операторов, которые перечислены в табл. 4.1.

ТАБЛИЦА 4.1. **Операторы в предложении WHERE**

Оператор	Проверка
=	Равенство
<>	Неравенство
!=	Неравенство
<	Меньше
<=	Меньше или равно
!<	Не меньше
>	Больше
>=	Больше или равно
!>	Не больше
BETWEEN	Вхождение в диапазон
IS NULL	Значение NULL

ПРЕДУПРЕЖДЕНИЕ: **совместимость операторов**

Некоторые из операторов, приведенных в табл. 4.1, избыточны. Например, <> — это то же самое, что и !=. Применение оператора !< (не меньше чем) дает тот же результат, что и >= (больше или равно). Однако учтите: не все из этих операторов поддерживаются всеми СУБД. Обратитесь к документации своей СУБД, чтобы узнать, какие логические операторы она поддерживает.

Сравнение с одиночным значением

Мы уже рассмотрели пример проверки на равенство. Теперь познакомимся с другими операторами.

В первом примере выводятся названия товаров, стоимость которых не превышает 10 долларов.

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price < 10;
```

Вывод ▼

```
prod_name              prod_price
----------------       -------------
Fish bean bag toy      3.49
Bird bean bag toy      3.49
Rabbit bean bag toy    3.49
8 inch teddy bear      5.99
12 inch teddy bear     8.99
Raggedy Ann            4.99
King doll              9.49
Queen doll             9.49
```

В следующей инструкции извлекаются все товары, которые стоят 10 долларов или меньше (результат будет таким же, как и в предыдущем примере, так как в базе данных нет товаров, которые стоили бы ровно 10 долларов).

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price <= 10;
```

Проверка на неравенство

В следующем примере выводятся все товары, не изготовленные фирмой DLL01.

Ввод ▼

```
SELECT vend_id, prod_price
FROM Products
WHERE vend_id <> 'DLL01';
```

Вывод ▼

```
vend_id       prod_name
---------     -------------
BRS01         8 inch teddy bear
BRS01         12 inch teddy bear
BRS01         18 inch teddy bear
FNG01         King doll
FNG01         Queen doll
```

> СОВЕТ: **когда использовать кавычки**
>
> Если вы внимательно изучите выражения в предыдущих предложениях WHERE, то заметите, что некоторые значения заключены в одинарные кавычки, а некоторые — нет. Одинарные кавычки служат для определения границ строки. При сравнении значения со столбцом, содержащим строковые данные, необходимо заключать строку в кавычки. При использовании числовых столбцов кавычки не нужны.

Ниже приведен тот же пример, только здесь уже используется оператор != вместо <>.

Ввод ▼

```
SELECT vend_id, prod_price
FROM Products
WHERE vend_id != 'DLL01';
```

> ПРЕДУПРЕЖДЕНИЕ: **оператор != или <>**
>
> Операторы != и <> обычно взаимозаменяемы. Однако не во всех СУБД поддерживаются обе формы оператора. Например, в Microsoft Access поддерживается оператор <> и не поддерживается !=. Если у вас возникли сомнения по поводу своей СУБД, обратитесь к документации.

Сравнение с диапазоном значений

Для сравнения с диапазоном значений можно использовать оператор BETWEEN. Его синтаксис немного отличается от других операторов в предложении WHERE, так как для него требуются два значения:

начальное и конечное. Например, данный оператор можно использовать для поиска товаров, цена которых находится в промежутке между 5 и 10 долларами, или всех дат, которые попадают в диапазон между указанными начальной и конечной датами.

В следующем примере демонстрируется применение оператора BETWEEN для извлечения всех товаров, цена которых выше 5 и ниже 10 долларов.

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price BETWEEN 5 AND 10;
```

Вывод ▼

```
prod_name              prod_price
----------------       -------------
8 inch teddy bear      5.99
12 inch teddy bear     8.99
King doll              9.49
Queen doll             9.49
```

Анализ ▼

Как видно из приведенного примера, при использовании оператора BETWEEN нужно указывать два значения: нижнюю и верхнюю границы диапазона. Оба значения должны быть разделены ключевым словом AND. При этом извлекаются все значения из диапазона, включая те, что равны граничным значениям.

Проверка на отсутствие значения

После создания таблицы разработчик может указать, допустимо ли, чтобы в отдельных ее столбцах не содержались никакие значения. Когда в столбце не содержится никакого значения, это значит, что в нем содержится значение NULL.

> **NULL**
>
> Отсутствие какого-либо значения, в отличие от поля, содержащего 0, пустую строку или просто несколько пробелов.

Чтобы проверить, содержит ли столбец значение NULL, нельзя просто записать = NULL. Для предложения WHERE предусмотрено специальное выражение, которое используется для проверки значений NULL в столбцах: IS NULL. Синтаксис выглядит следующим образом.

Ввод ▼

```
SELECT prod_name
FROM Products
WHERE prod_price IS NULL;
```

Эта инструкция возвращает список товаров без цены (поле prod_price пустое, а не содержит цену 0), а поскольку таковых нет, никаких данных мы не получим. Однако в таблице Customers есть столбцы со значениями NULL — в столбце cust_email будет содержаться NULL, если не указан адрес электронной почты.

Ввод ▼

```
SELECT cust_name
FROM Customers
WHERE cust_email IS NULL;
```

Вывод ▼

```
cust_name
--------------
Kids Place
Tha Toy Store
```

> СОВЕТ: **операторы, специфичные для конкретной СУБД**
>
> Во многих СУБД набор операторов расширен дополнительными фильтрами. Обратитесь к документации своей СУБД за дополнительной информацией.

> СОВЕТ: **значения NULL и проверки на неравенство**
>
> Многие пользователи ожидают, что при извлечении строк, не содержащих заданного значения, будут также возвращаться строки, содержащие значение NULL. Однако это не так. Значение NULL трактуется как неопределенное, и СУБД не может выполнить проверку такого значения ни на равенство, ни на неравенство.

Резюме

На этом уроке рассказывалось о том, как фильтровать возвращаемые данные с помощью предложения WHERE инструкции SELECT. Вы узнали, как проверить данные на равенство, неравенство, вхождение в диапазон значений, а также на отсутствие значения (NULL).

УРОК 5

Расширенная фильтрация данных

На этом уроке вы узнаете, как комбинировать предложения WHERE для создания сложных условий отбора. Будет также продемонстрировано применение операторов NOT и IN.

Комбинирование условий WHERE

Все предложения WHERE, представленные на уроке 4, фильтруют данные с использованием одного критерия. Чтобы создать более сложные фильтры, можно использовать несколько условий WHERE. Для этого предназначены операторы AND и OR.

> **Оператор**
>
> Специальное ключевое слово, используемое для объединения или изменения условий в предложении WHERE.

Оператор AND

Чтобы отфильтровать данные по нескольким столбцам, необходимо воспользоваться оператором AND для добавления условий в предложение WHERE. Вот как это делается.

Ввод ▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
WHERE vend_id = 'DLL01' AND prod_price <=4;
```

Анализ ▼

Посредством данного запроса извлекаются названия и цены всех товаров, предлагаемых поставщиком DLL01, с ценой 4 доллара и меньше. Предложение WHERE в инструкции SELECT содержит два условия, а оператор AND используется для их объединения. Оператор AND заставляет СУБД возвращать только те строки, которые удовлетворяют всем перечисленным условиям. Если товар предлагается поставщиком DLL01, но стоит больше 4 долларов, он не попадет в результаты запроса. Аналогично товары, которые стоят меньше 4 долларов и предлагаются иными поставщиками, также не будут выведены. Результаты запроса будут такими.

Вывод ▼

```
prod_id   prod_price      prod_name
-------   -------------   ----------------
BNBG02    3.4900          Bird bean bag toy
BNBG01    3.4900          Fish bean bag toy
BNBG03    3.4900          Rabbit bean bag toy
```

AND

Ключевое слово, используемое в предложении WHERE для того, чтобы возвращались только те строки, которые удовлетворяют всем указанным условиям.

В приведенном примере использовались два условия фильтрации, объединенные оператором AND. Можно задавать и большее число условий. Каждое из них должно отделяться оператором AND.

ПРИМЕЧАНИЕ: **отсутствие предложения** ORDER BY

Для экономии места в большинстве примеров не указывается предложение ORDER BY. Поэтому вполне возможно, что полученные вами результаты будут немного отличаться от тех, что приведены в книге. Количество возвращаемых строк всегда будет одинаковым, а вот их порядок может оказаться иным. Можете добавлять предложение ORDER BY по своему усмотрению. Главное, чтобы оно стояло после предложения WHERE.

Оператор OR

Действие оператора OR противоположно действию оператора AND. Он заставляет СУБД извлекать только те строки, которые удовлетворяют хотя бы одному условию. В большинстве СУБД второе условие даже не рассматривается в предложении WHERE, если выполняется первое условие (в таком случае строка будет выведена независимо от второго условия).

Рассмотрим следующую инструкцию SELECT.

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01';
```

Анализ ▼

Посредством этого запроса извлекаются названия и цены всех товаров, изготовленных одним из указанных поставщиков. Оператор OR заставляет СУБД применять какое-то одно условие, а не оба сразу. Если бы здесь использовался оператор AND, мы не получили бы никаких данных. Результаты запроса будут такими.

Вывод ▼

```
prod_name           prod_price
------------------- ------------
Fish bean bag toy   3.4900
Bird bean bag toy   3.4900
Rabbit bean bag toy 3.4900
8 inch teddy bear   5.9900
12 inch teddy bear  8.9900
18 inch teddy bear  11.9900
Raggedy Ann         4.9900
```

OR

Ключевое слово, применяемое в предложении WHERE для того, чтобы возвращались все строки, удовлетворяющие любому из указанных условий.

Порядок обработки операторов

Предложения WHERE могут содержать любое количество логических операторов AND и OR. Комбинируя их, можно создавать сложные фильтры.

Однако при комбинировании операторов AND и OR возникает одна проблема. Рассмотрим пример. Необходимо вывести список всех предлагаемых поставщиками DLL01 и BRS01 товаров, цена которых — 10 долларов и выше. В следующей инструкции SELECT используется комбинация операторов AND и OR для формирования условия отбора строк.

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01'
      AND prod_price >= 10;
```

Вывод ▼

```
prod_name              prod_price
---------------------  -------------
Fish bean bag toy      3.4900
Bird bean bag toy      3.4900
Rabbit bean bag toy    3.4900
18 inch teddy bear     11.9900
Raggedy Ann            4.9900
```

Анализ ▼

Взгляните на результат. В четырех возвращаемых строках значатся цены ниже 10 долларов — очевидно, строки не были отфильтрованы так, как надо. Что же произошло? Причина заключается в порядке обработки операторов. SQL (как и большинство других языков) вначале обрабатывает логические операторы AND, а потом уже — логические операторы OR. Когда СУБД встречает такое предложение WHERE, она интерпретирует его так: *извлечь все товары, которые стоят 10 долларов и больше и при этом предлагаются поставщиком BRS01, а также все товары, предлагаемые поставщиком DLL01,*

независимо от их цены. Другими словами, в связи с тем, что приоритет у логического оператора AND выше, были объединены не те операторы.

Решение этой проблемы состоит в использовании скобок для точной группировки необходимых логических операторов. Рассмотрим следующую инструкцию SELECT и ее результаты.

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE (vend_id = 'DLL01' OR vend_id = 'BRS01')
      AND prod_price >= 10;
```

Вывод ▼

```
prod_name              prod_price
--------------------   -------------
18 inch teddy bear     11.9900
```

Анализ ▼

Единственным различием между предыдущим запросом и этим являются скобки, в которые заключены первые два условия предложения WHERE. Поскольку скобки имеют еще больший приоритет, чем логические операторы AND и OR, СУБД вначале обрабатывает условие OR в скобках. Соответственно, запрос будет пониматься так: *извлечь все товары, предлагаемые либо поставщиком DLL01, либо поставщиком BRS01, которые стоят 10 долларов и больше*. Это именно то, что нужно.

> СОВЕТ: **использование скобок в предложении WHERE**
>
> Используя логические операторы AND и OR в предложении WHERE, всегда вставляйте скобки, чтобы точно сгруппировать условия. Не полагайтесь на порядок обработки по умолчанию, даже если он подразумевает необходимый вам результат. Тем самым вы всегда будете застрахованы от неожиданностей.

Оператор IN

Оператор IN служит для указания диапазона условий, любое из которых может быть выполнено. При этом значения, заключенные в скобки, перечисляются через запятую. Рассмотрим следующий пример.

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id IN ('DLL01', 'BRS01')
ORDER BY prod_name
```

Вывод ▼

```
prod_name              prod_price
---------------------  -------------
12 inch teddy bear     8.9900
18 inch teddy bear     11.9900
8 inch teddy bear      5.9900
Bird bean bag toy      3.4900
Fish bean bag toy      3.4900
Rabbit bean bag toy    3.4900
Raggedy Ann            4.9900
```

Анализ ▼

Инструкция SELECT извлекает все товары, предлагаемые поставщиками DLL01 и BRS01. После оператора IN указан список значений через запятую, а весь список заключен в скобки.

Если вы думаете, что оператор IN выполняет ту же функцию, что и OR, то будете совершенно правы. Следующий запрос равносилен предыдущему.

Ввод ▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01'
ORDER BY prod_name;
```

Вывод ▼

```
prod_name           prod_price
------------------- ------------
12 inch teddy bear  8.9900
18 inch teddy bear  11.9900
8 inch teddy bear   5.9900
Bird bean bag toy   3.4900
Fish bean bag toy   3.4900
Rabbit bean bag toy 3.4900
Raggedy Ann         4.9900
```

Зачем же нужен оператор IN? Его преимущества таковы.

- При работе с длинными списками необходимых значений синтаксис логического оператора IN гораздо понятнее.
- При использовании оператора IN совместно с операторами AND и OR гораздо легче управлять порядком обработки.
- Операторы IN почти всегда быстрее обрабатываются, чем списки операторов OR (впрочем, это сложно заметить в случае коротких списков).
- Самое большое преимущество логического оператора IN заключается в том, что в нем может содержаться еще одна инструкция SELECT, а это позволяет создавать очень гибкие предложения WHERE. Подробнее об этом рассказывается на уроке 11.

> **IN**
>
> Ключевое слово, используемое в предложении WHERE для указания списка значений, обрабатываемых так же, как это делается в случае применения оператора OR.

Оператор NOT

Логический оператор NOT в предложении WHERE служит только одной цели — отрицать все условия, следующие за ним. Поскольку он никогда не используется сам по себе (а только вместе с другими логическими операторами), его синтаксис немного отличается. Оператор NOT вставляется перед названием столбца, значения которого нужно отфильтровать, а не после.

NOT

Ключевое слово, применяемое в предложении WHERE для отрицания какого-либо условия.

В следующем примере демонстрируется применение логического оператора NOT. Здесь извлекается список товаров, предлагаемых всеми поставщиками, кроме DLL01.

Ввод ▼

```
SELECT prod_name
FROM Products
WHERE NOT vend_id = 'DLL01'
ORDER BY prod_name;
```

Вывод ▼

```
prod_name
----------------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
King doll
Queen doll
```

Анализ ▼

Логический оператор NOT отрицает условие, следующее за ним. Поэтому СУБД извлекает не те значения vend_id, которые совпадают с DLL01, а все остальные.

Предыдущий запрос можно было бы записать при помощи оператора <>.

Ввод ▼

```
SELECT prod_name
FROM Products
WHERE vend_id <> 'DLL01'
ORDER BY prod_name;
```

Вывод ▼

```
prod_name
--------------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
King doll
Queen doll
```

Анализ ▼

Зачем же использовать логический оператор NOT? Конечно, для таких простых предложений WHERE, которые здесь рассмотрены, он не обязателен. Его преимущества проявляются в более сложных условиях. Например, для нахождения всех строк, которые не совпадают со списком критериев, можно использовать логический оператор NOT в связке с оператором IN.

> ПРИМЕЧАНИЕ: **оператор NOT в MariaDB**
>
> В MariaDB оператор NOT используется только для отрицания операторов IN, BETWEEN и EXISTS. Это отличается от большинства СУБД, которые допускают применение данного оператора для отрицания любых условий.

Резюме

На этом уроке вы узнали, как использовать логические операторы AND и OR в предложениях WHERE. Было также показано, как управлять порядком обработки операторов и как применять операторы IN и NOT.

УРОК 6

Фильтрация с использованием метасимволов

На этом уроке вы узнаете, что такое метасимволы, как их использовать и выполнять поиск с применением метасимволов и логического оператора `LIKE` *для фильтрации извлекаемых данных.*

Использование оператора `LIKE`

Все предыдущие операторы, которые мы рассмотрели, выполняли фильтрацию по известным значениям. Они искали совпадения по одному или нескольким значениям, осуществляли проверку "больше–меньше" или проверку на вхождение в диапазон значений. При этом везде отыскивалось известное значение.

Однако фильтрация данных таким способом не всегда работает. Например, как бы вы искали товары, в названии которых содержатся слова *bean bag*? Этого нельзя сделать в простых операциях сравнения, и здесь на помощь приходит поиск с использованием метасимволов. Благодаря метасимволам можно создавать расширенные условия отбора строк. В данном примере, для того чтобы найти все товары, в названии которых содержатся слова *bean bag*, необходимо составить шаблон поиска, позволяющий найти текст *bean bag* в любом месте названия товара.

Метасимволы
Специальные символы, применяемые для поиска части значения.

Шаблон поиска

Условие отбора строк, состоящее из текста, метасимволов и любой их комбинации.

Метасимволы сами по себе являются специальными знаками, которые особым образом трактуются в условии WHERE. В SQL поддерживаются метасимволы нескольких типов. Чтобы использовать метасимволы в условиях отбора строк, необходимо задействовать ключевое слово LIKE. Оно сообщает СУБД о том, что следующий шаблон поиска необходимо анализировать с учетом метасимволов, а не искать точные совпадения.

Предикат

Когда оператор не является оператором? Когда он является *предикатом*, задающим размер множества проверяемых значений. Технически LIKE — это предикат, а не оператор. Суть операции остается той же, просто не пугайтесь этого термина, если встретите его в документации по SQL.

Поиск с использованием метасимволов может осуществляться только в текстовых полях (строках). Метасимволы нельзя применять при поиске нетекстовых полей.

Метасимвол "знак процента" (%)

Наиболее часто используемый метасимвол — знак процента (%). В шаблоне поиска знак % означает *найти все вхождения любого символа*. Например, чтобы найти все товары, названия которых начинаются со слова Fish, можно выполнить следующий запрос.

Ввод ▼

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE 'Fish%';
```

Вывод ▼

```
prod_id       prod_name
---------     ------------------
BNBG01        Fish bean bag toy
```

Анализ ▼

В этом примере используется шаблон поиска `'Fish%'`. При проверке данного условия возвращаются все значения, которые начинаются с символов `Fish`. Знак `%` заставляет СУБД принимать все символы после слова `Fish` независимо от их количества.

> ПРИМЕЧАНИЕ: **метасимволы в Microsoft Access**
>
> При работе в Microsoft Access необходимо использовать символ * вместо `%`.

> ПРИМЕЧАНИЕ: **чувствительность к регистру символов**
>
> В зависимости от СУБД и ее конфигурации операции поиска могут быть чувствительны к регистру символов. В таком случае значение `Fish bean bag toy` не будет соответствовать условию `'fish%'`.

Метасимволы могут встречаться в любом месте шаблона поиска, причем в неограниченном количестве. В следующем примере используются два метасимвола, по одному на обоих концах шаблона.

Ввод ▼

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '%bean bag%';
```

Вывод ▼

```
prod_id       prod_name
---------     ------------------
BNBG01        Fish bean bag toy
BNBG02        Bird bean bag toy
BNBG03        Rabbit bean bag toy
```

Анализ ▼

Шаблон `'%bean bag%'` означает *найти все значения, содержащие текст* `bean bag` *в любом месте названия, независимо от количества символов перед указанным текстом или после него.*

Метасимвол может находиться в шаблоне поиска, хотя это редко бывает полезным. В следующем примере выполняется поиск всех товаров, названия которых начинаются на F и заканчиваются на y.

Ввод ▼

```
SELECT prod_name
FROM Products
WHERE prod_name LIKE 'F%y';
```

> СОВЕТ: **поиск фрагментов адресов электронной почты**
>
> Это как раз та ситуация, в которой метасимволы оказываются очень полезными в середине поискового шаблона, например `WHERE email LIKE 'b%@forta.com'`.

Важно отметить, что, помимо соответствия одному или нескольким символам, знак % также означает отсутствие символов в указанном месте поискового шаблона.

> ПРИМЕЧАНИЕ: **следите за замыкающими пробелами**
>
> Многие СУБД, включая Microsoft Access, заполняют содержимое поля пробелами. Например, если столбец рассчитан на 50 символов, а в него вставлен текст `Fish bean bag toy` (17 символов), то, чтобы заполнить столбец, в него могут быть добавлены еще 33 пробела. Обычно это не влияет на отображение данных, но может негативно сказаться на рассмотренном выше SQL-запросе. По условию `WHERE prod_name LIKE 'F%y'` будут найдены только те значения `prod_name`, которые начинаются на F и заканчиваются на y. Если значение дополнено пробелами, оно не будет заканчиваться на y и строка `Fish bean bag toy` не будет извлечена. Одним из простых решений может быть добавление второго символа % в шаблон поиска: `'F%y%'`, после чего будут учитываться любые символы (в том числе пробелы) после буквы y. Но лучше "отсечь" пробелы с помощью функций, которые рассматриваются на уроке 8.

> ПРЕДУПРЕЖДЕНИЕ: **следите за значениями NULL**
>
> Может показаться, будто метасимвол % соответствует чему угодно, но есть одно исключение: NULL. Даже условие вида WHERE prod_name LIKE '%' не позволит отобрать строку, в которой в поле названия товара содержится NULL.

Метасимвол "знак подчеркивания" (_)

Еще одним полезным метасимволом является знак подчеркивания (_). Он используется так же, как и знак %, но при этом учитывается не множество символов, а только один.

> ПРИМЕЧАНИЕ: **метасимволы в DB2**
>
> Метасимвол _ не поддерживается в DB2.

> ПРИМЕЧАНИЕ: **метасимволы в Microsoft Access**
>
> При работе в Microsoft Access используйте символ ? вместо _.

Рассмотрим пример.

Ввод ▼

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '__ inch teddy bear';
```

> ПРИМЕЧАНИЕ: **следите за замыкающими пробелами**
>
> Как и в предыдущем примере, возможно, понадобится добавить метасимвол % в конец шаблона, чтобы пример работал.

Вывод ▼

```
prod_id      prod_name
---------    ------------------
BNBG02       12 inch teddy bear
BNBG03       18 inch teddy bear
```

Анализ ▼

В шаблоне поиска этого предложения WHERE использованы два метасимвола, после которых следует текст. В результате были отобраны только те строки, которые удовлетворяют условию: по двум символам подчеркивания было найдено число 12 в первой строке и 18 — во второй. Товар 8 inch teddy bear не был отобран, так как в шаблоне поиска требуется два совпадения, а не одно. Для сравнения: в следующей инструкции SELECT используется метасимвол %, вследствие чего извлекаются три товара.

Ввод ▼

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '% inch teddy bear';
```

Вывод ▼

```
prod_id       prod_name
---------     ------------------
BNBG01        8 inch teddy bear
BNBG02        12 inch teddy bear
BNBG03        18 inch teddy bear
```

В отличие от знака %, который подразумевает также отсутствие символов, знак _ всегда означает один символ — не больше и не меньше.

Метасимвол "квадратные скобки" ([])

Метасимвол "квадратные скобки" ([]) служит для указания набора символов, каждый из которых должен совпадать с искомым значением, причем в точно указанном месте (в позиции метасимвола).

> ПРИМЕЧАНИЕ: **наборы не всегда поддерживаются**
>
> В отличие от описанных ранее метасимволов, использование квадратных скобок для создания проверочных наборов не поддерживается многими СУБД. Наборы поддерживаются в Microsoft Access и Microsoft SQL Server. Обратитесь к документации своей СУБД, чтобы определить, поддерживаются ли в ней наборы.

Например, чтобы найти всех клиентов, имена которых начинаются на букву J или M, необходимо выполнить следующий запрос.

Ввод ▼

```
SELECT cust_contact
FROM Customers
WHERE cust_contact LIKE '[JM]%'
ORDER BY cust_contact
```

Вывод ▼

```
cust_contact
------------------
Jim Jones
John Smith
Michelle Green
```

Анализ ▼

Условие WHERE в этой инструкции выглядит так: '[JM]%'. В данном шаблоне поиска используются два разных метасимвола. По выражению [JM] осуществляется поиск всех клиентов, имена которых начинаются на одну из указанных в скобках букв, но при этом учитывается только один символ. Поэтому все имена длиннее одного символа не будут извлечены. Благодаря метасимволу %, следующему после [JM], выполняется поиск любого количества символов после первой буквы, что и приводит к нужному результату.

Можно использовать метасимвол, обозначающий противоположное действие: ^. Например, в следующем примере возвращаются все имена, которые *не* начинаются с буквы J или M (в отличие от предыдущего примера).

Ввод ▼

```
SELECT cust_contact
FROM Customers
WHERE cust_contact LIKE '[^JM]%'
ORDER BY cust_contact
```

> ПРИМЕЧАНИЕ: **инверсия наборов в Microsoft Access**
>
> Если вы работаете в Microsoft Access и требуется создать противоположный набор, необходимо использовать символ ! вместо ^, поэтому указывайте [!JM], а не [^JM].

Конечно, можно достичь того же результата, воспользовавшись логическим оператором NOT. Единственным преимуществом символа ^ является более простой синтаксис при наличии нескольких условий WHERE.

Ввод ▼

```
SELECT cust_contact
FROM Customers
WHERE NOT cust_contact LIKE '[JM]%'
ORDER BY cust_contact
```

Советы по использованию метасимволов

Как видите, метасимволы в SQL — очень мощный механизм. Но за эту мощь приходится платить: на поиск с использованием метасимволов уходит больше времени, чем на любые другие виды поиска, которые рассматривались ранее. Ниже приведено несколько советов по использованию метасимволов.

- Не злоупотребляйте метасимволами. Если можно использовать другой оператор поиска, задействуйте его.
- При использовании метасимволов старайтесь по возможности не вставлять их в начало шаблона поиска. Шаблоны, начинающиеся с метасимволов, обрабатываются медленнее всего.
- Внимательно следите за позицией метасимволов. Если они находятся не на своем месте, будут извлечены не те данные.

Тем не менее следует сказать, что метасимволы очень важны и полезны при поиске, — вы часто будете ими пользоваться.

Резюме

На этом уроке рассказывалось о том, что такое метасимволы и как применять их в условиях WHERE. Теперь вы знаете, что метасимволы нужно использовать осторожно, не злоупотребляя ими.

УРОК 7

Создание вычисляемых полей

На этом уроке вы узнаете, что такое вычисляемые поля, как их создавать и как применять псевдонимы для ссылки на такие поля из приложений.

Что такое вычисляемые поля

Данные, хранимые в таблицах базы данных, обычно бывают представлены не в таком виде, который необходим в приложениях. Вот несколько примеров.

- Вам необходимо отобразить поле, содержащее название компании и ее адрес, но эта информация находится в разных столбцах таблицы.
- Город, штат и почтовый индекс хранятся в отдельных столбцах (как и должно быть), но для программы печати почтовых наклеек эта информация нужна в одном корректно отформатированном поле.
- Данные в столбце введены прописными и строчными буквами, но в отчете необходимо использовать только прописные.
- В таблице с элементами заказа хранится цена каждого товара и его количество, но не полная стоимость (цена одного товара, умноженная на его количество). Чтобы распечатать инвойс, нужно вычислить полные цены.
- Вам нужно знать общую сумму, среднее значение или другие итоговые показатели, основанные на данных, хранящихся в таблице.

В каждом из этих примеров данные хранятся не в том виде, в котором их необходимо предоставить приложению. Вместо того чтобы извлекать данные, а затем переформатировать их в клиентском

приложении или отчете, лучше извлекать уже преобразованные, подсчитанные или отформатированные данные прямо из базы данных.

Именно здесь на помощь приходят вычисляемые поля. Вообще-то, в таблицах базы данных никаких вычисляемых столбцов нет. Они создаются на лету инструкцией SELECT.

> **Поле**
>
> По сути то же самое, что и *столбец*. В основном эти термины взаимозаменяемы, хотя столбцы таблиц обычно называют *столбцами*, а термин *поле* чаще применяется по отношению к вычисляемым полям.

Важно отметить, что только база данных "знает", какие столбцы в инструкции SELECT являются реальными столбцами таблицы, а какие — вычисляемыми полями. С точки зрения клиента (например, пользовательского приложения) данные вычисляемого поля возвращаются точно так же, как и данные из любого другого столбца.

> ПРИМЕЧАНИЕ: **клиентское или серверное форматирование?**
>
> Многие преобразования и изменения форматов, которые могут быть выполнены посредством инструкций SQL, могут быть также реализованы и клиентским приложением. Однако, как правило, эти операции гораздо быстрее выполняются на сервере баз данных, чем на стороне клиента.

Конкатенация полей

Чтобы продемонстрировать применение вычисляемых полей, рассмотрим простой пример: создание заголовка, состоящего из двух столбцов.

В таблице Vendors хранится название поставщика и его страны. Предположим, необходимо создать отчет по поставщику и указать его страну как часть его имени в виде *имя* (*страна*).

В отчете должно быть одно значение, а данные в таблице хранятся в двух столбцах: vend_name и vend_country. Кроме того, значение vend_country необходимо заключить в скобки, которых

нет в таблице базы данных. Инструкция SELECT, которая возвращает имена поставщиков и адреса, довольно проста, но как создать комбинированное значение?

> **Конкатенация**
>
> Комбинирование значений (путем присоединения их друг к другу) для получения одного "длинного" значения.

Для этого необходимо соединить два значения. В инструкции SELECT можно выполнить конкатенацию двух столбцов при помощи специального оператора. В зависимости от СУБД это может быть знак "плюс" (+) или две вертикальные черточки (||). В случае MySQL и MariaDB придется использовать специальную функцию.

> ПРИМЕЧАНИЕ: **оператор + или ||?**
>
> В Access и SQL Server для конкатенации используется оператор +. В DB2, Oracle, PostgreSQL, SQLite и OpenOffice Base поддерживается оператор ||. За более подробной информацией обратитесь к документации своей СУБД.

Ниже приведен пример использования оператора + (синтаксис, принятый в большинстве СУБД).

Ввод ▼

```
SELECT vend_name + ' (' + vend_country + ')'
FROM Vendors
ORDER BY vend_name;
```

Вывод ▼

```
---------------------------------------------
Bear Emporium                (USA            )
Bears R Us                   (USA            )
Doll House Inc.              (USA            )
Fun and Games                (England        )
Furball Inc.                 (USA            )
Jouets et ours               (France         )
```

Ниже приведена та же инструкция, но с использованием оператора ||.

Ввод ▼

```
SELECT vend_name || ' (' || vend_country || ')'
FROM Vendors
ORDER BY vend_name;
```

Вывод ▼

```
-----------------------------------------------
Bear Emporium                (USA            )
Bears R Us                   (USA            )
Doll House Inc.              (USA            )
Fun and Games                (England        )
Furball Inc.                 (USA            )
Jouets et ours               (France         )
```

А вот эквивалентная инструкция для MySQL и MariaDB.

Ввод ▼

```
SELECT Concat(vend_name, ' (', vend_country, ')')
FROM Vendors
ORDER BY vend_name;
```

Анализ ▼

В предыдущих инструкциях SELECT была выполнена конкатенация следующих элементов:

- имя, хранящееся в столбце vend_name;
- строка, содержащая пробел и открывающую круглую скобку;
- название страны, хранящееся в столбце vend_country;
- строка, содержащая закрывающую круглую скобку.

Как видно из результатов запроса, инструкция SELECT возвращает один столбец (вычисляемое поле), содержащий все четыре элемента как одно целое.

Взгляните еще раз на результат, возвращаемый инструкцией SELECT. Два столбца, объединенных в вычисляемое поле, дополнены пробелами. Во многих базах данных (но не во всех) сохраненный текст дополняется пробелами до ширины столбца. Чтобы получить

правильно отформатированные данные, необходимо убрать лишние пробелы. Это можно сделать с помощью SQL-функции RTRIM().

Ввод ▼

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
FROM Vendors
ORDER BY vend_name;
```

Вывод ▼

```
-----------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

Ниже приведена та же самая инструкция, но с использованием оператора ||.

Ввод ▼

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
FROM Vendors
ORDER BY vend_name;
```

Вывод ▼

```
-----------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

Анализ ▼

Функция RTRIM() отбрасывает все пробелы справа от указанного значения. При использовании функции RTRIM() каждый столбец форматируется корректно.

> ПРИМЕЧАНИЕ: **функции семейства TRIM**
>
> В большинстве СУБД поддерживается не только функция RTRIM() (которая, как мы увидели, обрезает правую часть строки), но также LTRIM() (которая удаляет левую часть строки) и TRIM() (которая обрезает строку слева и справа).

Использование псевдонимов

Инструкция SELECT, которая использовалась для конкатенации полей адреса, как видите, справилась со своей задачей. Но как же называется новый вычисляемый столбец? По правде говоря, никак. Этого может быть достаточно, если вы просматриваете результаты в программе тестирования SQL-запросов, однако столбец без названия нельзя использовать в клиентском приложении, так как клиент не сможет к нему обратиться.

Для решения указанной проблемы в SQL была включена поддержка псевдонимов. Псевдоним — это альтернативное имя для поля или значения. Псевдонимы присваиваются с помощью ключевого слова AS. Рассмотрим следующую инструкцию SELECT.

Ввод ▼

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Вывод ▼

```
vend_title
----------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

Ниже приведена та же инструкция, но с использованием оператора ||.

Ввод ▼

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

А вот эквивалентная инструкция для MySQL и MariaDB.

Ввод ▼

```
SELECT Concat(vend_name, ' (', vend_country, ')')
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Анализ ▼

Сама по себе эта инструкция SELECT ничем не отличается от предыдущей, за исключением того, что после вычисляемого поля стоит выражение AS vend_title. Оно заставляет СУБД создать вычисляемое поле с именем vend_title. Как видите, результат остается тем же, но столбец теперь называется vend_title, и любое клиентское приложение может обращаться к нему по имени, как если бы это был реальный столбец таблицы.

> ПРИМЕЧАНИЕ: **ключевое слово AS часто является необязательным**
>
> Ключевое слово AS является необязательным во многих СУБД, но его применение считается общепринятой практикой.

> СОВЕТ: **другие применения псевдонимов**
>
> Псевдонимы можно использовать и по-другому. Часто они применяются для переименования столбца, если в реальном названии встречаются недопустимые символы (например, пробелы) или если название слишком длинное и трудночитаемое.

ПРЕДУПРЕЖДЕНИЕ: **имена псевдонимов**

Псевдонимом может служить как одно слово, так и целая строка. Если используется строка, она должна быть заключена в кавычки. В принципе, поступать подобным образом не рекомендуется. Многословные имена, несомненно, удобнее читать, но они создают множество проблем для клиентских приложений. Таким образом, чаще всего псевдонимы применяются для переименования многословных названий столбцов в однословные.

ПРИМЕЧАНИЕ: **производные столбцы**

Псевдонимы иногда называют производными столбцами, но, независимо от того, какой термин употребляется, означают они одно и то же.

Выполнение математических вычислений

Еще одним способом использования вычисляемых полей является выполнение математических операций над извлеченными данными. Рассмотрим пример. В таблице `Orders` хранятся все полученные заказы, а в таблице `OrderItems` — списки товаров для каждого заказа. Следующий запрос извлекает все товары, относящиеся к заказу с номером 20008.

Ввод ▼

```
SELECT prod_id, quantity, item_price
FROM OrderItems
WHERE order_nam = 20008;
```

Вывод ▼

```
prod_id     quantity    item_price
----------  ----------  -------------
RGAN01      5           4.9900
BR03        5           11.9900
BNBG01      10          3.4900
BNBG02      10          3.4900
BNBG03      10          3.4900
```

В столбце item_price содержится цена за единицу товара, включенного в заказ. Чтобы узнать полную стоимость (цена за единицу, умноженная на количество товаров в заказе), необходимо модифицировать запрос следующим образом.

Ввод ▼

```
SELECT prod_id,
       quantity,
       item_price
       quantity*item_price AS expanded_price
FROM OrderItems
WHERE order_nam = 20008;
```

Вывод ▼

```
prod_id    quantity    item_price    expanded_price
--------   ---------   ----------    ---------------
RGAN01      5           4.9900        24.9500
BR03        5           11.9900       59.9500
BNBG01      10          3.4900        34.9000
BNBG02      10          3.4900        34.9000
BNBG03      10          3.4900        34.9000
```

Анализ ▼

Столбец expanded_price в данном случае является вычисляемым полем. Вычисление было простым: quantity*item_price. Теперь клиентское приложение может использовать новый вычисляемый столбец, как и любой другой в таблице.

В SQL поддерживаются основные математические операторы, перечисленные в табл. 7.1. Не забывайте, что для управления порядком обработки операторов можно использовать круглые скобки (см. урок 5).

ТАБЛИЦА 7.1. **Математические операторы в SQL**

Оператор	Описание
+	Сложение
–	Вычитание
*	Умножение
/	Деление

СОВЕТ: **как тестировать вычисляемые выражения**

С помощью инструкции SELECT удобно экспериментировать с различными функциями и вычислениями. Обычно эта инструкция применяется для извлечения данных из таблицы, однако предложение FROM можно просто опустить и работать только с выражениями, указанными в списке столбцов. Например, инструкция `SELECT 3*2;` вернет 6, инструкция `SELECT Trim('  abc  ');` вернет abc, а инструкция `SELECT Now();` использует функцию `Now()` для определения текущих даты и времени.

Резюме

На этом уроке вы узнали, что такое вычисляемые поля и как их создавать. Были рассмотрены примеры использования вычисляемых полей для конкатенации строк и выполнения математических операций. Кроме того, было показано, как создавать и применять псевдонимы, чтобы клиентское приложение могло обращаться к вычисляемым полям.

УРОК 8

Использование функций обработки данных

На этом уроке вы узнаете, что такое функции, какие типы функций поддерживаются в СУБД, как применять функции и какие проблемы при этом могут возникать.

Что такое функция

Как и в большинстве других языков программирования, в SQL поддерживается использование функций для работы с данными. Функции — это операции, которые чаще всего приходится выполнять над данными, включая различные преобразования и вычисления.

Примером может служить функция `RTRIM()`, которую мы применяли на предыдущем уроке для удаления пробелов в конце строки.

Проблемы с функциями

Прежде чем переходить к примерам, обращаю ваше внимание на то, что использование SQL-функций может быть проблематичным.

В отличие от инструкций SQL (например, `SELECT`), которые в основном поддерживаются всеми СУБД одинаково, в разных СУБД могут применяться разные функции для одних и тех же целей. Лишь некоторые функции в различных СУБД называются одинаково. Общая функциональность доступна в каждой СУБД, но названия функций и их синтаксис могут существенно отличаться. Чтобы стало понятно, насколько это может быть проблематичным, в табл. 8.1 приведены три наиболее часто возникающие задачи и названия соответствующих функций в различных СУБД.

ТАБЛИЦА 8.1. **Различия в названиях функций**

Задача	Синтаксис
Выборка части строки	В Access используется функция `MID()`, в DB2, Oracle, PostgreSQL и SQLite — `SUBSTR()`, в MariaDB, MySQL и SQL Server — `SUBSTRING()`
Преобразование типа данных	В Access и Oracle используется несколько функций, по одной на каждый тип преобразования. В DB2 и PostgreSQL используется функция `CAST()`, в MariaDB, MySQL и SQL Server — `CONVERT()`
Получение текущей даты	В Access используется функция `NOW()`, в DB2 и PostgreSQL — `CURRENT_DATE`, в MariaDB и MySQL — `CURDATE()`, в Oracle — `SYSDATE`, в SQL Server — `GETDATE()`, в SQLite — `DATE()`

Как видите, в отличие от инструкций, SQL-функции не являются переносимыми. Это означает, что код, который написан для одной реализации SQL, может не работать в другой.

Переносимый код

Код, который может работать в разных системах.

Ставя перед собой цель обеспечить переносимость кода, многие SQL-программисты стараются не использовать зависящие от реализации функции. В целом это довольно разумная позиция, но она не всегда удачна с точки зрения производительности. Программисту приходится искать другие способы выполнения того, что СУБД сделала бы более эффективно.

СОВЕТ: **стоит ли использовать функции?**

Решение зависит от вас, и здесь нет правильного или неправильного выбора. Если вы решили использовать функции, добавляйте подробные комментарии к коду, чтобы в будущем вы (или другой разработчик) смогли понять, для какой реализации SQL писался данный код.

Применение функций

В большинстве реализаций SQL поддерживаются следующие типы функций.

- *Текстовые функции.* Используются для обработки текстовых строк (например, для отсечения пробелов, или заполнения строк пробелами, или преобразования символов в верхний либо нижний регистр).
- *Числовые функции.* Используются для выполнения математических операций над числовыми данными (таких, как возведение в степень, извлечение корня и т.п.).
- *Функции даты и времени.* Используются для обработки значений даты и времени, а также для извлечения отдельных компонентов этих значений (например, для определения разницы между датами и проверки корректности даты).
- *Системные функции.* Возвращают информацию, специфичную для конкретной СУБД (например, сведения об учетной записи пользователя).

На предыдущем уроке нам встречалась функция, которая использовалась в списке столбцов инструкции SELECT, но это допустимо не для всех функций. Функции можно применять как в других предложениях инструкции SELECT (например, в условии WHERE), так и в других инструкциях SQL (об этом вы узнаете на следующих уроках).

Функции для работы с текстом

На предыдущем уроке функция RTRIM() применялась для удаления пробелов в конце значения столбца. Ниже приведен другой пример, в котором используется функция UPPER().

Ввод ▼

```
SELECT vend_name, UPPER(vend_name) AS vend_name_upcase
FROM Vendors
ORDER BY vend_name;
```

Вывод ▼

```
vend_name                vend_name_upcase
-----------------------  --------------------
Bear Emporium            Bear Emporium
Bears R Us               Bears R Us
Doll House Inc.          Doll House Inc.
Fun and Games            Fun and Games
Furball Inc.             Furball Inc.
Jouets et ours           Jouets et ours
```

Анализ ▼

Функция `UPPER()` преобразует символы в верхний регистр, поэтому в данном примере имя каждого поставщика перечислено дважды: первый раз в том виде, в котором оно хранится в таблице `Vendors`, а второй раз — будучи преобразованным в верхний регистр, в виде столбца `vend_name_upcase`.

В табл. 8.2 приведены наиболее часто используемые текстовые функции.

ТАБЛИЦА 8.2. **Наиболее часто используемые текстовые функции**

Функция	Описание
`LEFT()`	Возвращает символы из левой части строки
`LENGTH` (а также `DATALENGTH()` или `LEN()`)	Возвращает длину строки
`LOWER()` (`LCASE()` в Access)	Преобразует строку в нижний регистр
`LTRIM()`	Удаляет пробелы в левой части строки
`RIGHT()`	Возвращает символы из правой части строки
`RTRIM()`	Удаляет пробелы в правой части строки
`SOUNDEX()`	Возвращает значение `SOUNDEX` строки
`UPPER()` (`UCASE()` в Access)	Преобразует строку в верхний регистр

Одна функция из табл. 8.2 требует более подробного объяснения. SOUNDEX — это алгоритм, преобразующий текстовую строку в буквенно-цифровой шаблон, описывающий фонетическое представление данного текста. Функция SOUNDEX() берет в расчет похожие по звучанию буквы и слоги, позволяя сравнивать строки не по тому, как они пишутся, а по тому, как они звучат. Несмотря на то что SOUNDEX() не соответствует основным концепциям SQL, большинство СУБД поддерживает эту функцию.

> ПРИМЕЧАНИЕ: **поддержка функции SOUNDEX()**
>
> Функция SOUNDEX() не поддерживается в Microsoft Access и PostgreSQL, поэтому следующий пример не будет работать в указанных СУБД.
>
> Кроме того, она будет доступна в SQLite, только если при сборке приложения был задан параметр компиляции SQLITE_SOUNDEX. А поскольку по умолчанию данный параметр не используется, в большинстве реализаций SQLite функция SOUNDEX() тоже не поддерживается.

Ниже приведен пример использования функции SOUNDEX(). Клиент Kids Place находится в таблице Customers, и его контактным лицом является Michelle Green. Но что если это опечатка, и на самом деле контактное лицо пишется как Michael Green? Очевидно, поиск по корректному имени ничего не даст, как показано ниже.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_contact = 'Michael Green';
```

Вывод ▼

```
cust_name                cust_contact
-----------------------  --------------------
```

А теперь попробуем выполнить поиск с помощью функции SOUNDEX(), чтобы найти все имена контактных лиц, которые звучат как Michael Green.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE SOUNDEX(cust_contact) = SOUNDEX('Michael Green');
```

Вывод ▼

```
cust_name                cust_contact
-----------------------  --------------------
Kids Place               Michelle Green
```

Анализ ▼

В данном примере в предложении WHERE используется функция SOUNDEX() для преобразования значения столбца cust_contact и искомой строки в их SOUNDEX-значения. Поскольку Michael Green и Michelle Green звучат схожим образом, их SOUNDEX-значения совпадут и предложение WHERE корректно отфильтрует необходимые данные.

Функции для работы с датой и временем

Значения даты и времени хранятся в таблицах с использованием соответствующих типов данных, которые в каждой СУБД свои. Благодаря наличию специальных форматов эти значения можно быстро отсортировать или отфильтровать, а кроме того, они занимают достаточно мало места на диске.

Формат, в котором хранятся значения даты и времени, обычно нельзя использовать в приложениях, поэтому почти всегда приходится применять специальные функции для извлечения этих значений и манипулирования ими. В результате такие функции являются одними из наиболее важных в SQL. К сожалению, они наименее согласованы и наименее совместимы в различных реализациях SQL.

Чтобы продемонстрировать применение этих функций, рассмотрим простой пример. В таблице Orders все заказы хранятся с датой заказа. Чтобы извлечь список всех заказов, сделанных в 2012 году, в SQL Server необходимо выполнить следующую инструкцию.

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE DATEPART(yy, order_date) = 2012;
```

Вывод ▼

```
order_num
------------
20005
20006
20007
20008
20009
```

В Access инструкция будет такой.

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE DATEPART('yyyy', order_date) = 2012;
```

Анализ ▼

В этом примере (в версиях для SQL Server и Access) применяется функция DATEPART(), которая, как видно из названия, возвращает только фрагмент даты. У функции DATEPART() два аргумента: фрагмент, подлежащий извлечению, и дата, из которой этот фрагмент извлекается. В рассматриваемом примере функция DATEPART() возвращает только год из столбца order_date. Путем сравнения полученного значения со значением 2012 предложение WHERE возвращает только те заказы, которые были сделаны в указанном году.

Ниже приведена версия данного примера для PostgreSQL, в которой используется похожая функция DATE_PART().

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE DATE_PART('year', order_date) = 2012;
```

В Oracle тоже нет функции DATEPART(), но существует несколько других функций работы с датами, позволяющих сделать то же самое. Рассмотрим пример.

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE to_number(to_char(order_date, 'YYYY')) = 2012;
```

Анализ ▼

В этом примере функция to_char() используется для извлечения компонента даты, а функция to_number() — для преобразования полученного значения в числовой вид, чтобы его можно было сравнить со значением 2012.

Аналогичных результатов можно добиться с помощью оператора BETWEEN.

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE order_date BETWEEN to_date('01-01-2012')
      AND to_date('12-31-2012');
```

Анализ ▼

В этом примере функция Oracle to_date() используется для преобразования двух строк в даты. В одной строке задается дата 1 января 2012 года, а в другой — 31 декабря 2012 года. Стандартный оператор BETWEEN позволяет осуществить поиск всех заказов, сделанных в период между этими двумя датами. Подобный код не будет работать в SQL Server, так как в этой СУБД не поддерживается функция to_date(). Однако если заменить функцию to_date() функцией DATEPART(), данный синтаксис можно будет применять.

В MySQL и MariaDB имеются все функции работы с датами, за исключением DATEPART(). Пользователи этих СУБД могут применять функцию YEAR() для извлечения номера года из даты.

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE YEAR(order_date) = 2012;
```

В SQLite формат инструкции чуть сложнее.

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE strftime('%Y', order_date) = 2012;
```

В этом примере извлекается только часть даты (год). Чтобы отфильтровать заказы по месяцу, необходимо сделать то же самое, добавив ключевое слово AND для сравнения месяца и года.

Различные СУБД обычно могут выполнять гораздо больше действий с датами. В большинстве из них имеются функции для сравнения дат и выполнения арифметических операций с датами, а также опции форматирования дат и многое другое. Но, как уже было показано, функции даты и времени сильно зависят от конкретной СУБД. Обратитесь к документации своей СУБД и уточните, какие функции работы с датой и временем в ней поддерживаются.

Функции для работы с числами

Числовые функции предназначены для обработки числовых данных. Они применяются в основном для выполнения алгебраических, тригонометрических и геометрических вычислений, поэтому потребность в них возникает не так часто, как в строковых функциях или функциях работы с датой и временем.

По иронии судьбы, в большинстве СУБД именно числовые функции наиболее стандартизированы. В табл. 8.3 приведены наиболее часто используемые числовые функции.

ТАБЛИЦА 8.3. **Наиболее часто используемые числовые функции**

Функция	Что возвращается
`ABS()`	Модуль числа
`COS()`	Косинус заданного угла
`EXP()`	Экспонента заданного числа
`PI()`	Число π
`SIN()`	Синус заданного угла
`SQRT()`	Квадратный корень заданного числа
`TAN()`	Тангенс заданного угла

Обратитесь к документации своей СУБД, чтобы узнать, какие числовые функции она поддерживает.

Резюме

На этом уроке объяснялось, как применять SQL-функции, предназначенные для обработки данных. Несмотря на то что они могут быть весьма полезными при форматировании и фильтрации данных, они весьма различны в разных реализациях SQL.

УРОК 9

Итоговые вычисления

На этом уроке вы узнаете, что такое итоговые SQL-функции и как их применять для обработки табличных данных.

Использование итоговых функций

Часто бывает необходимо подвести итоги, не отображая исходные данные, и в SQL для этого предусмотрены специальные функции. SQL-запросы с такими функциями часто используются для анализа данных и создания отчетов. Примерами подобных запросов могут служить:

- подсчет числа строк в таблице (либо числа строк, которые удовлетворяют какому-то условию или содержат определенное значение);
- определение суммы по набору строк в таблице;
- поиск наибольшего, наименьшего и среднего значений в столбце таблицы (по всем или каким-то конкретным строкам).

В каждом из этих примеров пользователю нужны итоговые сводки по таблице, а не исходные данные. Поэтому извлечение данных из таблицы было бы пустой тратой времени и ресурсов. Итак, все, что вам нужно, — только итоговая информация.

Чтобы облегчить извлечение подобной информации, в SQL предусмотрен набор из пяти итоговых функций, которые приведены в табл. 9.1. Эти функции позволяют выполнять все варианты запросов, которые были перечислены выше. В отличие от функций обработки данных из предыдущего урока, итоговые функции поддерживаются без особых изменений в большинстве реализаций SQL.

Итоговые функции

Функции, обрабатывающие набор строк для вычисления одного обобщающего значения.

ТАБЛИЦА 9.1. **Итоговые функции в SQL**

Функция	Что возвращается
AVG()	Среднее значение по столбцу
COUNT()	Число строк в столбце
MAX()	Наибольшее значение в столбце
MIN()	Наименьшее значение в столбце
SUM()	Сумма значений столбца

Функция AVG()

Функция AVG() предназначена для определения среднего значения по столбцу путем подсчета числа строк в таблице и суммирования их значений. Эту функцию можно применять для вычисления среднего значения всех столбцов или же определенных столбцов либо строк.

В первом примере функция AVG() используется для нахождения средней цены всех товаров в таблице Products.

Ввод ▼

```
SELECT AVG(prod_price) AS avg_price
FROM Products;
```

Вывод ▼

```
avg_price
-----------
6.823333
```

Анализ ▼

Приведенная выше инструкция SELECT возвращает одно значение, avg_price, соответствующее средней цене всех товаров в таблице Products. Здесь avg_price — это псевдоним (см. урок 7).

Функцию AVG() можно также применять для нахождения среднего значения по определенным столбцам или строкам. В следующем примере возвращается средняя цена товаров, предлагаемых поставщиком DLL01.

Ввод ▼

```
SELECT AVG(prod_price) AS avg_price
FROM Products
WHERE vend_id = 'DLL01';
```

Вывод ▼

```
avg_price
-----------
3.8650
```

Анализ ▼

Эта инструкция отличается от предыдущей только тем, что в ней содержится предложение WHERE. В соответствии с условием WHERE выбираются только те товары, значение vend_id для которых равно DLL01, поэтому значение, полученное в столбце с псевдонимом avg_price, является средним только для товаров данного поставщика.

> ПРЕДУПРЕЖДЕНИЕ: **только отдельные столбцы**
>
> Функцию AVG() можно использовать только для вычисления среднего значения конкретного числового столбца. Имя этого столбца должно быть указано в качестве аргумента функции. Чтобы определить среднее значение по нескольким столбцам, необходимо использовать несколько функций AVG().

> ПРИМЕЧАНИЕ: **значения NULL**
>
> Строки столбца, содержащие значения NULL, игнорируются функцией AVG().

Функция COUNT()

Функция COUNT() подсчитывает число строк. С ее помощью можно узнать общее число строк в таблице или число строк, удовлетворяющих определенному критерию.

Эту функцию можно использовать двумя способами:

- в виде выражения COUNT(*) для подсчета числа строк в таблице независимо от того, содержат столбцы значения NULL или нет;
- в виде выражения COUNT(*столбец*) для подсчета числа строк, которые имеют значения в указанных столбцах, причем значения NULL игнорируются.

В первом примере возвращается общее число имен клиентов, содержащихся в таблице Customers.

Ввод ▼

```
SELECT COUNT(*) AS num_cust
FROM Customers;
```

Вывод ▼

```
num_cust
-----------
5
```

Анализ ▼

В этом примере функция COUNT(*) используется для подсчета всех строк независимо от их значений. Сумма возвращается в виде столбца с псевдонимом num_cust.

В следующем примере подсчитываются только клиенты, имеющие адреса электронной почты.

Ввод ▼

```
SELECT COUNT(cust_email) AS num_cust
FROM Customers;
```

Вывод ▼

```
num_cust
-----------
3
```

Анализ ▼

В этой инструкции функция COUNT() используется для подсчета только строк, имеющих ненулевое значение в столбце cust_email. В данном случае число строк равно 3 (т.е. только 3 из 5 клиентов имеют адрес электронной почты).

> ПРИМЕЧАНИЕ: **значения NULL**
>
> Строки со значениями NULL игнорируются функцией COUNT(), если указано имя столбца, и учитываются, если используется звездочка (*).

Функция MAX()

Функция MAX() возвращает наибольшее значение в указанном столбце. Для этой функции необходимо задать имя столбца, как показано ниже.

Ввод ▼

```
SELECT MAX(prod_price) AS max_price
FROM Products;
```

Вывод ▼

```
max_price
-----------
11.9900
```

Анализ ▼

Здесь функция MAX() возвращает цену самого дорогого товара в таблице Products.

СОВЕТ: **использование функции MAX() с нечисловыми данными**

Несмотря на то что функция MAX() обычно используется для поиска наибольшего числового значения или даты, многие (но не все) СУБД позволяют применять ее для нахождения наибольшего значения среди всех столбцов, включая текстовые. При работе с текстовыми данными функция MAX() возвращает строку, которая была бы последней, если бы данные были отсортированы по этому столбцу.

ПРИМЕЧАНИЕ: **значения NULL**

Строки со значениями NULL игнорируются функцией MAX().

Функция MIN()

Функция MIN() выполняет противоположное по отношению к MAX() действие: она возвращает наименьшее значение в указанном столбце. В качестве аргумента также требуется указать имя столбца.

Ввод ▼

```
SELECT MIN(prod_price) AS min_price
FROM Products;
```

Вывод ▼

```
min_price
-----------
3.4900
```

Анализ ▼

Здесь функция MIN() возвращает цену самого дешевого товара в таблице Products.

> СОВЕТ: **использование функции MIN() с нечисловыми данными**
>
> Несмотря на то что функция MIN() обычно используется для поиска наименьшего числового значения или даты, многие (но не все) СУБД позволяют применять ее для нахождения наименьшего значения среди всех столбцов, включая текстовые. При работе с текстовыми данными функция MIN() возвращает строку, которая была бы первой, если бы данные были отсортированы по этому столбцу.

> ПРИМЕЧАНИЕ: **значения NULL**
>
> Строки со значениями NULL игнорируются функцией MIN().

Функция SUM()

Функция SUM() возвращает сумму значений в указанном столбце.

Рассмотрим пример. В таблице OrderItems содержатся элементы заказа, причем каждому элементу соответствует определенное количество, указанное в заказе. Общее число заказанных товаров (сумму всех значений столбца quantity) можно определить следующим образом.

Ввод ▼

```
SELECT SUM(quantity) AS item_ordered
FROM OrderItems
WHERE order_item = 20005;
```

Вывод ▼

```
item_ordered
-----------
200
```

Анализ ▼

Функция SUM(quantity) возвращает общее количество всех элементов заказа, а предложение WHERE гарантирует, что будут учитываться только товары из указанного заказа.

Функцию SUM() можно также применять для подсчета вычисляемых полей. В следующем примере общая стоимость заказа вычисляется путем суммирования выражений item_price*quantity по каждому элементу.

Ввод ▼

```
SELECT SUM(item_price*quantity) AS total_price
FROM OrderItems
WHERE order_item = 20005;
```

Вывод ▼

```
total_price
-----------
1648.0000
```

Анализ ▼

Функция SUM(item_price*quantity) возвращает сумму всех цен в заказе, а предложение WHERE гарантирует, что учитываться будут только товары из указанного заказа.

> СОВЕТ: **вычисления с несколькими столбцами**
>
> Все итоговые функции позволяют выполнять вычисления над несколькими столбцами с использованием стандартных математических операторов, как было показано в данном примере.

> ПРИМЕЧАНИЕ: **значения NULL**
>
> Строки со значениями NULL игнорируются функцией SUM().

Итоговые вычисления для уникальных значений

Все пять итоговых функций могут быть использованы двумя способами:

- для выполнения вычислений по всем строкам при наличии ключевого слова ALL или без указания какого-либо аргумента (так как ALL является аргументом по умолчанию);
- для выполнения вычислений по уникальным значениям при наличии ключевого слова DISTINCT.

> СОВЕТ: **аргумент ALL задан по умолчанию**
>
> Ключевое слово ALL не обязательно указывать, так как оно является аргументом по умолчанию. Если не задано ключевое слово DISTINCT, то подразумевается аргумент ALL.

> ПРИМЕЧАНИЕ: **только не в Access**
>
> Microsoft Access не поддерживает использование ключевого слова DISTINCT в итоговых функциях, поэтому следующий пример не будет работать в Access. Для получения аналогичного результата в этой СУБД необходимо создать в инструкции SELECT COUNT(*) подзапрос, возвращающий уникальные строки.

В следующем примере функция AVG() используется для определения средней цены товаров, предлагаемых заданным поставщиком. Это такая же инструкция SELECT, как и та, что была рассмотрена ранее, но теперь в ней указано ключевое слово DISTINCT — при вычислении среднего значения учитываются только уникальные цены.

Ввод ▼

```
SELECT AVG(DISTINCT prod_price) AS avg_price
FROM Products
WHERE vend_id = 'DLL01';
```

Вывод ▼

```
avg_price
-----------
4.2400
```

Анализ ▼

В этом примере вследствие наличия ключевого слова DISTINCT значение avg_price получается более высоким, так как в таблице есть несколько товаров с одинаково низкой ценой. Не учитывая их, мы получаем более высокую среднюю стоимость.

> ПРЕДУПРЕЖДЕНИЕ: **не используйте ключевое слово DISTINCT с функцией COUNT(*)**
>
> Ключевое слово DISTINCT можно использовать с функцией COUNT() только в том случае, если указано имя столбца. Его нельзя применять с функцией COUNT(*). Аналогично, ключевое слово DISTINCT должно стоять перед именем столбца, а не перед вычисляемым полем или выражением.

> СОВЕТ: **использование ключевого слова DISTINCT с функциями MIN() и MAX()**
>
> Несмотря на то что ключевое слово DISTINCT разрешается использовать с функциями MIN() и MAX(), реальной необходимости в этом нет. Минимальные и максимальные значения в столбце будут теми же, независимо от того, учитываются уникальные значения или нет.

> ПРИМЕЧАНИЕ: **дополнительные аргументы итоговых функций**
>
> Помимо ключевых слов DISTINCT и ALL, некоторые СУБД поддерживают дополнительные предикаты, такие как TOP и TOP PERCENT, позволяющие выполнять действия над подмножествами результатов запроса. Обратитесь к документации своей СУБД, чтобы узнать, какие ключевые слова можно использовать.

Комбинирование итоговых функций

Во всех примерах применения итоговых функций, приведенных до сих пор, использовалась только одна функция. Но в действительности инструкция SELECT может содержать столько итоговых функций, сколько нужно для запроса. Рассмотрим пример.

Ввод ▼

```
SELECT COUNT(*) AS num_items,
       MIN(prod_price) AS price_min,
       MAX(prod_price) AS price_max,
       AVG(prod_price) AS proce_avg
FROM Products;
```

Вывод ▼

```
num_items     price_min     price_max     price_avg
---------     ----------    ----------    ---------
9             3.4900        11.9900       6.823333
```

Анализ ▼

В данном случае в одной инструкции SELECT используются сразу четыре итоговые функции и возвращаются четыре значения (число элементов в таблице Products, самая высокая, самая низкая и средняя стоимость товаров).

> ПРЕДУПРЕЖДЕНИЕ: **имена псевдонимов**
>
> При указании псевдонимов для хранения результатов итоговой функции старайтесь не использовать реальные названия столбцов в таблице, поскольку во многих реализациях SQL такое поведение не приветствуется — будет выдано сообщение об ошибке.

Резюме

Итоговые функции предназначены для вычисления базовых статистических данных. В SQL поддерживаются пять таких функций,

каждая из которых может использоваться несколькими способами для получения только необходимых в данный момент результатов. Эти функции достаточно эффективны и обычно возвращают результат гораздо быстрее, чем если бы аналогичные вычисления выполнялись в клиентском приложении.

УРОК 10

Группировка данных

На этом уроке вы узнаете, как группировать данные таким образом, чтобы можно было подводить итоги по подмножеству записей таблицы. Для этого предназначены два предложения инструкции SELECT: GROUP BY и HAVING.

Принципы группировки данных

На предыдущем уроке вы узнали, что итоговые функции SQL можно применять для получения статистических показателей. Это позволяет подсчитывать число строк, вычислять суммы и средние значения, а также определять наибольшее и наименьшее значения, не прибегая к извлечению всех данных.

Прежде все итоговые вычисления выполнялись над всеми данными таблицы или над данными, которые соответствовали условию WHERE. В качестве напоминания приведем пример, в котором возвращается количество товаров, предлагаемых поставщиком DLL01.

Ввод ▼

```
SELECT COUNT(*) AS num_prods
FROM Products
WHERE vend_id = 'DLL01';
```

Вывод ▼

```
num_prods
----------
4
```

Но что если вы хотите узнать количество товаров, предлагаемых каждым поставщиком? Или выяснить, какие поставщики предлагают только один товар или, наоборот, несколько товаров?

Именно в таких случаях нужно использовать *группы*. Группировка дает возможность разделить все данные на логические наборы, благодаря чему становится возможным выполнение статистических вычислений отдельно по каждой группе.

Создание групп

Группы создаются с помощью предложения GROUP BY инструкции SELECT.

Лучше всего это можно продемонстрировать на конкретном примере.

Ввод ▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
GROUP BY vend_id;
```

Вывод ▼

```
vend_id     num_prods
---------   ---------
BRS01       3
DLL01       4
FNG01       2
```

Анализ ▼

Данная инструкция SELECT выводит два столбца: vend_id, содержащий идентификатор поставщика товара, и num_prods, содержащий вычисляемые поля (он создается с помощью функции COUNT(*)). Предложение GROUP BY заставляет СУБД отсортировать данные и сгруппировать их по столбцу vend_id. В результате значение num_prods будет вычисляться по одному разу для каждой группы записей vend_id, а не один раз для всей таблицы products. Как видите, в результатах указывается, что поставщик BRS01 предлагает три товара, поставщик DLL01 — четыре, а поставщик FNG01 — два.

Поскольку было использовано предложение GROUP BY, не пришлось указывать каждую группу, для которой должны быть выполнены вычисления. Это было сделано автоматически. Предложение GROUP BY заставляет СУБД сначала группировать данные, а затем выполнять вычисления по каждой группе, а не по всему набору результатов.

Прежде чем применять предложение GROUP BY, ознакомьтесь с важными правилами, которыми необходимо руководствоваться.

- В предложениях GROUP BY можно указывать произвольное число столбцов. Это позволяет вкладывать группы одна в другую, благодаря чему обеспечивается тщательный контроль над тем, какие данные подлежат группировке.
- Если в предложении GROUP BY используются вложенные группы, данные подытоживаются для последней указанной группы. Другими словами, если задана группировка, вычисления осуществляются для всех указанных столбцов (вы не сможете получить данные для каждого отдельного столбца).
- Каждый столбец, указанный в предложении GROUP BY, должен быть извлекаемым столбцом или выражением (но не итоговой функцией). Если в инструкции SELECT используется какое-то выражение, то же самое выражение должно быть указано в предложении GROUP BY. Псевдонимы применять нельзя.
- В большинстве реализаций SQL нельзя указывать в предложении GROUP BY столбцы, в которых содержатся данные переменной длины (например, текстовые поля или поля комментариев).
- За исключением инструкций, связанных с итоговыми вычислениями, каждый столбец, упомянутый в инструкции SELECT, должен быть представлен в предложении GROUP BY.
- Если столбец, по которому выполняется группировка, содержит строку со значением NULL, оно будет трактоваться как отдельная группа. Если имеется несколько строк со значениями NULL, они будут сгруппированы вместе.
- Предложение GROUP BY должно стоять после предложения WHERE и перед предложением ORDER BY.

> СОВЕТ: **ключевое слово ALL**
>
> В некоторых реализациях SQL (например, в Microsoft SQL Server) поддерживается необязательное ключевое слово ALL в предложении GROUP BY. Его можно применять для извлечения всех групп, даже тех, которые не имеют соответствующих строк (в таком случае итоговая функция возвращает значение NULL). Обратитесь к документации своей СУБД, чтобы узнать, поддерживает ли она ключевое слово ALL.

ПРЕДУПРЕЖДЕНИЕ: **указание столбцов по их относительному положению**

Некоторые реализации SQL позволяют указывать столбцы в предложении GROUP BY по их положению в списке инструкции SELECT. Например, выражение GROUP BY 2,1 может означать группировку по второму извлекаемому столбцу, а затем — по первому. И хотя такой сокращенный синтаксис довольно удобен, он поддерживается не всеми реализациями SQL. Его применение также является рискованным в том смысле, что весьма высока вероятность возникновения ошибок при редактировании инструкций SQL.

Фильтрация по группам

SQL позволяет не только группировать данные с помощью предложения GROUP BY, но и осуществлять их фильтрацию, т.е. указывать, какие группы должны быть включены в результаты запроса, а какие — исключены из них. Например, вам может понадобиться список клиентов, которые сделали хотя бы два заказа. Чтобы получить такие данные, необходим фильтр, относящийся к целой группе, а не к отдельным строкам.

Вы уже знаете, как работает предложение WHERE (см. урок 4). Однако в данном случае его нельзя использовать, поскольку условия WHERE касаются строк, а не групп. Собственно говоря, предложение WHERE "не знает", что такое группы.

Но что тогда следует применить вместо предложения WHERE? В SQL предусмотрено другое предложение, подходящее для этих целей: HAVING. Оно очень напоминает предложение WHERE. И действительно, все типы выражений в предложении WHERE, с которыми вы уже знакомы, допустимы и в предложении HAVING. Единственная разница состоит в том, что WHERE фильтрует строки, а HAVING — группы.

СОВЕТ: **предложение HAVING поддерживает все операторы предложения WHERE**

На уроках 4 и 5 было показано, как применять предложение WHERE (включая использование метасимволов и логических операторов). Все эти метасимволы и операторы поддерживаются и в предложении HAVING. Синтаксис точно такой же, отличаются только начальные слова.

Как же осуществляется фильтрация по группам? Рассмотрим следующий пример.

Ввод ▼

```
SELECT cust_id, COUNT(*) AS orders
FROM Orders
GROUP BY cust_id
HAVING COUNT(*) >= 2;
```

Вывод ▼

```
cust_id       orders
----------    ----------
1000000001    2
```

Анализ ▼

Первые три строки этого запроса напоминают инструкцию SELECT, рассмотренную ранее. Однако в последней строке появляется предложение HAVING, которое фильтрует группы с помощью выражения COUNT(*) >= 2 — два или больше заказов.

Как видите, предложение WHERE здесь не работает, поскольку фильтрация основана на итоговом значении группы, а не на значениях отобранных строк.

> ПРИМЕЧАНИЕ: **разница между предложениями HAVING и WHERE**
>
> Вот как это можно объяснить: предложение WHERE фильтрует строки до того, как данные будут сгруппированы, а предложение HAVING — после того, как данные были сгруппированы. Это важное различие. Строки, которые были исключены по условию WHERE, не войдут в группу, иначе это могло бы изменить вычисляемые значения, которые, в свою очередь, могли бы повлиять на фильтрацию групп в предложении HAVING.

А теперь подумаем: возникает ли необходимость в использовании как предложения WHERE, так и предложения HAVING в одной инструкции? Конечно, возникает. Предположим, вы хотите усовершенствовать фильтр предыдущей инструкции таким образом, чтобы возвращались имена всех клиентов, которые сделали два или более заказа за последние 12 месяцев. Чтобы добиться этого, можно

добавить предложение WHERE, которое учитывает только заказы, сделанные за последние 12 месяцев. Затем вы добавляете предложение HAVING, чтобы отфильтровать только те группы, в которых имеются минимум две строки.

Чтобы лучше разобраться в этом, рассмотрим следующий пример, в котором перечисляются все поставщики, предлагающие не менее двух товаров по цене 4 доллара и более за единицу.

Ввод ▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
WHERE prod_price >= 4
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

Вывод ▼

```
vend_id       num_prods
----------     -----------
BRS01          3
FNG01          2
```

Анализ ▼

Данный пример нуждается в пояснении. Первая строка представляет собой основную инструкцию SELECT, использующую итоговую функцию, — точно так же, как и в предыдущих примерах. Предложение WHERE фильтрует все строки со значениями в столбце prod_price не менее 4. Затем данные группируются по столбцу vend_id, после чего предложение HAVING фильтрует только группы, содержащие не менее двух членов. При отсутствии предложения WHERE была бы получена лишняя строка (поставщик, предлагающий четыре товара, каждый из которых дешевле 4 долларов), как показано ниже.

Ввод ▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

Вывод ▼

```
vend_id      num_prods
----------   ----------
BRS01        3
DLL01        4
FNG01        2
```

ПРИМЕЧАНИЕ: **использование предложений HAVING и WHERE**

Предложение HAVING столь сильно напоминает предложение WHERE, что в большинстве СУБД оно трактуется точно так же, если только не указано предложение GROUP BY. Тем не менее следует знать, что между ними существует разница. Используйте предложение HAVING только вместе с предложением GROUP BY, а предложение WHERE — для стандартной фильтрации на уровне строк.

Группировка и сортировка

Важно понимать, что предложения GROUP BY и ORDER BY существенно различаются, хотя с их помощью иногда можно добиться одинаковых результатов. Разобраться в этом поможет табл. 10.1.

ТАБЛИЦА 10.1. **Сравнение предложений ORDER BY и GROUP BY**

ORDER BY	GROUP BY
Сортирует полученные результаты	Группирует строки. Однако отображаемый результат может не соответствовать порядку группировки
Могут быть использованы любые столбцы (даже не указанные в предложении SELECT)	Могут быть использованы только извлекаемые столбцы или выражения; должно быть указано каждое выражение из предложения SELECT
Не является необходимым	Требуется, если используются столбцы (или выражения) с итоговыми функциями

Первое из отличий, перечисленных в табл. 10.1, является очень важным. Чаще всего вы обнаружите, что данные, сгруппированные

с помощью предложения GROUP BY, будут отображаться в порядке группировки. Но так будет не всегда, и в действительности этого не требуется в спецификациях SQL. Более того, даже если СУБД сортирует данные так, как указано в предложении GROUP BY, вам может понадобиться отсортировать их по-другому. То, что вы группируете данные определенным способом (чтобы получить для группы необходимые итоговые значения), не означает, что требуемый результат должен быть отсортирован именно так. Следует явным образом указывать предложение ORDER BY, даже если оно совпадает с предложением GROUP BY.

> СОВЕТ: **не забывайте использовать предложение ORDER BY**
>
> Как правило, всякий раз, когда вы используете предложение GROUP BY, приходится указывать и предложение ORDER BY. Это единственный способ, гарантирующий, что данные будут отсортированы правильно. Не следует надеяться на то, что данные будут отсортированы предложением GROUP BY.

Чтобы продемонстрировать совместное использование предложений GROUP BY и ORDER BY, рассмотрим пример. Следующая инструкция SELECT аналогична тем, которые использовались ранее: она выводит номер заказа и количество товаров для всех заказов, которые содержат три товара или больше.

Ввод ▼

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3;
```

Вывод ▼

```
order_num      items
----------     -----
20006          3
20007          5
20008          5
20009          3
```

Чтобы отсортировать результат по количеству заказанных товаров, все, что необходимо сделать, — это добавить предложение ORDER BY, как показано ниже.

Ввод ▼

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3;
ORDER BY items, order_num;
```

> ПРИМЕЧАНИЕ: **несовместимость с Access**
>
> Microsoft Access не позволяет осуществлять сортировку по псевдонимам, и для данной СУБД этот пример неприменим. Выход состоит в замене столбца items (в предложении ORDER BY) вычисляемым выражением или номером поля. По существу, будут работать оба варианта: ORDER BY COUNT(*), order_num и ORDER BY 1, order_num.

Вывод ▼

```
order_num      items
----------     -----
20006          3
20009          3
20007          5
20008          5
```

Анализ ▼

В этом примере предложение GROUP BY используется для группировки данных по номеру заказа (столбец order_num), благодаря чему функция COUNT(*) может вернуть количество товаров в каждом заказе. Предложение HAVING фильтрует данные таким образом, что возвращаются только заказы с тремя и более товарами. Наконец, результат сортируется за счет использования предложения ORDER BY.

Порядок предложений в инструкции SELECT

Предложения инструкции SELECT должны указываться в определенном порядке. В табл. 10.2 перечислены все предложения, которые мы изучили до сих пор, в порядке, в котором они должны следовать.

ТАБЛИЦА 10.2. **Предложения инструкции SELECT и порядок их следования**

Предложение	Описание	Необходимость
SELECT	Столбцы или выражения, которые должны быть получены	Да
FROM	Таблица для извлечения данных	Только если извлекаются данные из таблицы
WHERE	Фильтрация на уровне строк	Нет
GROUP BY	Определение группы	Только если выполняются итоговые вычисления по группам
HAVING	Фильтрация на уровне групп	Нет
ORDER BY	Порядок сортировки результатов	Нет

Резюме

На предыдущем уроке вы узнали, как применять итоговые функции SQL для выполнения сводных вычислений. На этом уроке рассказывалось о том, как использовать предложение GROUP BY для выполнения аналогичных вычислений по отношению к группам данных и получения отдельных результатов для каждой группы. Было показано, как с помощью предложения HAVING осуществлять фильтрацию на уровне групп. Кроме того, объяснялось, какова разница между предложениями ORDER BY и GROUP BY, а также между предложениями WHERE и HAVING.

УРОК 11

Подзапросы

На этом уроке вы узнаете, что такое подзапросы и как их применять.

Что такое подзапросы

Инструкции SELECT — это запросы SQL. Все инструкции, с которыми мы имели дело до сих пор, представляли собой простые запросы: посредством отдельных инструкций извлекались данные из определенных таблиц.

> **Запрос**
>
> Какая-либо инструкция SQL. Однако чаще всего этот термин используют по отношению к инструкциям SELECT.

В SQL можно также создавать *подзапросы*: запросы, которые вложены в другие запросы. Почему возникает необходимость в подзапросах? Лучший способ объяснить эту концепцию — рассмотреть несколько примеров.

> ПРИМЕЧАНИЕ: **поддержка в MySQL**
>
> Если вы работаете с MySQL, то учтите, что подзапросы поддерживаются этой СУБД начиная с версии 4.1. В более ранних версиях MySQL примеры из данного урока работать не будут.

Фильтрация с помощью подзапросов

Таблицы баз данных, используемые во всех примерах книги, являются реляционными (см. приложение А, в котором описана каждая из таблиц). Заказы хранятся в двух таблицах. Таблица Orders

содержит по одной строке для каждого заказа; в ней указываются номер заказа, идентификатор клиента и дата заказа. Отдельные элементы заказов хранятся в таблице OrderItems. Таблица Orders не содержит информацию о клиентах. Она хранит только идентификатор клиента. Информация о клиентах находится в таблице Customers.

Теперь предположим, что вы хотите получить список всех клиентов, которые заказали товар RGAN01. Для этого необходимо выполнить следующее:

1) извлечь номера всех заказов, содержащих товар RGAN01;
2) получить идентификаторы всех клиентов, которые сделали заказы, перечисленные на предыдущем шаге;
3) извлечь информацию обо всех клиентах, идентификаторы которых были получены на предыдущем шаге.

Каждый из этих пунктов можно выполнить в виде отдельного запроса. Поступая так, вы используете результаты, возвращаемые одной инструкцией SELECT, чтобы заполнить предложение WHERE для следующей инструкции SELECT.

Но можно также воспользоваться подзапросами для того, чтобы объединить все три запроса в одну-единственную инструкцию.

Первая инструкция SELECT извлекает столбец order_num для всех элементов заказов, у которых в столбце prod_id значится RGAN01. Результат представляет собой номера двух заказов, содержащих данный товар.

Ввод ▼

```
SELECT order_num
FROM OrderItems
WHERE prod_id = 'RGAN01';
```

Вывод ▼

```
order_num
-----------
20007
20008
```

Следующий шаг состоит в получении идентификаторов клиентов, связанных с заказами 20007 и 20008. Используя предложение IN, о котором говорилось на уроке 5, можно создать показанную ниже инструкцию SELECT.

Ввод ▼

```
SELECT cust_id
FROM Orders
WHERE order_num IN (20007,20008);
```

Вывод ▼

```
cust_id
----------
1000000004
1000000005
```

Теперь объединим эти два запроса путем превращения первого из них (того, который возвращает номера заказов) в подзапрос.

Ввод ▼

```
SELECT cust_id
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE prod_id = 'RGAN01');
```

Вывод ▼

```
cust_id
----------
1000000004
1000000005
```

Анализ ▼

Подзапросы всегда обрабатываются, начиная с самой внутренней инструкции SELECT в направлении "изнутри наружу". При обработке предыдущей инструкции СУБД в действительности выполняет две операции.

Вначале она выполняет следующий подзапрос:

```
SELECT order_num FROM OrderItems WHERE prod_id='RGAN01'
```

В результате возвращаются два номера заказа: 20007 и 20008. Эти два значения затем передаются в предложение WHERE внешнего запроса в формате с разделителем в виде запятой, необходимом для оператора IN. Теперь внешний запрос становится таким:

```
SELECT cust_id FROM orders WHERE order_num IN (20007,20008)
```

Как видите, результат корректен и является точно таким же, как и полученный путем жесткого кодирования предложения WHERE в предыдущем примере.

> СОВЕТ: **форматируйте SQL-запросы**
>
> Инструкции SELECT, содержащие подзапросы, могут оказаться трудными для чтения и отладки, особенно если их сложность возрастает. Разбиение запросов на несколько строк и выравнивание строк отступами, как показано в рассматриваемых примерах, значительно облегчает работу с подзапросами.

Теперь у нас есть идентификаторы всех клиентов, заказавших товар RGAN01. Следующий шаг состоит в получении клиентской информации для каждого из этих идентификаторов. Инструкция SQL, осуществляющая выборку двух столбцов, выглядит так.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN ('1000000004','1000000005');
```

Но вместо жесткого указания идентификаторов клиентов можно превратить данное предложение WHERE в подзапрос.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id =
                                            'RGAN01'));
```

Вывод ▼

```
cust_name                cust_contact
-----------------------  ------------------
Fun4All                  Denise L. Stephens
The Toy Store            Kim Howard
```

Чтобы выполнить такой запрос, СУБД должна по сути обработать три инструкции SELECT. Самый внутренний подзапрос возвращает перечень номеров заказов, который затем используется как предложение WHERE для подзапроса, внешнего по отношению к данному. Этот подзапрос возвращает перечень идентификаторов клиентов, которые используются в предложении WHERE запроса самого высокого уровня. Запрос верхнего уровня возвращает искомые данные.

Как видите, благодаря подзапросам можно создавать очень мощные и гибкие инструкции SQL. Не существует ограничений на число подчиненных запросов, хотя на практике можно столкнуться с ощутимым снижением производительности, которое подскажет вам, что было использовано слишком много уровней подзапросов.

> ПРЕДУПРЕЖДЕНИЕ: **только один столбец**
>
> Инструкции SELECT в подзапросах могут возвращать только один столбец. Попытка извлечь несколько столбцов приведет к появлению сообщения об ошибке.

> ПРЕДУПРЕЖДЕНИЕ: **подзапросы и производительность**
>
> Представленные здесь примеры работают и приводят к достижению необходимых результатов. Однако подзапросы — не всегда самый эффективный способ получения данных такого типа. Более подробно об этом рассказывается на уроке 12, где повторно будет рассмотрен тот же самый пример.

Использование подзапросов в качестве вычисляемых полей

Другой способ использования подзапросов заключается в создании вычисляемых полей. Предположим, необходимо вывести общее количество заказов, сделанных каждым клиентом из таблицы Customers (клиенты). Заказы хранятся в таблице Orders вместе с соответствующими идентификаторами клиентов.

Чтобы выполнить эту операцию, необходимо сделать следующее:

1) извлечь список клиентов из таблицы Customers;
2) для каждого выбранного клиента подсчитать число его заказов в таблице Orders.

Как объяснялось на предыдущих двух уроках, можно выполнить инструкцию SELECT COUNT(*) для подсчета строк в таблице, а используя предложение WHERE для фильтрации идентификатора конкретного клиента, можно подсчитать заказы только этого клиента. Например, с помощью следующего запроса можно подсчитать количество заказов, сделанных клиентом 1000000001.

Ввод ▼

```
SELECT COUNT(*) AS orders
FROM Orders
WHERE cust_id = '1000000001';
```

Чтобы получить итоговую информацию посредством функции COUNT(*) для каждого клиента, используйте выражение COUNT(*) как подзапрос. Рассмотрим следующий пример.

Ввод ▼

```
SELECT cust_name,
       cust_state,
       (SELECT COUNT(*)
        FROM Orders
        WHERE Orders.cust_id = Customers.cust_id) AS
orders
FROM Customers
ORDER BY cust_name;
```

Вывод ▼

```
cust_name          cust_state  orders
------------------ ----------  ------
Fun4All            IN          1
Fun4All            AZ          1
Kids Place         OH          0
The Toy Store      IL          1
Village Toys       MI          2
```

Анализ ▼

Эта инструкция SELECT возвращает три столбца для каждого клиента из таблицы Customers: cust_name, cust_state и orders. Поле Orders является вычисляемым; оно формируется

в результате выполнения подзапроса, который заключен в круглые скобки. Подзапрос выполняется один раз для каждого выбранного клиента. В приведенном примере подзапрос выполняется пять раз, потому что были получены имена пяти клиентов.

Предложение WHERE в подзапросе несколько отличается от предложений WHERE, с которыми мы работали ранее, потому что в нем используются полные имена столбцов. Следующее предложение требует от СУБД, чтобы было проведено сравнение значения cust_id в таблице Orders с тем, которое в данный момент извлекается из таблицы Customers.

```
WHERE Orders.cust_id = Customers.cust_id
```

Подобный синтаксис — имя таблицы и имя столбца разделяются точкой — должен применяться всякий раз, когда может возникнуть неопределенность в именах столбцов. В данном примере имеются два столбца cust_id: один — в таблице Customers, другой — в таблице Orders. Без использования полностью определенных имен столбцов СУБД будет считать, что вы сравниваете поле cust_id в таблице Orders с самим собой. Поэтому следующий запрос будет всегда возвращать общее число заказов в таблице Orders, а это не тот результат, который нам нужен.

```
SELECT COUNT(*) FROM Orders WHERE cust_id = cust_id
```

Ввод ▼

```
SELECT cust_name,
       cust_state,
       (SELECT COUNT(*)
        FROM Orders
        WHERE cust_id = cust_id) AS orders
FROM Customers
ORDER BY cust_name;
```

Вывод ▼

```
cust_name      cust_state    orders
-------------- ----------    ------
Fun4All        IN            5
Fun4All        AZ            5
Kids Place     OH            5
The Toy Store  IL            5
Village Toys   MI            5
```

Подзапросы чрезвычайно полезны при создании инструкций SELECT такого типа, однако внимательно следите за тем, чтобы были правильно указаны неоднозначные имена столбцов.

> ПРЕДУПРЕЖДЕНИЕ: **полностью определенные имена столбцов**
>
> Вы только что наглядно убедились, почему так важно указывать полностью определенные имена столбцов. Без дополнительных уточнений СУБД вернула неправильные результаты, потому что не смогла правильно интерпретировать ваши намерения. В некоторых случаях неопределенность с названиями столбцов способна даже привести к появлению сообщения об ошибке. Это может, например, произойти, если предложение WHERE или ORDER BY содержит имя столбца, встречающееся в нескольких таблицах. Вот почему хорошей практикой является указание полностью определенных имен столбцов всякий раз, когда в инструкции SELECT перечислено несколько таблиц. Это позволит избежать любых неопределенностей.

> СОВЕТ: **подзапросы не всегда являются оптимальным решением**
>
> Несмотря на то что показанный здесь пример работоспособен, зачастую он оказывается не самым эффективным способом извлечения данных такого типа. Мы еще раз рассмотрим этот пример на одном из следующих уроков.

Резюме

На этом уроке вы узнали, что такое подзапросы и как их применять. Чаще всего подзапросы используются в операторах IN предложения WHERE, а также для заполнения вычисляемых столбцов. Были приведены примеры операций обоих типов.

УРОК 12

Объединение таблиц

На этом уроке вы узнаете, что такое объединения, для чего они нужны и как создавать инструкции SELECT, использующие объединения.

Что такое объединение

Одной из ключевых особенностей SQL является возможность на лету объединять таблицы при выполнении запросов, связанных с извлечением данных. Объединения — это самые мощные операции, которые можно выполнить с помощью инструкции SELECT, поэтому понимание объединений и их синтаксиса является важной частью процесса изучения SQL.

Прежде чем вы сможете эффективно применять объединения, следует разобраться, что такое реляционные таблицы и как проектируются реляционные базы данных. В столь маленькой книге полностью осветить такую обширную тему не удастся, но нескольких уроков будет вполне достаточно для того, чтобы вы смогли получить общее представление.

Что такое реляционные таблицы

Понять, что представляют собой реляционные таблицы, поможет пример из реальной жизни.

Предположим, определенная таблица базы данных содержит каталог товаров, в котором для каждого элемента выделена одна строка. Информация, хранящаяся о каждом товаре, должна включать описание товара и его цену, а также сведения о компании, выпустившей данный товар.

Теперь предположим, что в каталоге имеется целая группа товаров от одного поставщика. Где следует хранить информацию о поставщике (такую, как название компании, адрес и контактная информация)? Эти сведения не рекомендуется хранить вместе с данными о товарах по нескольким причинам.

- Информация о поставщике одна и та же для всех его товаров. Повторение этой информации для каждого товара приведет к напрасной потере времени и места на диске.
- Если информация о поставщике изменяется (например, если он переезжает или изменяется его почтовый код), вам придется обновить все записи о его товарах.
- Когда данные повторяются (а такое происходит, когда информация о поставщике указывается для каждого товара), высока вероятность того, что где-то данные будут введены с ошибкой. Несовместимые данные очень трудно использовать при создании отчетов.

Отсюда можно сделать вывод, что хранить множество экземпляров одних и тех же данных крайне нежелательно. Именно этот принцип и лежит в основе реляционных баз данных. Реляционные таблицы разрабатываются таким образом, что вся информация распределяется по множеству таблиц, причем для данных каждого типа создается отдельная таблица. Эти таблицы соотносятся (связываются) между собой через общие поля.

В нашем примере можно создать две таблицы: одну — для хранения информации о поставщике, другую — о его товарах. Таблица `Vendors` содержит информацию о поставщиках, по одной строке для каждого поставщика, с обязательным указанием его уникального идентификатора. Это значение называется *первичным ключом*.

В таблице `Products` хранится только информация о товарах, но нет никакой конкретной информации о поставщиках, за исключением их идентификаторов (первичного ключа таблицы `Vendors`). Этот ключ связывает таблицу `Vendors` с таблицей `Products`. Благодаря применению идентификатора поставщика можно использовать таблицу `Vendors` для поиска информации о соответствующем поставщике.

Что это дает? Ниже указаны ключевые преимущества.

- Информация о поставщике никогда не повторяется, благодаря чему экономится время, требуемое для заполнения базы данных, а также место на диске.
- Если информация о поставщике изменяется, достаточно обновить всего одну запись о нем, единственную в таблице `Vendors`. Данные в связанных с нею таблицах изменять не нужно.

- Поскольку никакие данные не повторяются, они, очевидно, оказываются непротиворечивыми, благодаря чему составление отчетов значительно упрощается.

Таким образом, данные в реляционных таблицах хранятся достаточно эффективно, и ими легко манипулировать. Вот почему реляционные базы данных масштабируются значительно лучше, чем базы данных других типов.

Масштабирование

Возможность справляться со все возрастающей нагрузкой без сбоев. О хорошо спроектированных базах данных или приложениях говорят, что они хорошо масштабируются.

Зачем нужны объединения

Распределение данных по многим таблицам обеспечивает их более эффективное хранение, упрощает обработку данных и повышает масштабируемость базы данных в целом. Однако эти преимущества не достигаются даром — за все приходится платить.

Если данные хранятся во многих таблицах, то как их извлечь с помощью одной инструкции SELECT?

Ответ таков: посредством объединений. *Объединение* представляет собой механизм слияния таблиц в инструкции SELECT. Используя особый синтаксис, можно объединить несколько исходных таблиц в одну общую, которая будет на лету связывать нужные строки из каждой таблицы.

ПРИМЕЧАНИЕ: **использование интерактивных инструментов СУБД**

Важно понимать, что объединение не является физической таблицей — другими словами, оно не существует как реальная таблица в базе данных. Объединение создается СУБД по мере необходимости и сохраняется только на время выполнения запроса. Многие СУБД предлагают графический интерфейс, который можно использовать для интерактивного определения связей таблицы. Эти инструменты могут оказаться чрезвычайно полезными для поддержания ссылочной целостности.

При использовании реляционных таблиц важно, чтобы в связанные столбцы заносились только корректные данные. Вернемся к нашему примеру: если в таблице Products хранится недостоверный идентификатор поставщика, соответствующие товары окажутся недоступными, поскольку они не будут относиться ни к одному поставщику. Во избежание этого база данных должна позволять пользователю вводить только достоверные значения (т.е. такие, которые представлены в таблице Vendors) в столбце идентификаторов поставщиков в таблице Products. Ссылочная целостность означает, что СУБД заставляет пользователя соблюдать правила, обеспечивающие непротиворечивость данных. И контроль этих правил часто обеспечивается благодаря интерфейсам СУБД.

Создание объединения

Создание объединения — очень простая процедура. Нужно указать все таблицы, которые должны быть включены в объединение, а также подсказать СУБД, как они должны быть связаны между собой. Рассмотрим следующий пример.

Ввод ▼

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products
WHERE Vendors.vend_id = Products.vend_id;
```

Вывод ▼

```
vend_name       prod_name           prod_price
---------------  ------------------  ----------
Doll House Inc.  Fish bean bag toy   3.4900
Doll House Inc.  Bird bean bag toy   3.4900
Doll House Inc.  Rabbit bean bag toy 3.4900
Bears R Us       8 inch teddy bear   5.9900
Bears R Us       12 inch teddy bear  8.9900
Bears R Us       18 inch teddy bear  11.9900
Doll House Inc.  Raggedy Ann         4.9900
Fun and Games    King doll           9.4900
Fun and Games    Queen doll          9.4900
```

Анализ ▼

Инструкция SELECT начинается точно так же, как и все инструкции, которые мы до сих пор рассматривали, — с указания столбцов, которые должны быть извлечены. Ключевая разница состоит в том, что два из указанных столбцов (prod_name и prod_price) находятся в одной таблице, а третий (vend_name) — в другой.

Взгляните на предложение FROM. В отличие от предыдущих инструкций SELECT, оно содержит две таблицы: Vendors и Products. Это имена двух таблиц, которые должны быть объединены в данном запросе. Таблицы корректно объединяются в предложении WHERE, которое заставляет СУБД связать идентификатор поставщика vend_id из таблицы Vendors с полем vend_id таблицы Products.

Обратите внимание на то, что эти столбцы указаны как Vendors.vend_id и Products.vend_id. Полностью определенные имена необходимы здесь потому, что, если вы укажете только vend_id, СУБД не сможет понять, на какие именно столбцы vend_id вы ссылаетесь (их два, по одному в каждой таблице). Как видно из представленных результатов, одна инструкция SELECT сумела извлечь данные из двух разных таблиц.

> ПРЕДУПРЕЖДЕНИЕ: **полностью определенные имена столбцов**
>
> Используйте полностью определенные имена столбцов (в которых названия таблиц и столбцов разделяются точкой) всякий раз, когда может возникнуть неоднозначность относительно того, на какой столбец вы ссылаетесь. В большинстве СУБД будет выдано сообщение об ошибке, если вы введете неоднозначное имя столбца, не определив его полностью путем указания имени таблицы.

Важность предложения WHERE

Использование предложения WHERE для установления связи между таблицами может показаться странным, но на то есть весомая причина. Вспомните: когда таблицы объединяются в инструкции SELECT, отношение создается на лету. В определениях таблиц базы данных ничего не говорится о том, как СУБД должна объединять их. Вы должны указать это сами. Когда вы объединяете две таблицы,

то в действительности создаете пары, состоящие из каждой строки первой таблицы и каждой строки второй таблицы. Предложение WHERE действует как фильтр, позволяющий включать в результат только строки, которые соответствуют указанному условию фильтрации — в данном случае условию объединения. Без предложения WHERE каждая строка в первой таблице будет образовывать пару с каждой строкой второй таблицы независимо от того, есть ли логика в их объединении или нет.

Декартово произведение

Результаты, возвращаемые при слиянии таблиц без указания условия объединения. Количество полученных строк будет равно числу строк в первой таблице, умноженному на число строк во второй таблице.

Для того чтобы разобраться в этом, рассмотрим следующую инструкцию SELECT и результат ее выполнения.

Ввод ▼

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products;
```

Вывод ▼

```
vend_name       prod_name           prod_price
--------------  ------------------  ----------
Bears R Us      8 inch teddy bear   5.99
Bears R Us      12 inch teddy bear  8.99
Bears R Us      18 inch teddy bear  11.99
Bears R Us      Fish bean bag toy   3.49
Bears R Us      Bird bean bag toy   3.49
Bears R Us      Rabbit bean bag toy 3.49
Bears R Us      Raggedy Ann         4.99
Bears R Us      King doll           9.49
Bears R Us      Queen doll          9.49
Bear Emporium   8 inch teddy bear   5.99
Bear Emporium   12 inch teddy bear  8.99
Bear Emporium   18 inch teddy bear  11.99
Bear Emporium   Fish bean bag toy   3.49
Bear Emporium   Bird bean bag toy   3.49
Bear Emporium   Rabbit bean bag toy 3.49
```

```
Bear Emporium      Raggedy Ann          4.99
Bear Emporium      King doll            9.49
Bear Emporium      Queen doll           9.49
Doll House Inc.    8 inch teddy bear    5.99
Doll House Inc.    12 inch teddy bear   8.99
Doll House Inc.    18 inch teddy bear   11.99
Doll House Inc.    Fish bean bag toy    3.49
Doll House Inc.    Bird bean bag toy    3.49
Doll House Inc.    Rabbit bean bag toy  3.49
Doll House Inc.    Raggedy Ann          4.99
Doll House Inc.    King doll            9.49
Doll House Inc.    Queen doll           9.49
Furball Inc.       8 inch teddy bear    5.99
Furball Inc.       12 inch teddy bear   8.99
Furball Inc.       18 inch teddy bear   11.99
Furball Inc.       Fish bean bag toy    3.49
Furball Inc.       Bird bean bag toy    3.49
Furball Inc.       Rabbit bean bag toy  3.49
Furball Inc.       Raggedy Ann          4.99
Furball Inc.       King doll            9.49
Furball Inc.       Queen doll           9.49
Fun and Games      8 inch teddy bear    5.99
Fun and Games      12 inch teddy bear   8.99
Fun and Games      18 inch teddy bear   11.99
Fun and Games      Fish bean bag toy    3.49
Fun and Games      Bird bean bag toy    3.49
Fun and Games      Rabbit bean bag toy  3.49
Fun and Games      Raggedy Ann          4.99
Fun and Games      King doll            9.49
Fun and Games      Queen doll           9.49
Jouets et ours     8 inch teddy bear    5.99
Jouets et ours     12 inch teddy bear   8.99
Jouets et ours     18 inch teddy bear   11.99
Jouets et ours     Fish bean bag toy    3.49
Jouets et ours     Bird bean bag toy    3.49
Jouets et ours     Rabbit bean bag toy  3.49
Jouets et ours     Raggedy Ann          4.99
Jouets et ours     King doll            9.49
Jouets et ours     Queen doll           9.49
```

Анализ ▼

Как видно из представленных результатов, декартово произведение вы, скорее всего, будете использовать очень редко. Данные, полученные таким способом, ставят в соответствие каждому товару

каждого поставщика, включая товары с указанием "не того" поставщика (и даже поставщиков, которые вообще не предлагают никаких товаров).

> ПРЕДУПРЕЖДЕНИЕ: **не забудьте указать предложение WHERE**
>
> Проверьте, включили ли вы в запрос предложение WHERE, иначе СУБД вернет намного больше данных, чем вам нужно. Кроме того, убедитесь в том, что предложение WHERE сформулировано правильно. Некорректное условие фильтрации приведет к тому, что СУБД выдаст неверные данные.

> **Перекрестное объединение**
>
> Иногда объединение, которое возвращает декартово произведение, называют перекрестным объединением.

Внутренние объединения

Объединение, которое мы до сих пор использовали, называется объединением по равенству — оно основано на проверке равенства записей двух таблиц. Объединение такого типа называют также *внутренним объединением*. Для подобных объединений можно применять несколько иной синтаксис, явно указывающий на тип объединения. Следующая инструкция SELECT возвращает те же самые данные, что и в предыдущем примере.

Ввод ▼

```
SELECT vend_name, prod_name, prod_price
FROM Vendors INNER JOIN Products
  ON Vendors.vend_id = Products.vend_id;
```

Анализ ▼

Предложение SELECT здесь точно такое же, как и в предыдущем случае, а вот предложение FROM другое. В данном запросе отношение между двумя таблицами определяется в предложении FROM,

содержащем спецификацию INNER JOIN. При использовании такого синтаксиса условие объединения задается с помощью специального предложения ON, а не WHERE. Фактическое условие, указываемое в предложении ON, то же самое, которое задавалось бы в предложение WHERE.

Обратитесь к документации своей СУБД, чтобы узнать, какой синтаксис предпочтительнее использовать.

> ПРИМЕЧАНИЕ: **"правильный" синтаксис**
>
> Согласно спецификации ANSI SQL, предпочтителен синтаксис INNER JOIN. В то же время большинство СУБД поддерживает оба синтаксиса. Изучите оба формата и применяйте тот из них, который кажется вам более удобным.

Объединение нескольких таблиц

SQL не ограничивает число таблиц, которые могут быть объединены посредством инструкции SELECT. Основные правила создания объединения остаются теми же. Вначале перечисляются все таблицы, а затем определяются отношения между ними. Рассмотрим пример.

Ввод ▼

```
SELECT prod_name, vend_name, prod_price, quantity
FROM OrderItems, Products, Vendors
WHERE Products.vend_id = Vendors.vend_id
  AND OrderItems.prod_id = Products.prod_id
  AND order_num = 20007;
```

Вывод ▼

```
prod_name           vend_name       prod_price   quantity
------------------- --------------- ---------- -------
18 inch teddy bear  Bears R Us      11.9900      50
Fish bean bag toy   Doll House Inc. 3.4900       100
Bird bean bag toy   Doll House Inc. 3.4900       100
Rabbit bean bag toy Doll House Inc. 3.4900       100
Raggedy Ann         Doll House Inc. 4.9900       50
```

Анализ ▼

В этом примере выводятся элементы заказа номер 20007. Все они находятся в таблице OrderItems. Каждый элемент хранится в соответствии с идентификатором, который ссылается на товар в таблице Products. Эти товары связаны с соответствующими поставщиками в таблице Vendors по идентификатору поставщика, который хранится вместе с каждой записью о товаре. В предложении FROM данного запроса перечисляются три таблицы, а предложение WHERE определяет оба названных условия объединения. Дополнительное условие служит для фильтрации только элементов заказа 20007.

> ПРЕДУПРЕЖДЕНИЕ: **к вопросу о производительности**
>
> Все СУБД обрабатывают объединения динамически, затрачивая время на обработку каждой указанной таблицы. Этот процесс может оказаться очень ресурсоемким, поэтому не следует использовать объединения таблиц без особой надобности. Чем больше таблиц вы объединяете, тем ниже производительность.

> ПРЕДУПРЕЖДЕНИЕ: **максимальное число таблиц в объединении**
>
> Несмотря на то что SQL не накладывает каких-либо ограничений на число таблиц в объединении, многие СУБД на самом деле имеют такие ограничения. Обратитесь к документации своей СУБД, чтобы узнать, какие ограничения такого рода она налагает (если они есть).

Теперь самое время вернуться к примеру из урока 11, в котором инструкция SELECT возвращала список клиентов, заказавших товар RGAN01.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id =
                                            'RGAN01'));
```

Анализ ▼

Как упоминалось на уроке 11, подзапросы не всегда являются самым эффективным способом выполнения сложных инструкций SELECT, поэтому тот же самый запрос можно переписать с использованием синтаксиса объединений.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
  AND OrderItems.order_num = Orders.order_num
  AND prod_id = 'RGAN01';
```

Вывод ▼

```
cust_name          cust_contact
-----------------  -----------------
Fun4All            Denise L. Stephens
The Toy Store      Kim Howard
```

Анализ ▼

Как уже говорилось на уроке 11, получение необходимых для этого запроса данных требует обращения к трем таблицам. Однако вместо подзапросов здесь были применены два объединения таблиц, а также указаны три условия WHERE. Первые два связывают таблицы в объединение, а последнее фильтрует данные по товару RGAN01.

> СОВЕТ: **экспериментируйте**
>
> Как видите, часто существует несколько способов выполнения одного и того же SQL-запроса. И редко удается однозначно сказать, какой из них правильный. Производительность может зависеть от типа операции, используемой СУБД, количества данных в таблицах, наличия либо отсутствия индексов и ключей, а также целого ряда других критериев. Следовательно, зачастую бывает целесообразно поэкспериментировать с различными типами запросов для выяснения того, какой из них работает быстрее.

Резюме

Объединения — одно из самых важных и востребованных средств SQL, но их эффективное применение возможно только на основе знаний о структуре реляционной базы данных. На этом уроке вы ознакомились с основами построения баз данных, а также узнали, как создавать объединение по равенству (называемое также внутренним объединением), которое используют чаще всего. На следующем уроке вы научитесь создавать объединения других типов.

УРОК 13

Создание расширенных объединений

На этом уроке вы узнаете о дополнительных типах объединений — что они собой представляют и когда они нужны. Вы также узнаете, как применять псевдонимы таблиц и использовать итоговые функции совместно с объединениями.

Использование псевдонимов таблиц

На уроке 7 вы узнали, как использовать псевдонимы в качестве ссылок на извлекаемые столбцы таблицы. Синтаксис псевдонимов столбцов выглядит следующим образом.

Ввод ▼

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Псевдонимы можно применять не только для имен столбцов и вычисляемых полей, но и вместо имен таблиц. На то есть две основные причины:

- сокращение синтаксиса запросов;
- возможность многоразового использования одной и той же таблицы в инструкции SELECT.

Рассмотрим следующую инструкцию SELECT. В основном она такая же, как и в примерах предыдущего урока, но здесь она модифицирована с учетом псевдонимов.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM Customers AS C, Orders AS O, OrderItems AS OI
WHERE C.cust_id = O.cust_id
  AND OI.order_num = O.order_num
  AND prod_id = 'RGAN01';
```

Анализ ▼

Заметьте, что все три таблицы в предложениях FROM имеют псевдонимы. Например, выражение Customers AS C задает C в качестве псевдонима для таблицы Customers, что позволяет использовать сокращение C вместо полного имени Customers. В данном примере псевдонимы таблиц указаны только в предложении WHERE, но их можно применять и в других местах, например в списке извлекаемых таблиц, в предложении ORDER BY, а также в любой другой части инструкции SELECT.

> СОВЕТ: **в Oracle нет ключевого слова AS**
>
> Oracle не поддерживает ключевое слово AS. Чтобы создать псевдоним в Oracle, просто укажите его без ключевого слова AS, например Customers C вместо Customers AS C.

Следует отметить, что псевдонимы таблиц существуют только во время выполнения запроса. В отличие от псевдонимов столбцов, они никогда не сообщаются клиенту.

Объединения других типов

До сих пор мы применяли только простые объединения, которые называются внутренними. Теперь рассмотрим три других типа объединения: самообъединение, естественное объединение и внешнее объединение.

Самообъединения

Одна из основных причин для использования псевдонимов таблиц состоит в возможности обращения к одной и той же таблице несколько раз в одной инструкции SELECT. Продемонстрируем это на примере.

Предположим, вы хотите послать письма по всем контактным адресам клиентов, которые работают с той же компанией, с которой работает Джим Джонс. Такой запрос требует, чтобы вначале вы выяснили, с какой компанией работает Джим Джонс, а затем — какие клиенты работают с этой же компанией. Один из способов решения данной задачи приведен ниже.

Ввод ▼

```
SELECT cust_id, cust_name, cust_contact
FROM Customers
WHERE cust_name = (SELECT cust_name
                   FROM Customers
                   WHERE cust_contact = 'Jim Jones');
```

Вывод ▼

```
cust_id       cust_name      cust_contact
----------    ---------------  --------------
1000000003    Fun4All        Jim Jones
1000000004    Fun4All        Denise L. Stephens
```

Анализ ▼

В первом решении используются подзапросы. Внутренняя инструкция SELECT возвращает название компании (cust_name), с которой работает Джим Джонс. Именно это название используется в предложении WHERE внешнего запроса, благодаря чему извлекаются имена всех служащих, работающих с данной компанией. (О подзапросах см. урок 11.)

Теперь рассмотрим тот же самый запрос, но с использованием объединений.

Ввод ▼

```
SELECT c1.cust_id, c1.cust_name, c1.cust_contact
FROM Customers AS c1, Customers AS c2
WHERE c1.cust_name = c2.cust_name
  AND c2.cust_contact = 'Jim Jones';
```

Вывод ▼

```
cust_id          cust_name     cust_contact
----------       ----------    -------------
1000000003       Fun4All       Jim Jones
1000000004       Fun4All       Denise L. Stephens
```

> СОВЕТ: **в Oracle нет ключевого слова AS**
>
> Пользователи Oracle, не забывайте убирать из своих инструкций ключевое слово AS.

Анализ ▼

Две таблицы, необходимые для выполнения запроса, на самом деле являются одной и той же таблицей, поэтому таблица `Customers` появляется в предложении `FROM` дважды. И хотя это совершенно законно, любые ссылки на таблицу `Customers` оказались бы неоднозначными, потому что СУБД не знает, на какую именно таблицу `Customers` вы ссылаетесь.

Для решения данной проблемы и предназначены псевдонимы. Первый раз для таблицы `Customers` назначается псевдоним `c1`, а второй раз — псевдоним `c2`. Теперь эти псевдонимы можно применять в качестве имен таблиц. В частности, инструкция `SELECT` использует префикс `c1` для однозначного указания полного имени нужного столбца. Если этого не сделать, СУБД выдаст сообщение об ошибке, потому что имеется по два столбца с именами `cust_id`, `cust_name` и `cust_contact`. СУБД не может знать, какой именно столбец вы имеете в виду (даже если в действительности это один и тот же столбец). Первое предложение `WHERE` объединяет обе копии таблицы, а затем фильтрует данные второй таблицы по столбцу `cust_contact`, чтобы вернуть только нужные данные.

СОВЕТ: **самообъединения вместо подзапросов**

Самообъединения часто применяются для замены инструкций с подзапросами, которые извлекают данные из той же таблицы, что и внешняя инструкция. Несмотря на то что конечный результат получается тем же самым, многие СУБД обрабатывают объединения гораздо быстрее, чем подзапросы. Стоит поэкспериментировать с тем и другим, чтобы определить, какой запрос работает быстрее.

Естественные объединения

Всякий раз, когда объединяются таблицы, по крайней мере один столбец будет появляться более чем в одной таблице (по нему и выполняется объединение). Обычные объединения (внутренние, которые мы рассмотрели на предыдущем уроке) возвращают все данные, даже многократные вхождения одного и того же столбца. Естественное объединение просто удаляет эти многократные вхождения, и в результате возвращается только один столбец.

Естественным называется объединение, в котором извлекаются только не повторяющиеся столбцы. Обычно это делается с помощью метасимвола (SELECT *) для одной таблицы и указания явного подмножества столбцов для всех остальных таблиц. Рассмотрим пример.

Ввод ▼

```
SELECT C.*, O.order_num, O.order_date,
       OI.prod_id, OI.quantity, OI.item_price
FROM Customers AS C, Orders AS O,
     OrderItems AS OI
WHERE C.cust_id = O.cust_id
  AND OI.order_num = O.order_num
  AND prod_id = 'RGAN01';
```

СОВЕТ: **в Oracle нет ключевого слова AS**

Пользователи Oracle, не забывайте убирать из кода ключевое слово AS.

Анализ ▼

В этом примере метасимвол * используется только для первой таблицы. Все остальные столбцы указаны явно, поэтому никакие дубликаты столбцов не извлекаются.

В действительности каждое внутреннее объединение, которое мы использовали до сих пор, представляло собой естественное объединение, и, возможно, вам никогда не понадобится внутреннее объединение, не являющееся естественным.

Внешние объединения

Большинство объединений связывают строки одной таблицы со строками другой, но в некоторых случаях вам может понадобиться включать в результат строки, не имеющие пар. Например, объединения можно использовать для решения следующих задач:

- подсчет количества заказов каждого клиента, включая клиентов, которые еще не сделали заказ;
- составление перечня товаров с указанием количества заказов на них, включая товары, которые никем не были заказаны;
- вычисление средних объемов продаж с учетом клиентов, которые еще не сделали заказ.

В каждом из этих случаев объединение должно включать строки, не имеющие ассоциированных с ними строк в связанной таблице. Объединение такого типа называется внешним.

> ПРЕДУПРЕЖДЕНИЕ: **различия в синтаксисе**
>
> Важно отметить, что синтаксис внешнего объединения может несколько отличаться в разных реализациях SQL. Различные формы синтаксиса, описанные далее, охватывают большинство реализаций, но все же, прежде чем начинать работу, обратитесь к документации своей СУБД и уточните, какой синтаксис необходимо применять.

Следующая инструкция SELECT позволяет выполнить простое внутреннее объединение. Она извлекает список всех клиентов и их заказы.

Ввод ▼

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers INNER JOIN Orders
  ON Customers.cust_id = Orders.cust_id;
```

Синтаксис внешнего объединения похож на этот. Для получения имен всех клиентов, включая тех, которые еще не сделали заказов, можно сделать следующее.

Ввод ▼

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers LEFT OUTER JOIN Orders
  ON Customers.cust_id = Orders.cust_id;
```

Вывод ▼

```
cust_id       order_num
----------    ---------
1000000001    20005
1000000001    20009
1000000002    NULL
1000000003    20006
1000000004    20007
1000000005    20008
```

Анализ ▼

Аналогично внутреннему объединению, которое мы рассматривали на прошлом уроке, в этой инструкции SELECT используется спецификация OUTER JOIN для указания типа объединения (в предложении FROM, а не WHERE). Но, в отличие от внутренних объединений, которые связывают строки двух таблиц, внешние объединения включают в результат также строки, не имеющие пар. При использовании спецификации OUTER JOIN необходимо указать ключевое слово RIGHT или LEFT, чтобы определить таблицу, все строки которой будут включены в результаты запроса (RIGHT для таблицы, имя которой стоит справа от OUTER JOIN, и LEFT — для той, имя которой значится слева). В предыдущем примере используется спецификация LEFT OUTER JOIN для извлечения всех строк таблицы, указанной в левой части предложения FROM (таблицы Customers).

Чтобы извлечь все строки из таблицы, указанной справа, используйте правое внешнее объединение (RIGHT OUTER JOIN), как показано в следующем примере.

Ввод ▼

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers RIGHT OUTER JOIN Orders
  ON Orders.cust_id = Customers.cust_id;
```

> ПРЕДУПРЕЖДЕНИЕ: **внешние объединения в SQLite**
>
> SQLite поддерживает левое внешнее объединение, но не правое. К счастью, существует очень простое решение, объясняемое в следующем совете.

> СОВЕТ: **типы внешних объединений**
>
> Существуют две основные формы внешнего объединения: левое и правое. Единственная разница между ними состоит в порядке указания связываемых таблиц. Другими словами, левое внешнее объединение может быть превращено в правое просто за счет изменения порядка указания имен таблиц в предложении FROM или WHERE. А раз так, то эти два типа внешнего объединения могут заменять друг друга, и решение о том, какое именно из них нужно использовать, определяется личными предпочтениями.

Существует и другой вариант внешнего объединения — полное внешнее объединение, которое извлекает все строки из обеих таблиц и связывает между собой те, которые могут быть связаны. В отличие от левого и правого внешних объединений, которые включают в результат несвязанные строки только из одной таблицы, полное внешнее объединение включает в результат несвязанные строки из обеих таблиц. Синтаксис полного внешнего объединения таков.

Ввод ▼

```
SELECT Customers.cust_id, Orders.order_num
FROM Orders FULL OUTER JOIN Customers
  ON Orders.cust_id = Customers.cust_id;
```

> ПРЕДУПРЕЖДЕНИЕ: **поддержка полного внешнего объединения**
>
> Синтаксис `FULL OUTER JOIN` не поддерживается в Access, MariaDB, MySQL, OpenOffice Base и SQLite.

Использование объединений совместно с итоговыми функциями

Как было показано на уроке 9, итоговые функции служат для получения базовых статистических показателей. Во всех рассмотренных до сих пор примерах итоговые функции применялись только для одной таблицы, но их можно использовать и по отношению к объединениям.

Рассмотрим пример. Допустим, вы хотите получить список всех клиентов и число сделанных ими заказов. Для этого в следующем запросе применяется функция `COUNT()`.

Ввод ▼

```
SELECT Customers.cust_id,
       COUNT(Orders.order_num) AS num_ord
FROM Customers INNER JOIN Orders
  ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

Вывод ▼

```
cust_id      num_ord
----------   --------
1000000001   2
1000000003   1
1000000004   1
1000000005   1
```

Анализ ▼

В этой инструкции используется спецификация `INNER JOIN` для связи таблиц `Customers` и `Orders` между собой. Предложение

GROUP BY группирует данные по клиентам, и, таким образом, вызов функции COUNT(Orders.order_num) позволяет подсчитать количество заказов для каждого клиента и вернуть результат в виде столбца num_ord.

Итоговые функции можно также использовать с объединениями других типов.

Ввод ▼

```
SELECT Customers.cust_id,
       COUNT(Orders.order_num) AS num_ord
FROM Customers LEFT OUTER JOIN Orders
  ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

> СОВЕТ: **в Oracle нет ключевого слова AS**
>
> Еще раз напоминаю пользователям Oracle о необходимости удаления ключевого слова AS из кода запроса.

Вывод ▼

```
cust_id       num_ord
----------    -------
1000000001    2
1000000002    0
1000000003    1
1000000004    1
1000000005    1
```

Анализ ▼

В этом примере используется левое внешнее объединение для включения в результат всех клиентов, даже тех, которые не сделали ни одного заказа. Как видите, клиент 1000000002 также включен в список, хотя на данный момент у него ноль заказов.

Правила создания объединений

Прежде чем завершить обсуждение объединений, которое заняло два урока, имеет смысл напомнить о ключевых моментах, касающихся объединений.

- Будьте внимательны при выборе типа объединения. Возможно, что чаще вы будете применять внутреннее объединение, хотя в зависимости от ситуации это может быть и внешнее объединение.
- Посмотрите в документации к СУБД, какой именно синтаксис объединений она поддерживает. (Большинство СУБД поддерживает одну из форм синтаксиса, описанных на этих двух уроках.)
- Проверьте, правильно ли указано условие объединения (независимо от используемого синтаксиса), иначе будут получены неверные данные.
- Не забывайте указывать условие объединения, в противном случае вы получите декартово произведение таблиц.
- Можно включать в объединение несколько таблиц и даже применять для каждой из них свой тип объединения. Несмотря на то что это допустимо и часто оказывается полезным, желательно проверить каждое объединение отдельно, прежде чем применять их вместе. Это намного упростит поиск ошибок.

Резюме

Этот урок стал продолжением предыдущего, посвященного объединениям. Вначале было показано, как и для чего используют псевдонимы, а затем мы продолжили рассмотрение объединений различных типов и вариантов синтаксиса для каждого из них. Вы также узнали, как применять итоговые функции совместно с объединениями и какие правила важно соблюдать при использовании объединений.

УРОК 14

Комбинированные запросы

На этом уроке вы узнаете, как применять оператор `UNION` *для объединения нескольких инструкций* `SELECT` *с целью получения единого набора результатов.*

Что такое комбинированные запросы

В большинстве SQL-запросов применяется одна инструкция `SELECT`, посредством которой извлекаются данные из одной или нескольких таблиц. SQL позволяет также выполнять множественные запросы (за счет многократного использования инструкции `SELECT`) и возвращать результаты в виде единого набора. Такие запросы обычно называются *соединениями*, или *комбинированными запросами*.

Комбинированные запросы обычно нужны в двух ситуациях:

- получение одинаковым образом структурированных данных из различных таблиц посредством одного запроса;
- выполнение многократных запросов к одной таблице и получение данных в виде единого набора.

> СОВЕТ: **комбинированные запросы и многократные условия `WHERE`**
>
> Результат комбинирования двух запросов к одной и той же таблице в основном аналогичен результату, полученному при выполнении одного запроса с несколькими условиями в предложении `WHERE`. Другими словами, как будет показано в следующем разделе, любую инструкцию `SELECT` с несколькими условиями `WHERE` тоже можно рассматривать как комбинированный запрос.

Создание комбинированных запросов

Запросы в SQL комбинируются с помощью оператора UNION, который позволяет многократно указывать инструкцию SELECT, возвращая один набор результатов.

Использование оператора UNION

Использовать оператор UNION довольно просто. Все, что необходимо сделать, — это указать каждую инструкцию SELECT и вставить между ними ключевое слово UNION.

Рассмотрим пример. Допустим, требуется получить отчет, содержащий сведения обо всех клиентах из штатов Иллинойс, Индиана и Мичиган. Вы также хотите включить в него данные о клиенте Fun4All независимо от штата. Конечно, можно создать условие WHERE, благодаря которому будут выполнены указанные требования, но в данном случае гораздо удобнее прибегнуть к оператору UNION.

Как уже говорилось, применение оператора UNION подразумевает многократное использование инструкций SELECT. Вначале рассмотрим отдельные компоненты комбинированного запроса.

Ввод ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI');
```

Вывод ▼

```
cust_name      cust_contact      cust_email
-----------    -------------     ------------
Village Toys   John Smith        sales@villagetoys.com
Fun4All        Jim Jones         jjones@fun4all.com
The Toy Store  Kim Howard        NULL
```

Ввод ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

Вывод ▼

```
cust_name      cust_contact          cust_email
---------      ------------          ----------
Fun4All        Jim Jones             jjones@fun4all.com
Fun4All        Denise L. Stephens    dstephens@fun4all.com
```

Анализ ▼

Первая инструкция SELECT извлекает все строки, относящиеся к штатам Иллинойс, Индиана и Мичиган, аббревиатуры которых указаны в операторе IN. Вторая инструкция SELECT использует простую проверку на равенство, чтобы найти все вхождения клиента Fun4All.

Чтобы скомбинировать оба запроса, выполните следующее.

Ввод ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

Вывод ▼

```
cust_name      cust_contact          cust_email
---------      ------------          ----------
Fun4All        Denise L.Stephens     dstephens@fun4all.com
Fun4All        Jim Jones             jjones@fun4all.com
Village        Toys John Smith       sales@villagetoys.com
The Toy Store  Kim Howard            NULL
```

Анализ ▼

Данный запрос содержит исходные инструкции SELECT, разделенные ключевым словом UNION. Оно заставляет СУБД выполнить обе инструкции и вывести результаты в виде одного набора результатов.

Для сравнения приведем тот же самый запрос, но использующий не оператор UNION, а несколько предложений WHERE.

Ввод ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
   OR cust_name = 'Fun4All';
```

В данном простом примере применение оператора UNION может показаться более громоздким, чем использование предложения WHERE. Но если условие фильтрации окажется более сложным или понадобится извлекать данные из нескольких таблиц (а не только из одной), то оператор UNION может значительно упростить процесс.

> СОВЕТ: **ограничения оператора UNION**
>
> В стандарте SQL не существует ограничений на число инструкций SELECT, которые могут быть скомбинированы посредством оператора UNION. Однако лучше все же обратиться к документации своей СУБД и убедиться в том, что она не накладывает каких-либо ограничений на максимально допустимое число инструкций.

> ПРЕДУПРЕЖДЕНИЕ: **проблемы, связанные с производительностью**
>
> В большинстве СУБД имеется внутренний оптимизатор запросов, комбинирующий инструкции SELECT, прежде чем СУБД начинает их обработку. Теоретически это означает, что с точки зрения производительности нет реальной разницы между использованием нескольких предложений WHERE и оператора UNION. Мы говорим "теоретически", потому что на практике многие оптимизаторы запросов не всегда выполняют свою работу так хорошо, как следовало бы. Лучше всего протестировать оба метода и посмотреть, какой из них лучше подходит.

Правила применения оператора UNION

Как видите, оператор UNION очень прост в применении. Но существует несколько правил, четко указывающих, что именно может быть объединено.

- Оператор UNION должен включать две или более инструкции SELECT, отделенные одна от другой ключевым словом UNION (таким образом, если в запросе четыре инструкции SELECT, должно быть указано три ключевых слова UNION).
- Каждый запрос в операторе UNION должен содержать одни и те же столбцы, выражения или итоговые функции (кроме того, некоторые СУБД требуют, чтобы столбцы были перечислены в одном и том же порядке).
- Типы данных столбцов должны быть совместимыми. Столбцы не обязательно должны быть одного типа, но они должны быть того типа, который СУБД сможет неявно преобразовать (например, это могут быть различные числовые типы данных или различные типы даты).

При соблюдении этих основных правил и ограничений комбинированные запросы можно применять для решения любых задач по извлечению данных.

Включение или исключение повторяющихся строк

Вернемся к предыдущему примеру и рассмотрим использованные в нем инструкции SELECT. Несложно заметить, что, когда они выполняются по-отдельности, первая инструкция SELECT возвращает три строки, а вторая — две. Но, когда эти две инструкции комбинируются с помощью ключевого слова UNION, возвращаются только четыре строки, а не пять.

Оператор UNION автоматически удаляет все повторяющиеся строки из набора результатов (иными словами, он работает точно так же, как и несколько предложений WHERE в одной инструкции SELECT). В частности, здесь имеется запись о клиенте Fun4All из штата Индиана — эта строка была возвращена обеими инструкциями SELECT. В случае оператора UNION повторяющаяся строка удаляется.

Таково поведение оператора UNION по умолчанию, но при желании его можно изменить. Если требуется, чтобы возвращались все вхождения, необходимо использовать оператор UNION ALL, а не UNION.

Рассмотрим следующий пример.

Ввод ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
UNION ALL
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

Вывод ▼

```
cust_name      cust_contact        cust_email
-----------    ------------        ----------
Village Toys   John Smith          sales@villagetoys.com
Fun4All        Jim Jones           jjones@fun4all.com
The Toy        Store Kim Howard    NULL
Fun4All        Jim Jones           jjones@fun4all.com
Fun4All        Denise L.Stephens   dstephens@fun4all.com
```

Анализ ▼

При использовании оператора UNION ALL СУБД не удаляет дубликаты. Поэтому в данном примере получено пять строк, и одна из них повторяется дважды.

> СОВЕТ: **UNION или WHERE**
>
> В начале урока говорилось о том, что оператор UNION почти всегда выполняет то же самое, что и несколько условий WHERE. Оператор UNION ALL является разновидностью оператора UNION, делая то, что не способны выполнить предложения WHERE. Если вы хотите получить все вхождения для каждого условия (включая дубликаты), используйте оператор UNION ALL, а не предложение WHERE.

Сортировка результатов комбинированных запросов

Результаты запроса SELECT сортируются с помощью предложения ORDER BY. При комбинировании запросов посредством оператора UNION только одно предложение ORDER BY может быть использовано, и оно должно стоять после заключительной инструкции SELECT. Не имеет смысла сортировать часть результатов запроса одним способом, а часть — другим, поэтому применять несколько предложений ORDER BY не разрешается.

В следующем примере сортируются результаты, возвращаемые предыдущим оператором UNION.

Ввод ▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All'
ORDER BY cust_name, cust_contact;
```

Вывод ▼

```
cust_name      cust_contact       cust_email
-----------    ------------       ----------
Fun4All        Denise L.Stephens  dstephens@fun4all.com
Fun4All        Jim Jones          jjones@fun4all.com
The Toy Stor   Kim Howard         NULL
Village Toys   John Smith         sales@villagetoys.com
```

Анализ ▼

В этом операторе UNION используется одно предложение ORDER BY после заключительной инструкции SELECT. Несмотря на то что предложение ORDER BY является частью только последней инструкции SELECT, в действительности СУБД будет применять его для сортировки всех результатов, возвращаемых обеими инструкциями.

ПРИМЕЧАНИЕ: **другие типы комбинированных запросов**

Некоторые СУБД поддерживают два дополнительных типа комбинированных запросов. Оператор EXCEPT (иногда называемый MINUS) может быть использован для извлечения строк, которые существуют только в первой таблице, но не во второй, а оператор INTERSECT можно применять для извлечения только тех строк, которые имеются в обеих таблицах. Однако на практике такие запросы нужны редко, поскольку те же самые результаты могут быть получены посредством объединений.

СОВЕТ: **работа с несколькими таблицами**

Ради простоты в примерах данного урока оператор UNION применялся для объединения запросов к одной и той же таблице. Но на практике этот оператор особенно полезен для объединения данных из нескольких таблиц, в частности таких, которые содержат несовпадающие имена столбцов. В последнем случае можно применить псевдонимы для получения единого набора результатов.

Резюме

На этом уроке вы узнали, как комбинировать инструкции SELECT с помощью оператора UNION. Используя этот оператор, можно вернуть результаты нескольких инструкций в виде одного комбинированного запроса, включающего или исключающего дубликаты. За счет оператора UNION можно значительно упростить сложные предложения WHERE и запросы, связанные с извлечением данных из нескольких таблиц.

УРОК 15

Добавление данных

На этом уроке вы фузнаете, как добавлять данные в таблицы, используя инструкцию INSERT.

Способы добавления данных

Несомненно, SELECT является наиболее часто используемой инструкцией SQL (именно поэтому мы посвятили ее изучению 14 уроков). Но помимо нее в SQL регулярно применяются еще три инструкции, которыми необходимо уметь пользоваться. Первая из них — INSERT. (О двух других мы расскажем на следующем уроке.)

Как следует из названия, инструкция INSERT предназначена для добавления строк в таблицу базы данных. Это можно осуществить несколькими способами:

- добавить одну полную строку;
- добавить часть одной строки;
- добавить результаты запроса.

Далее будут рассмотрены все вышеперечисленные варианты.

> СОВЕТ: **инструкция INSERT и безопасность системы**
>
> Для выполнения инструкции INSERT в клиент-серверной СУБД могут потребоваться особые права доступа. Прежде чем применять инструкцию, убедитесь в том, что у вас есть на это право.

Добавление полных строк

Простейший способ добавления данных в таблицу реализуется с помощью базового синтаксиса инструкции INSERT. Для этого нужно указать имя таблицы и значения, которые должны быть введены в новую строку. Рассмотрим пример.

Ввод ▼

```
INSERT INTO Customers
VALUES('1000000006',
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
       'USA',
       NULL,
       NULL);
```

Анализ ▼

В этом примере в таблицу добавляются сведения о новом клиенте. Данные, которые должны быть сохранены в каждом столбце таблицы, указываются в предложении VALUES. Значения должны быть заданы для каждого столбца. Если для какого-то столбца нет соответствующего значения (как в случае столбцов `cust_contact` и `cust_email` в данном примере), следует указать NULL (предполагается, что таблица допускает отсутствие значений в этих столбцах). Столбцы должны заполняться в порядке, в котором они перечислены в определении таблицы.

> СОВЕТ: **ключевое слово INTO**
>
> В некоторых реализациях SQL ключевое слово INTO после инструкции INSERT является необязательным. Однако хорошей практикой считается указание этого ключевого слова даже в тех случаях, когда этого не требуется. Поступая таким образом, вы обеспечите переносимость кода между разными СУБД.

Показанный синтаксис довольно прост, но не вполне безопасен, поэтому его применения следует всячески избегать. Результаты выполнения вышеприведенной инструкции весьма чувствительны к порядку, в котором столбцы определены в таблице. Необходимо также иметь доступ к определению таблицы. Но даже если в данный момент порядок соблюдается, то нет гарантий, что столбцы будут расположены в том же самом порядке, когда таблица будет редактироваться в следующий раз. Следовательно, использовать инструкцию

SQL, результаты применения которой зависят от порядка следования столбцов, весьма небезопасно. Если вы будете пренебрегать этим советом, вас ждут неприятности.

Безопасный (и, к сожалению, более громоздкий) способ записи инструкции INSERT таков.

Ввод ▼

```
INSERT INTO Customers(cust_id,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country,
                      cust_contact,
                      cust_email)
VALUES('1000000006',
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
       'USA',
       NULL,
       NULL);
```

Анализ ▼

В данном примере делается в точности то же самое, что и в предыдущем случае, но на этот раз имена столбцов явно указаны в круглых скобках после имени таблицы. Когда строка вводится в таблицу, СУБД устанавливает соответствие каждого элемента в списке столбцов с соответствующим значением в списке VALUES. Первое значение в списке VALUES соответствует первому указанному имени столбца, второе значение — второму имени и т.д.

Поскольку имена столбцов перечислены в явном виде, значения, указанные в предложении VALUES, должны соответствовать им в том же самом порядке, причем он не обязательно должен совпадать с порядком столбцов в реальной таблице. Преимущество данного способа таково: даже если расположение столбцов в таблице меняется, инструкция INSERT все равно будет работать корректно.

Следующая инструкция INSERT заполняет все столбцы строки (как и в предыдущем примере), но делает это в другом порядке. Поскольку имена столбцов указываются явно, добавление будет выполнено правильно.

Ввод ▼

```
INSERT INTO Customers(cust_id,
                      cust_contact,
                      cust_email,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
VALUES('1000000006',
       NULL,
       NULL,
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
```

СОВЕТ: **всегда указывайте список столбцов**

Как правило, инструкция INSERT не используется без явного указания списка столбцов. Благодаря этому значительно возрастает вероятность того, что вы сможете успешно выполнить запрос, даже если в таблице произойдут изменения.

ПРЕДУПРЕЖДЕНИЕ: **аккуратно используйте предложение VALUES**

Независимо от синтаксиса инструкции INSERT, значения в предложении VALUES должны быть указаны правильно. Если имена столбцов отсутствуют, должно быть приведено значение для каждого столбца таблицы. Если имена столбцов указываются, должно быть задано значение для каждого столбца, включенного в список. Если что-то пропущено, будет сгенерировано сообщение об ошибке и строка не будет вставлена в таблицу.

Добавление части строки

Рекомендуемый в предыдущем разделе способ использования инструкции INSERT заключается в явном указании имен столбцов таблицы. Применяя такой синтаксис, вы также получаете возможность пропустить определенные столбцы. Это означает, что вы вводите значения для одних столбцов и пропускаете — для других.

Рассмотрим следующий пример.

Ввод ▼

```
INSERT INTO Customers(cust_id,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country)
VALUES('1000000006',
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
       'USA');
```

Анализ ▼

В приведенном ранее примере для двух столбцов — `cust_contact` и `cust_email` — вводились значения NULL. Это означает, что нет причин включать данные столбцы в инструкцию INSERT. Поэтому рассмотренная здесь инструкция INSERT не включает указанные два столбца и два соответствующих им значения.

> ПРЕДУПРЕЖДЕНИЕ: **пропуск столбцов**
>
> Столбцы можно исключать из инструкции INSERT, если это допускается определением таблицы. Должно соблюдаться одно из следующих условий.
>
> - Столбец определен как допускающий значения NULL (отсутствие какого-либо значения).
> - В определении столбца задано значение по умолчанию. Это означает, что, если не указано никакое конкретное значение, будет использовано значение по умолчанию.

ПРЕДУПРЕЖДЕНИЕ: **пропуск обязательных значений**

Если вы пропускаете столбец, для которого не допускаются значения NULL и не заданы значения по умолчанию, СУБД выдаст сообщение об ошибке и строка не будет добавлена.

Добавление результатов запроса

Обычно инструкция INSERT служит для добавления строки в таблицу с использованием явно заданных значений. Существует и другая форма инструкции INSERT, которую можно применять для добавления результатов запроса SELECT. Такая инструкция называется INSERT SELECT и, как подсказывает ее название, выполняет то же самое, что делают инструкции INSERT и SELECT по отдельности.

Предположим, необходимо занести в таблицу Customers список клиентов из другой таблицы. Вместо того чтобы извлекать по одной строке и затем добавлять каждую из них посредством инструкции INSERT, можно сделать следующее.

ПРИМЕЧАНИЕ: **пояснения к следующему примеру**

В следующем примере данные импортируются из таблицы CustNew в таблицу Customers. Сначала создайте и заполните таблицу CustNew. Ее формат должен быть таким же, как и у таблицы Customers, описанной в приложении А. При заполнении таблицы CustNew удостоверьтесь в том, что не используются значения cust_id, которые уже существуют в таблице Customers (последующая операция INSERT потерпит неудачу, если значения первичного ключа будут повторяться).

Ввод ▼

```
INSERT INTO Customers(cust_id,
                      cust_contact,
                      cust_email,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country)
```

```
SELECT cust_id,
       cust_contact,
       cust_email,
       cust_name,
       cust_address,
       cust_city,
       cust_state,
       cust_zip,
       cust_country
FROM CustNew;
```

Анализ ▼

В этом примере для импорта всех данных из таблицы `CustNew` в таблицу `Customers` применяется инструкция `INSERT SELECT`. Вместо того чтобы перечислять значения, которые должны быть добавлены, инструкция `SELECT` извлекает их из таблицы `CustNew`. Каждый столбец в инструкции `SELECT` соответствует столбцу в списке `INSERT`. Сколько же строк добавит эта инструкция? Все зависит от того, сколько строк содержится в таблице `CustNew`. Если таблица пуста, никакие строки добавлены не будут (и никакое сообщение об ошибке не будет выдано, поскольку подобная операция допустима). Если таблица содержит данные, все они будут добавлены в таблицу `Customers`.

> СОВЕТ: **имена столбцов в инструкции `INSERT SELECT`**
>
> В данном примере ради простоты были использованы одинаковые имена столбцов в инструкциях `INSERT` и `SELECT`. Однако это вовсе не обязательно. В действительности СУБД вообще не обращает внимания на имена столбцов, возвращаемых инструкцией `SELECT`. Она учитывает лишь положение столбца, так что первый столбец в инструкции `SELECT` (независимо от имени) будет использован для заполнения первого указанного столбца таблицы и т.д.

Инструкция `SELECT`, используемая в запросе `INSERT SELECT`, может включать предложение `WHERE` для фильтрации данных, которые должны быть добавлены.

СОВЕТ: **добавление нескольких строк**

Инструкция INSERT обычно добавляет только одну строку. Чтобы добавить несколько строк, нужно выполнить несколько инструкций INSERT. Исключением из этого правила является инструкция INSERT SELECT, которая может быть использована для добавления множества строк посредством одного запроса — какие бы данные ни вернула инструкция SELECT, все они будут добавлены в таблицу.

Копирование данных из одной таблицы в другую

Существует другой способ добавления данных, при котором инструкция INSERT вообще не применяется. Чтобы скопировать содержимое какой-то таблицы в новую таблицу (которая создается на лету), можно использовать инструкцию SELECT INTO.

ПРИМЕЧАНИЕ: **не поддерживается в DB2**

DB2 не поддерживает использование инструкции SELECT INTO описанным здесь способом.

В отличие от инструкции INSERT SELECT, посредством которой данные добавляются в уже существующую таблицу, инструкция SELECT INTO копирует данные в новую таблицу (и, в зависимости от СУБД, может перезаписать таблицу, если она уже существует).

ПРИМЕЧАНИЕ: **разница между инструкциями INSERT SELECT и SELECT INTO**

Одно из различий между инструкциями SELECT INTO и INSERT SELECT заключается в том, что первая экспортирует данные, а вторая — импортирует.

В следующем примере демонстрируется применение инструкции SELECT INTO.

Ввод ▼

```
SELECT *
INTO CustCopy
FROM Customers;
```

Анализ ▼

Эта инструкция создает новую таблицу `CustCopy` и копирует в нее все содержимое таблицы `Customers`. Поскольку применяется синтаксис `SELECT *`, каждый столбец таблицы `Customers` будет воссоздан в таблице `CustCopy` (и заполнен соответствующим образом). Чтобы скопировать только часть доступных столбцов, следует явно указать их имена, а не использовать метасимвол `*` (звездочка).

В MariaDB, MySQL, Oracle, PostgreSQL и SQLite поддерживается несколько иной синтаксис.

Ввод ▼

```
CREATE TABLE CustCopy AS
SELECT * FROM Customers;
```

При использовании инструкции `SELECT INTO` нужно обращать внимание на следующие нюансы.

- Разрешается применять любые ключевые слова и предложения инструкции `SELECT`, включая `WHERE` и `GROUP BY`.
- Для добавления данных из нескольких таблиц можно использовать объединения.
- Данные можно добавить только в одну таблицу независимо от того, из скольких таблиц они были извлечены.

> СОВЕТ: **создание копий таблиц**
>
> Инструкция `SELECT INTO` является прекрасным средством создания копий таблиц для экспериментов с новыми для вас инструкциями SQL. Создав копию, вы получите возможность протестировать инструкции SQL на этой копии, а не на таблицах реальной базы данных.

ПРИМЕЧАНИЕ: **дополнительные примеры**

Если хотите увидеть другие примеры использования инструкции INSERT, ознакомьтесь со сценариями заполнения таблиц, описанными в приложении А.

Резюме

На этом уроке вы научились добавлять строки в таблицу базы данных. Вы ознакомились с несколькими способами применения инструкции INSERT и узнали, почему желательно в явном виде указывать имена столбцов. Вы также научились применять инструкцию INSERT SELECT для импорта строк из другой таблицы и инструкцию SELECT INTO — для экспорта строк в новую таблицу. На следующем уроке будет показано, как с помощью инструкций UPDATE и DELETE обновлять и удалять строки.

УРОК 16

Обновление и удаление данных

На этом уроке вы узнаете о том, как применять инструкции UPDATE и DELETE для обновления и удаления записей в таблицах.

Обновление данных

Для обновления (модификации) данных какой-либо таблицы предназначена инструкция UPDATE, которую можно использовать двумя способами:

- обновление определенных строк в таблице;
- обновление всех строк в таблице.

Рассмотрим оба способа.

> ПРЕДУПРЕЖДЕНИЕ: **не забывайте указывать предложение WHERE**
>
> Применять инструкцию UPDATE следует с особой осторожностью, потому что можно по ошибке обновить все строки таблицы. Прочитайте весь раздел, посвященный инструкции UPDATE, прежде чем начинать создавать соответствующие запросы.

> СОВЕТ: **инструкция UPDATE и безопасность системы**
>
> Для выполнения инструкции UPDATE в клиент-серверной СУБД могут потребоваться особые права доступа. Прежде чем применять инструкцию, убедитесь в том, что у вас есть на это право.

Инструкция UPDATE очень проста. Она состоит из трех основных частей:

- имя таблицы, подлежащей обновлению;
- имена столбцов и их новые значения;
- условия фильтрации, определяющие, какие именно строки должны быть обновлены.

Рассмотрим простой пример. Допустим, у клиента 1000000005 появился адрес электронной почты, поэтому его запись нужно обновить. Такое обновление можно выполнить посредством следующей инструкции.

Ввод ▼

```
UPDATE Customers
SET cust_email = 'kim@thetoystore.com'
WHERE cust_id = '1000000005';
```

Инструкция UPDATE всегда начинается с имени таблицы, подлежащей обновлению. В нашем примере это таблица Customers. Затем используется предложение SET, чтобы ввести в столбец новое значение. В данном случае обновляется значение столбца cust_email.

```
SET cust_email = 'kim@thetoystore.com'
```

Заканчивается инструкция UPDATE предложением WHERE, которое сообщает СУБД, какая строка подлежит обновлению. При отсутствии такого предложения СУБД обновила бы все строки таблицы Customers, введя в них новый (причем один и тот же!) адрес электронной почты, а это, конечно же, не то, что нам нужно.

Для обновления нескольких столбцов необходим иной синтаксис.

Ввод ▼

```
UPDATE Customers
SET cust_contact = 'Sam Roberts',
    cust_email = 'sam@toyland.com'
WHERE cust_id = '1000000006';
```

В этом случае используется только одно предложение SET, а каждая пара "столбец–значение" отделяется запятой (после завершающей пары запятая не ставится). В нашем примере оба столбца, cust_contact и cust_email, будут обновлены для клиента 1000000006.

> СОВЕТ: **использование подзапросов в инструкции UPDATE**
>
> В инструкциях UPDATE могут быть использованы подзапросы, что дает возможность обновлять столбцы данными, извлеченными посредством инструкции SELECT (см. урок 11).

> СОВЕТ: **предложение FROM**
>
> Некоторые реализации SQL поддерживают предложение FROM в инструкции UPDATE. Оно может быть использовано для обновления строк одной таблицы данными из другой. Обратитесь к документации своей СУБД и выясните, поддерживает ли она такую возможность.

Чтобы удалить значение из столбца, можно присвоить ему значение NULL (если определение таблицы позволяет вводить в нее значения NULL). Это можно сделать следующим образом.

Ввод ▼

```
UPDATE Customers
SET cust_email = NULL
WHERE cust_id = '1000000005';
```

Здесь ключевое слово NULL используется для удаления значения из столбца cust_email. Это совсем не то же самое, что хранение пустой строки. Сама по себе пустая строка (записывается как '') является значением, тогда как NULL указывает на отсутствие какого-либо значения.

Удаление данных

Для удаления данных из таблицы предназначена инструкция DELETE. Ее можно использовать двумя способами:

- для удаления определенных строк из таблицы;
- для удаления всех строк из таблицы.

Рассмотрим оба способа.

> ПРЕДУПРЕЖДЕНИЕ: **не забывайте указывать предложение WHERE**
>
> Применять инструкцию DELETE следует с особой осторожностью, потому что можно по ошибке удалить все строки таблицы. Прочитайте весь раздел, посвященный инструкции DELETE, прежде чем начинать создавать соответствующие запросы.

> СОВЕТ: **инструкция DELETE и безопасность системы**
>
> Для выполнения инструкции DELETE в клиент-серверной СУБД могут потребоваться особые права доступа. Прежде чем применять инструкцию, убедитесь в том, что у вас есть на это право.

Выше уже говорилось о том, что инструкция UPDATE очень проста. К счастью, инструкция DELETE еще проще.

Следующая инструкция удаляет одну строку из таблицы Customers.

Ввод ▼

```
DELETE FROM Customers
WHERE cust_id = '1000000006';
```

Предложение DELETE FROM требует указания имени таблицы, из которой должны быть удалены данные. Предложение WHERE фильтрует строки, определяя, какие из них должны быть удалены. В нашем примере должна быть удалена строка, относящаяся к клиенту 1000000006. Если бы предложение WHERE было пропущено, инструкция удалила бы все строки из таблицы.

СОВЕТ: **научитесь работать с внешними ключами**

На уроке 12 вы ознакомились с концепцией объединений и узнали, что в объединяемых таблицах должны быть общие поля. Но можно заставить СУБД создавать принудительные связи между таблицами с помощью внешних ключей (примеры их определений можно найти в приложении А). Когда имеются внешние ключи, СУБД использует их, чтобы гарантировать ссылочную целостность. Например, если попытаться добавить новый товар в таблицу `Products`, указав неизвестный идентификатор поставщика, СУБД не позволит этого сделать, поскольку столбец `vend_id` является внешним ключом для таблицы `Vendors`. Вы спросите, какое отношение все это имеет к инструкции `DELETE`? Положительным побочным эффектом использования внешних ключей для поддержания ссылочной целостности является то, что СУБД обычно запрещает удаление строк, которые обеспечивают корректность отношений между таблицами. Например, если попытаться удалить из таблицы `Products` товар, указанный в существующих заказах в таблице `OrderItems`, инструкция `DELETE` завершится неудачей и будет выдано сообщение об ошибке. Это веская причина для того, чтобы всегда определять внешние ключи.

СОВЕТ: **ключевое слово FROM**

В некоторых реализациях SQL ключевое слово `FROM` после инструкции `DELETE` является необязательным. Однако хорошей практикой считается указание этого ключевого слова даже в тех случаях, когда этого не требуется. Поступая таким образом, вы обеспечите переносимость кода между разными СУБД.

Инструкция `DELETE` не принимает имена столбцов или метасимволы. Она удаляет строки целиком, а не отдельные столбцы. Для удаления конкретного столбца следует использовать инструкцию `UPDATE`.

ПРИМЕЧАНИЕ: **содержимое таблиц, но не сами таблицы**

Инструкция `DELETE` удаляет из таблицы отдельные строки или даже все строки за один раз, но она никогда не удаляет саму таблицу.

> СОВЕТ: **более быстрое удаление**
>
> Если необходимо удалить все строки из таблицы, не используйте инструкцию DELETE. Для этого существует инструкция TRUNCATE TABLE, которая делает то же самое, но гораздо быстрее (потому что изменения данных не регистрируются в журнале СУБД).

Советы по обновлению и удалению данных

Все инструкции UPDATE и DELETE, рассмотренные в предыдущих разделах, сопровождались предложениями WHERE, и на то есть веская причина. Если пропустить предложение WHERE, инструкция будет применена по отношению ко всем строкам таблицы. Другими словами, если выполнить инструкцию UPDATE без предложения WHERE, каждая строка таблицы будет заменена новыми значениями. Аналогичным образом, если выполнить инструкцию DELETE без предложения WHERE, будет удалено все содержимое таблицы.

Ниже даны рекомендации, которым следуют большинство разработчиков SQL.

- Никогда не выполняйте инструкцию UPDATE или DELETE без предложения WHERE, если только на самом деле не хотите обновить или удалить каждую строку таблицы.
- Убедитесь в том, что каждая таблица имеет первичный ключ (см. урок 12, если забыли, что это такое), и используйте его в предложении WHERE всякий раз, когда это возможно. (Можно указывать отдельные первичные ключи, несколько значений или диапазоны значений.)
- Прежде чем использовать предложение WHERE с инструкцией UPDATE или DELETE, сначала проверьте его с инструкцией SELECT, дабы убедиться в том, что оно правильно фильтрует записи. Можно ошибиться и записать неправильное условие.
- Используйте внешние ключи, чтобы СУБД не позволяла удалять строки, для которых в других таблицах имеются связанные с ними данные.
- Некоторые СУБД позволяют администраторам баз данных устанавливать ограничения, препятствующие выполнению

инструкций UPDATE или DELETE без предложения WHERE. Если ваша СУБД поддерживает такую возможность, не пренебрегайте ею.

Помните о том, что в SQL нет кнопки отмены. Будьте очень внимательны, выполняя инструкции UPDATE и DELETE, иначе можно вдруг обнаружить, что удалены или обновлены не те данные.

Резюме

На этом уроке вы узнали, как использовать инструкции UPDATE и DELETE для обновления и удаления табличных данных. Вы ознакомились с синтаксисом каждой из этих инструкций, а также с опасностями, которыми чревато их применение. Вы также узнали, почему столь важно указывать предложение WHERE в инструкциях UPDATE и DELETE, и изучили основные правила, которым нужно следовать, чтобы по неосторожности не повредить данные.

УРОК 17

Создание таблиц и работа с ними

На этом уроке вы ознакомитесь с основными правилами создания, изменения и удаления таблиц.

Создание таблиц

SQL применяется не только для работы с табличными данными, но и для выполнения всех операций с базами данных, включая создание и модификацию таблиц.

Существуют два способа создания таблиц.

- Большинство СУБД содержат инструменты администрирования, которые можно применять для интерактивного создания таблиц и управления ими.
- С таблицами можно также работать посредством инструкций SQL.

Для создания таблиц программным способом предназначена инструкция `CREATE TABLE`. Стоит отметить, что, когда вы применяете интерактивный инструментарий, в действительности вся работа выполняется инструкциями SQL. Но вам не приходится писать эти инструкции, поскольку СУБД создает и выполняет их незаметно для вас (то же самое справедливо и для процедуры изменения существующих таблиц).

> ПРЕДУПРЕЖДЕНИЕ: **различия в синтаксисе**
>
> Точный синтаксис инструкции CREATE TABLE может немного отличаться в различных реализациях SQL. Обязательно обратитесь к документации своей СУБД за дополнительной информацией и выясните, какой в точности синтаксис необходим для нее и какие возможности она поддерживает.

Полное рассмотрение всех параметров, применяемых при создании таблиц, не входит в задачи данного урока. Мы рассмотрим только основы. За дополнительной информацией обратитесь к документации своей СУБД.

> ПРИМЕЧАНИЕ: **примеры для конкретных СУБД**
>
> Инструкции CREATE TABLE для конкретных СУБД приведены в приложении А.

Создание простой таблицы

Чтобы создать таблицу с помощью инструкции CREATE TABLE, нужно указать следующие данные:

- имя новой таблицы, которое задается после ключевых слов CREATE TABLE;
- имена и определения столбцов таблицы, разделенные запятыми;
- в некоторых СУБД также требуется, чтобы было задано расположение таблицы.

Следующая инструкция создает таблицу Products, часто используемую в книге.

Ввод ▼

```
CREATE TABLE Products
(
   prod_id       CHAR(10)        NOT NULL,
   vend_id       CHAR(10)        NOT NULL,
   prod_name     CHAR(254)       NOT NULL,
```

```
    prod_price    DECIMAL(8,2)     NOT NULL,
    prod_desc     VARCHAR(1000)    NULL
);
```

Анализ ▼

Как видите, имя таблицы указывается сразу же после ключевых слов CREATE TABLE. Определение таблицы (все ее столбцы) заключается в круглые скобки. Определения столбцов разделяются запятыми. Приведенная в данном примере таблица состоит из пяти столбцов. Определение каждого столбца начинается с имени столбца (которое должно быть уникальным в пределах данной таблицы), а за ним указывается тип данных. (Обратитесь к уроку 1, чтобы вспомнить, что такое типы данных. Кроме того, в приложении Г приведен перечень основных типов данных в SQL.) Инструкция в целом заканчивается точкой с запятой, которая стоит после закрывающей круглой скобки.

Ранее уже говорилось, что синтаксис инструкции CREATE TABLE зависит от СУБД, и данный пример наглядно доказывает это. В Oracle, PostgreSQL, SQL Server и SQLite инструкция будет работать в приведенной форме, а вот в MySQL тип VARCHAR должен быть заменен типом text. В DB2 значение NULL должно быть удалено из последнего столбца. Именно поэтому пришлось использовать различные сценарии создания таблиц для каждой СУБД (как объясняется в приложении А).

> СОВЕТ: **форматирование инструкции**
>
> Помните о том, что пробелы игнорируются инструкциями SQL. Инструкцию можно ввести в одной длинной строке или разбить ее на несколько строк; разницы между ними не будет. Это позволяет форматировать листинги SQL так, как вам удобно. Показанная выше инструкция CREATE TABLE — хороший пример форматирования SQL-запроса. Код разбит на несколько строк, а определения столбцов выровнены пробелами для удобства чтения и редактирования. Форматировать инструкции SQL подобным образом не обязательно, но все же настоятельно рекомендуется.

СОВЕТ: **замена существующих таблиц**

Когда вы создаете новую таблицу, указываемое вами имя не должно существовать в СУБД, иначе будет выдано сообщение об ошибке. Чтобы избежать случайной перезаписи, SQL требует, чтобы вы вначале вручную удалили таблицу (подробности описаны далее), а затем вновь создали ее, а не просто перезаписали.

Работа со значениями NULL

На уроке 4 рассказывалось о том, что такое значение NULL. Оно подразумевает, что в столбце не содержится никакого значения. Столбец, в котором допускаются значения NULL, позволяет добавлять в таблицу строки, в которых не предусмотрено значение для данного столбца. Столбец, в котором не допускаются значения NULL, не принимает строки с отсутствующим значением. Другими словами, для этого столбца всегда потребуется вводить какое-то значение при добавлении или обновлении строк.

Каждый столбец таблицы может быть либо пустым (NULL), либо не пустым (NOT NULL), и это его состояние оговаривается в определении таблицы на этапе ее создания. Рассмотрим следующий пример.

Ввод ▼

```
CREATE TABLE Orders
(
   order_num     INTEGER     NOT NULL,
   order_date    DATETIME    NOT NULL,
   cust_id       CHAR(10)    NOT NULL
);
```

Анализ ▼

Посредством этой инструкции создается таблица Orders, неоднократно использованная в книге. Таблица состоит из трех столбцов: номер и дата заказа, а также идентификатор клиента. Все три столбца являются необходимыми, и каждый из них содержит спецификацию NOT NULL, которая будет препятствовать добавлению в таблицу

столбцов с отсутствующим значением. При попытке добавления такого столбца будет выдано сообщение об ошибке, и добавить неполную запись не удастся.

В следующем примере создается таблица, в которой могут быть столбцы обоих типов, NULL и NOT NULL.

Ввод ▼

```
CREATE TABLE Vendors
(
   vend_id          CHAR(10)   NOT NULL,
   vend_name        CHAR(50)   NOT NULL,
   vend_address     CHAR(50)   ,
   vend_city        CHAR(50)   ,
   vend_state       CHAR(5)    ,
   vend_zip         CHAR(10)   ,
   vend_country     CHAR(50)
);
```

Анализ ▼

Посредством этой инструкции создается таблица Vendors, также неоднократно использованная в книге. Столбцы с идентификатором и именем поставщика необходимы, поэтому оба определены как NOT NULL (т.е. не допускающие значений NULL). Пять остальных столбцов допускают значения NULL, поэтому для них не указана спецификация NOT NULL. Значение NULL принято по умолчанию, и в отсутствие спецификации NOT NULL предполагается, что значения NULL допустимы.

> ПРЕДУПРЕЖДЕНИЕ: **указание значения NULL**
>
> Во многих СУБД отсутствие спецификации NOT NULL трактуется как NULL. Однако не во всех. В DB2 наличие ключевого слова NULL является обязательным, и если оно не указано, генерируется сообщение об ошибке. Обратитесь к документации своей СУБД, чтобы получить исчерпывающую информацию о синтаксисе инструкции.

СОВЕТ: **первичные ключи и значения NULL**

На уроке 1 говорилось о том, что первичные ключи — это столбцы, значения которых уникально идентифицируют каждую строку таблицы. Столбцы, допускающие отсутствие значений, не могут использоваться в качестве уникальных идентификаторов.

ПРЕДУПРЕЖДЕНИЕ: **что такое NULL**

Не путайте значения NULL с пустыми строками. NULL означает отсутствие значения; это не пустая строка. Если указать в коде '' (две одинарных кавычки, между которыми ничего нет), это значение можно будет ввести в столбец типа NOT NULL. Пустая строка является допустимым значением; она не означает отсутствие значения. Значения NULL задаются только посредством ключевого слова NULL, но не пустыми строками.

Определение значений по умолчанию

SQL позволяет определять значения по умолчанию, которые будут использованы в том случае, если при добавлении строки значение одного из полей не указано. Значения по умолчанию задаются с помощью ключевого слова DEFAULT в определениях столбцов в инструкции CREATE TABLE.

Рассмотрим следующий пример.

Ввод ▼

```
CREATE TABLE OrderItems
(
   order_num     INTEGER        NOT NULL,
   order_item    INTEGER        NOT NULL,
   prod_id       CHAR(10)       NOT NULL,
   quantity      INTEGER        NOT NULL   DEFAULT 1,
   item_price    DECIMAL(8,2)   NOT NULL
);
```

Анализ ▼

Посредством этой инструкции создается таблица OrderItems, содержащая отдельные элементы заказов (сам заказ хранится в

таблице Orders). Столбец quantity содержит количество каждого товара в заказе. В данном примере добавление спецификации DEFAULT 1 в описание столбца заставляет СУБД подразумевать количество, равное 1, если не указано иное.

Значения по умолчанию часто используются для хранения даты и времени. К примеру, системная дата может быть назначена как дата по умолчанию путем указания функции или переменной, используемой для ссылки на системную дату. В частности, пользователи MySQL могут указать дату как DEFAULT CURRENT_DATE(), в то время как пользователям Oracle нужно вводить дату как DEFAULT SYSDATE, а пользователям SQL Server — как DEFAULT GETDATE(). К сожалению, команда, используемая для получения системной даты, в каждой СУБД своя. В табл. 17.1 приведен синтаксис для нескольких СУБД (если ваша СУБД не представлена в этом списке, обратитесь к ее документации).

ТАБЛИЦА 17.1. **Получение системной даты**

СУБД	Функция/переменная
Access	NOW()
DB2	CURRENT_DATE
MySQL	CURRENT_DATE()
Oracle	SYSDATE
PostgreSQL	CURRENT_DATE
SQL Server	GETDATE()
SQLite	date('now')

> СОВЕТ: **использование значений DEFAULT вместо NULL**
>
> Многие разработчики баз данных применяют значения DEFAULT вместо столбцов NULL, особенно в столбцах, которые будут использованы в вычислениях или при группировке строк.

Обновление таблиц

Для того чтобы обновить определение таблицы, следует воспользоваться инструкцией ALTER TABLE. Несмотря на то что все СУБД поддерживают эту инструкцию, ее возможности в значительной степени зависят от конкретной СУБД. Ниже приведен ряд соображений по поводу применения инструкции ALTER TABLE.

- В идеальном случае структура таблицы вообще не должна меняться после того, как в таблицу введены данные. В процессе разработки таблиц потратьте время на анализ будущих потребностей пользователей, чтобы позже не пришлось вносить в структуру таблиц существенные изменения.
- Все СУБД позволяют добавлять столбцы в уже существующие таблицы, но некоторые СУБД ограничивают типы данных, которые могут быть добавлены (а заодно и правила использования значений NULL и DEFAULT).
- Многие СУБД не позволяют удалять или изменять столбцы в таблице.
- Большинство СУБД разрешает переименовывать столбцы.
- Многие СУБД налагают серьезные ограничения на изменения, которым могут подвергнуться заполненные столбцы, и значительно меньшие ограничения — в случае незаполненных столбцов.

Как видите, вносить изменения в существующие таблицы ничуть не проще, чем создавать их заново. Обратитесь к документации своей СУБД, чтобы уточнить, что именно можно изменять.

Чтобы изменить таблицу посредством инструкции ALTER TABLE, нужно задать следующую информацию:

- имя таблицы, подлежащей изменению (таблица с таким именем должна существовать, иначе будет выдано сообщение об ошибке);
- список изменений, которые должны быть сделаны.

Поскольку добавление столбцов в таблицу — единственная операция, поддерживаемая всеми СУБД, именно ее мы и рассмотрим в качестве примера.

Ввод ▼

```
ALTER TABLE Vendors
ADD vend_phone CHAR(20);
```

Анализ ▼

Посредством этой инструкции в таблицу Vendors добавляется столбец vend_phone. Должен быть указан тип данных столбца. .

Другие операции, такие как изменение или удаление столбцов, задание ограничений или ключей, требуют похожего синтаксиса. (Отметим, что следующий пример будет работать уже не во всех СУБД.)

Ввод ▼

```
ALTER TABLE Vendors
DROP COLUMN vend_phone;
```

Сложные изменения структуры таблицы обычно выполняются вручную и включают следующие этапы.

1. Создание новой таблицы с новым расположением столбцов.
2. Использование инструкции INSERT SELECT (см. урок 15) для копирования данных из старой таблицы в новую. В случае необходимости задействуются функции преобразования и вычисляемые поля.
3. Проверка того факта, что новая таблица содержит нужные данные.
4. Переименование старой таблицы (или удаление ее).
5. Присвоение имени новой таблице, которое ранее принадлежало старой таблице.
6. Восстановление триггеров, хранимых процедур, индексов и внешних ключей, если это необходимо.

> ПРИМЕЧАНИЕ: **инструкция ALTER TABLE в SQLite**
>
> SQLite ограничивает перечень операций, которые можно выполнять с помощью инструкции ALTER TABLE. Одно из наиболее важных ограничений заключается в том, что в этой СУБД нельзя применять данную инструкцию для изменения первичных и внешних ключей. Они должны указываться только в начальной инструкции CREATE TABLE.

ПРЕДУПРЕЖДЕНИЕ: **аккуратно используйте инструкцию ALTER TABLE**

Инструкцию ALTER TABLE следует использовать с особой осторожностью. Прежде чем выполнять данный запрос, убедитесь в том, что у вас есть полный комплект резервных копий (схемы и самих данных). Внесение изменений в таблицу базы данных нельзя отменить. Если вы добавите в нее ненужные столбцы, у вас не будет возможности удалить их. Аналогично, если вы удалите столбец, который вам на самом деле нужен, то рискуете потерять все данные, содержавшиеся в нем.

Удаление таблиц

Удаление таблиц (имеется в виду удаление самих таблиц, а не их содержимого) — очень простой процесс. Таблицы удаляются с помощью инструкции DROP TABLE.

Ввод ▼

```
DROP TABLE CustCopy;
```

Анализ ▼

Эта инструкция удаляет таблицу CustCopy (которую мы создали на уроке 15). В данном случае не требуется никакого подтверждения, и невозможно вернуться к прежнему состоянию — в результате применения инструкции DROP TABLE таблица будет безвозвратно удалена.

СОВЕТ: **использование реляционных правил для предотвращения ошибочного удаления**

Во многих СУБД применяются правила, препятствующие удалению таблиц, связанных с другими таблицами. Если эти правила действуют и вы применяете инструкцию DROP TABLE по отношению к таблице, которая связана с другой таблицей, СУБД заблокирует операцию до тех пор, пока не будет удалена данная связь. Такое поведение приветствуется, поскольку благодаря ему можно воспрепятствовать ошибочному удалению нужных таблиц.

Переименование таблиц

В разных СУБД переименование таблиц осуществляется поразному. Не существует жестких, устоявшихся стандартов на выполнение этой операции. Пользователи DB2, MariaDB, MySQL, Oracle и PostgreSQL могут применять инструкцию RENAME. Пользователям SQL Server доступна хранимая процедура sp_rename. SQLite поддерживает переименование таблиц посредством инструкции ALTER TABLE.

Базовый синтаксис для всех операций переименования требует указания старого и нового имен. Однако существуют различия, зависящие от реализации. Обратитесь к документации своей СУБД, чтобы узнать детали относительно поддерживаемого ею синтаксиса.

Резюме

На этом уроке вы ознакомились с несколькими новыми инструкциями SQL. Инструкция CREATE TABLE предназначена для создания новых таблиц, ALTER TABLE — для изменения столбцов таблицы (или других объектов, таких как ограничения или индексы), а инструкция DROP TABLE позволяет полностью удалить таблицу. Все эти инструкции нужно использовать с особой осторожностью и только после создания резервных копий базы данных. Поскольку точный синтаксис этих инструкций варьируется в зависимости от СУБД, вам придется обратиться к документации своей СУБД за дополнительной информацией.

УРОК 18

Представления

На этом уроке рассказывается о том, что такое представления, как они работают и зачем они нужны. Вы также узнаете, как использовать представления для упрощения некоторых SQL-запросов, изученных нами на предыдущих уроках.

Что такое представления

Представления — это виртуальные таблицы. В отличие от таблиц, содержащих данные, представления содержат запросы, которые динамически извлекают данные, когда это необходимо.

> ПРИМЕЧАНИЕ: **поддержка в разных СУБД**
>
> Microsoft Access поддерживает представления, но не так, как это принято в стандартных реализациях SQL. Таким образом, примеры данного урока не применимы к Access.
>
> Поддержка представлений была добавлена в MySQL версии 5. В более ранних версиях СУБД примеры данного урока работать не будут.
>
> SQLite поддерживает представления, доступные только для чтения. Их можно создавать и просматривать, но их содержимое нельзя обновлять.

Лучший способ объяснить, что такое представления, — рассмотреть конкретный пример. Вернемся к уроку 12, на котором была создана следующая инструкция SELECT для извлечения данных сразу из трех таблиц.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
  AND OrderItems.order_num = Orders.order_num
  AND prod_id = 'RGAN01';
```

Этот запрос позволяет извлечь информацию о клиентах, которые заказали указанный товар. Любой пользователь, которому необходимы такие данные, должен был бы разобраться в структуре таблицы, а также в методике объединения таблиц. Чтобы извлечь аналогичные данные для другого товара (или для нескольких товаров), последнее условие в предложении WHERE придется модифицировать.

Теперь предположим, что весь этот запрос можно сохранить в виде виртуальной таблицы ProductCustomers. Тогда для получения тех же самых данных достаточно было бы сделать следующее.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM ProductCustomers
WHERE prod_id = 'RGAN01';
```

Это как раз тот случай, когда нужны представления. Таблица ProductCustomers является представлением, поскольку она не содержит каких-либо столбцов или данных. Вместо них хранится запрос — тот самый, который был использован выше для объединения таблиц.

> СОВЕТ: **согласованность СУБД**
>
> Синтаксис создания представлений одинаков во всех основных СУБД.

Зачем нужны представления

Выше был рассмотрен один из случаев использования представления. Довольно часто они применяются для выполнения следующих операций.

- Повторное использование инструкций SQL.
- Упрощение сложных запросов. После того как запрос подготовлен, его можно с легкостью использовать повторно, и для этого не придется разбираться в нюансах его работы.
- Вывод фрагментов таблицы вместо всей таблицы.
- Защита данных. Пользователям можно предоставить доступ к определенному подмножеству таблиц, а не ко всем таблицам.
- Изменение форматирования и способа отображения данных. Представления могут возвращать данные, отформатированные и отображаемые не так, как они хранятся в таблицах.

Созданные представления можно использовать точно так же, как и таблицы. Можно выполнять инструкции SELECT по отношению к ним, фильтровать и сортировать в них данные, объединять представления с другими представлениями или таблицами и, возможно, даже добавлять в них данные или обновлять их. (На последнюю операцию налагаются определенные ограничения. Об этом будет рассказано далее.)

Важно не забывать о том, что представления — это виртуальные таблицы, данные которых хранятся в других таблицах. Представления не содержат данных как таковых, поэтому строки, которые они возвращают, извлекаются из других таблиц. Если данные этих таблиц изменяются, представления обновятся автоматически.

> ПРЕДУПРЕЖДЕНИЕ: **проблемы производительности**
>
> Поскольку представления не содержат данных, каждый раз, когда происходит обращение к ним, для выполнения запроса приходится проводить определенный поиск. Если вы создали сложное представление с несколькими объединениями и фильтрами или если были задействованы вложенные представления, производительность СУБД резко снизится. Рекомендуется провести тестирование, прежде чем создавать приложения, в которых интенсивно используются представления.

Правила и ограничения представлений

Прежде чем создавать представления, следует узнать о связанных с ними ограничениях. К сожалению, представления весьма специфичны для каждой СУБД, поэтому обязательно обратитесь к документации своей СУБД.

Ниже приведено несколько самых общих правил и ограничений, которыми следует руководствоваться при создании и использовании представлений.

- Представления, как и таблицы, должны иметь уникальные имена. (Они не могут быть названы так же, как другие таблицы или представления.)
- Не существует ограничения на количество представлений, которые могут быть созданы.
- Для того чтобы создать представление, необходимо иметь соответствующие права доступа. Обычно их предоставляет администратор базы данных.
- Представления могут быть вложенными. Это означает, что представление может быть создано посредством запроса, который извлекает данные из другого представления. Точное количество уровней вложения зависит от СУБД. (Вложенные представления могут серьезно снизить производительность при выполнении запроса, поэтому их нужно основательно протестировать, прежде чем применять на практике.)
- Во многих СУБД запрещается использование предложения `ORDER BY` в запросах к представлениям.
- В некоторых СУБД требуется, чтобы каждый возвращаемый столбец обладал именем, — это подразумевает использование псевдонимов, если столбцы представляют собой вычисляемые поля. (См. урок 7, где рассказывалось о псевдонимах столбцов.)
- Представления нельзя индексировать. Они также не могут иметь триггеров или связанных с ними значений по умолчанию.
- В некоторых СУБД представления трактуются как запросы, предназначенные только для чтения. Это означает, что из представлений можно извлекать данные, но их нельзя заносить в таблицы, на основе которых было создано представление. Обратитесь к документации своей СУБД, чтобы узнать детали.
- Некоторые СУБД позволяют создавать представления, которые не разрешают добавлять или обновлять строки, если это может привести к тому, что строки уже не будут являться частью данного представления. Например, если представление возвращает только информацию о клиентах, имеющих адреса электронной почты, обновление информации о клиенте с

целью удаления его адреса электронной почты приведет к тому, что данный клиент будет исключен из представления. Таково поведение по умолчанию, и оно допускается, но некоторые СУБД способны препятствовать возникновению подобных случаев.

> СОВЕТ: **обратитесь к документации своей СУБД**
>
> Список этих правил довольно длинный, и документация вашей СУБД почти наверняка содержит еще какие-то правила. Придется потратить некоторое время на изучение подобных ограничений, прежде чем браться за создание представлений.

Создание представлений

Итак, вы знаете, что такое представления (и какими правилами следует руководствоваться при работе с ними). Теперь разберемся, как они создаются.

Представления создаются с помощью инструкции `CREATE VIEW`. Аналогично инструкции `CREATE TABLE`, данную инструкцию можно использовать только для создания представления, которого прежде не существовало.

> ПРИМЕЧАНИЕ: **удаление представлений**
>
> Для удаления представления предназначена инструкция `DROP VIEW`. Ее синтаксис прост:
>
> ```
> DROP VIEW имя_представления;
> ```
>
> Чтобы перезаписать (или обновить) представление, вначале нужно применить по отношению к нему инструкцию `DROP VIEW`, а потом заново создать представление.

Использование представлений для упрощения сложных объединений

Чаще всего представления используются для упрощения сложных запросов, и нередко это относится к объединениям. Рассмотрим следующий пример.

Ввод ▼

```
CREATE VIEW ProductCustomers AS
SELECT cust_name, cust_contact, prod_id
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
  AND OrderItems.order_num = Orders.order_num;
```

Анализ ▼

Посредством этой инструкции создается представление `ProductCustomers`, которое объединяет три таблицы для получения списка клиентов, заказавших какой-нибудь товар. Если затем выполнить инструкцию `SELECT * FROM ProductCustomers`, она вернет список всех клиентов, сделавших заказы.

Для получения списка клиентов, заказавших товар `RGAN01`, необходимо выполнить следующее.

Ввод ▼

```
SELECT cust_name, cust_contact
FROM ProductCustomers
WHERE prod_id = 'RGAN01';
```

Вывод ▼

```
cust_name          cust_contact
-----------------  ----------------
Fun4All            Denise L. Stephens
The Toy Store      Kim Howard
```

Анализ ▼

Эта инструкция извлекает указанные данные из представления благодаря предложению `WHERE`. Когда СУБД обрабатывает такой запрос, она добавляет указанное условие к любому уже существующему предложению `WHERE` в запросе самого представления, благодаря чему данные фильтруются правильно.

Таким образом, представления могут значительно упростить сложные инструкции SQL. Используя представления, можно один раз записать код SQL и затем повторно применять его, когда возникает такая необходимость.

СОВЕТ: **создание повторно используемых представлений**

Хорошей идеей является создание представлений, не привязанных к конкретным данным. Например, представление, созданное в предыдущем примере, возвращает имена клиентов, заказавших все товары, а не только товар RGAN01 (для которого представление первоначально и создавалось). Расширение диапазона представления позволяет многократно использовать его, и тем самым устраняется необходимость в создании и хранении множества похожих представлений. Это делает такое представление еще более эффективным.

Использование представлений для переформатирования извлекаемых данных

Как уже говорилось ранее, другим распространенным случаем использования представлений является переформатирование извлекаемых данных. Следующая инструкция SELECT (см. урок 7) возвращает имя поставщика и его местонахождение в одном комбинированном вычисляемом столбце.

Ввод ▼

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Вывод ▼

```
vend_title
----------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

Показанная ниже инструкция аналогична вышеприведенной, но в ней применяется оператор || (его мы также рассматривали на уроке 7).

Ввод ▼

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Вывод ▼

```
vend_title
----------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

Теперь предположим, что результаты регулярно требуются в таком формате. Вместо того чтобы выполнять конкатенацию всякий раз, когда в этом возникает необходимость, можно создать представление и использовать его вместо объединения. Для превращения этой инструкции в представление нужно поступить следующим образом.

Ввод ▼

```
CREATE VIEW VendorLocations AS
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
       AS vend_title
FROM Vendors;
```

А вот синтаксис с использованием оператора ||.

Ввод ▼

```
CREATE VIEW VendorLocations AS
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
       AS vend_title
FROM Vendors;
```

Анализ ▼

Посредством этой инструкции создается представление, использующее в точности тот же самый запрос, что и в предыдущей инструкции SELECT. Чтобы извлечь данные, необходимые для создания почтовых наклеек, выполните следующее.

Ввод ▼

```
SELECT *
FROM VendorLocations;
```

Вывод ▼

```
vend_title
----------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

> ПРИМЕЧАНИЕ: **все ограничения инструкции SELECT сохраняются**
>
> Ранее уже говорилось о том, что синтаксис создания представлений достаточно согласован в разных СУБД. Но откуда тогда так много версий синтаксиса инструкций? Дело в том, что представление просто скрывает инструкцию SELECT, синтаксис которой должен четко соответствовать правилам и ограничениям, принятым в используемой СУБД.

Использование представлений для фильтрации нежелательных данных

Представления могут также оказаться полезными для применения распространенных условий WHERE. Например, вам может понадобиться определить представление CustomerEMailList таким образом, чтобы оно отфильтровывало клиентов, не имеющих адресов электронной почты. Для этого необходимо создать следующую инструкцию.

Ввод ▼

```
CREATE VIEW CustomerEMailList AS
SELECT cust_id, cust_name, cust_email
FROM Customers
WHERE cust_email IS NOT NULL;
```

Анализ ▼

Очевидно, отправляя сообщение в список рассылки, следовало бы пропустить клиентов, у которых нет адреса электронной почты. В данном случае предложение WHERE отфильтровывает строки, имеющие значения NULL в столбце cust_email, так что соответствующие записи не будут извлекаться.

Теперь представление CustomerEMailList можно использовать подобно любой другой таблице.

Ввод ▼

```
SELECT *
FROM CustomerEMailList;
```

Вывод ▼

```
cust_id      cust_name      cust_email
----------   ------------   --------------------
1000000001   Village Toys   sales@villagetoys.com
1000000003   Fun4All        jjones@fun4all.com
1000000004   Fun4All        dstephens@fun4all.com
```

> ПРИМЕЧАНИЕ: **предложения WHERE**
>
> Если предложение WHERE используется при извлечении данных из представления, два набора условий (одно в самом представлении и другое, передаваемое ему) будут скомбинированы автоматически.

Использование представлений с вычисляемыми полями

Представления чрезвычайно полезны для упрощения запросов с вычисляемыми полями. Ниже приведена инструкция SELECT,

впервые использованная нами на уроке 7. Она извлекает элементы указанного заказа и вычисляет суммарную стоимость для каждого элемента.

Ввод ▼

```
SELECT prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM OrderItems
WHERE order_num = 20008;
```

Вывод ▼

```
prod_id     quantity      item_price     expanded_price
-------     --------      ----------     --------------
RGAN01      5             4.9900         24.9500
BR03        5             11.9900        59.9500
BNBG01      10            3.4900         34.9000
BNBG02      10            3.4900         34.9000
BNBG03      10            3.4900         34.9000
```

Для превращения его в представление необходимо выполнить следующее.

Ввод ▼

```
CREATE VIEW OrderItemsExpanded AS
SELECT order_num,
       prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM OrderItems;
```

Чтобы получить информацию о заказе 20008 (она была выведена выше), необходимо выполнить следующий запрос.

Ввод ▼

```
SELECT *
FROM OrderItemsExpanded
WHERE order_num = 20008;
```

Вывод ▼

```
order_num  prod_id  quantity  item_price  expanded_price
---------  -------  --------  ----------  --------------
20008      RGAN01    5         4.99        24.95
20008      BR03      5         11.99       59.95
20008      BNBG01    10        3.49        34.90
20008      BNBG02    10        3.49        34.90
20008      BNBG03    10        3.49        34.90
```

Как видите, представления легко создавать, а использовать — еще легче. В эффективно спроектированной базе данных представления могут существенно упростить сложные запросы.

Резюме

Представления — это виртуальные таблицы. Вместо самих данных они содержат запросы, посредством которых данные извлекаются в случае необходимости. Представления обеспечивают должный уровень инкапсуляции инструкций SELECT и могут быть использованы для упрощения работы с данными, а также для переформатирования данных и ограничения доступа к ним.

УРОК 19

Хранимые процедуры

На этом уроке вы узнаете, что такое хранимые процедуры, для чего и как они применяются. Вы также ознакомитесь с базовым синтаксисом, используемым при создании и запуске хранимых процедур.

Что такое хранимые процедуры

Большинство запросов SQL, которые мы до сих пор создавали, просты в том смысле, что в них применяется только одна инструкция по отношению к одной или нескольким таблицам. Но не все запросы столь просты — зачастую приходится использовать несколько инструкций для выполнения сложного запроса. Рассмотрим, к примеру, следующий сценарий.

- При обработке заказа необходимо удостовериться в том, что соответствующие товары есть на складе.
- Если товары есть на складе, они должны быть зарезервированы, чтобы их не продали кому-то другому, а их количество, доступное другим покупателям, должно быть уменьшено соответственно сделанному заказу.
- Товары, отсутствующие на складе, должны быть запрошены у поставщика.
- Клиенту необходимо сообщить, какие товары есть на складе (и могут быть отгружены немедленно), а какие — запрашиваются под заказ.

Очевидно, это не полный список действий, но суть должна быть ясна. Решение подобной задачи потребует применения многих инструкций SQL по отношению ко многим таблицам. Помимо того что сами инструкции, подлежащие выполнению в подобных случаях, и их порядок не постоянны, они могут (и будут) изменяться в зависимости от того, какие товары имеются на складе, а каких там нет.

Как бы вы написали такой код? Можно было бы записать каждую из инструкций SQL отдельно и выполнять инструкции в зависимости от полученных результатов. Вам пришлось бы делать это каждый раз, когда возникала бы необходимость в подобной обработке данных (и для каждого приложения, которое в ней нуждается).

Альтернативный вариант — создать хранимую процедуру. Это набор из нескольких инструкций, сохраненный для последующего выполнения. Хранимую процедуру можно рассматривать как командный файл, хотя это нечто большее.

> ПРИМЕЧАНИЕ: **поддержка в различных СУБД**
>
> Хранимые процедуры не поддерживаются в Access и SQLite. Таким образом, примеры данного урока не применимы в этих СУБД.
>
> Поддержка хранимых процедур была добавлена в MySQL версии 5. В более ранних версиях СУБД примеры урока работать не будут.

> ПРИМЕЧАНИЕ: **гораздо больше информации**
>
> Хранимые процедуры — тема довольно сложная, и полностью рассмотреть ее можно только в отдельной книге. Данный урок не научит вас всему, что необходимо знать о хранимых процедурах. Скорее это введение в данную тему, призванное ознакомить вас с тем, что собой представляют хранимые процедуры и что с их помощью можно делать. По существу, представленные здесь примеры соответствуют только синтаксису Oracle и SQL Server.

Зачем нужны хранимые процедуры

Теперь, когда вы знаете, что такое хранимые процедуры, возникает другой вопрос: для чего их применять? На то существует множество причин, ниже перечислены лишь основные.

- Для упрощения сложных запросов (как уже говорилось выше) за счет инкапсуляции инструкций в один блок, удобный для выполнения.

- Для обеспечения непротиворечивости данных за счет того, что не требуется снова и снова воспроизводить одну и ту же последовательность инструкций. Если все разработчики и приложения используют одни и те же хранимые процедуры, значит, будет выполняться один и тот же код.

  Следствием этого является предотвращение ошибок. Чем больше действий необходимо выполнить, тем выше вероятность появления ошибок. Отсутствие ошибок обеспечивает целостность данных.
- Для упрощения управления изменениями. Если таблицы, имена столбцов, деловые правила (или что-то подобное) изменяются, обновлять приходится только код хранимой процедуры и ничего больше.

  Следствием этого является повышение безопасности. Предоставление доступа к основным данным только через хранимые процедуры снижает вероятность повреждения данных (случайного или преднамеренного).
- Поскольку хранимые процедуры обычно хранятся в скомпилированном виде, СУБД тратит меньше времени на их обработку. Это приводит к повышению производительности.
- Некоторые возможности SQL реализуются только в одиночных запросах. Хранимые процедуры можно применять для написания более гибкого и мощного кода.

Итак, имеются три основных преимущества: простота, безопасность и производительность. Очевидно, все они чрезвычайно важны. Однако, прежде чем бросаться превращать весь свой SQL-код в хранимые процедуры, следует узнать и о другой стороне медали.

- Синтаксис хранимых процедур сильно зависит от СУБД. Написать по-настоящему переносимый код хранимой процедуры практически невозможно. В то же время сами вызовы хранимых процедур (их имена и способы передачи аргументов) могут быть достаточно переносимыми, поэтому, если вам необходимо перейти на другую СУБД, по крайней мере код клиентского приложения, возможно, не придется менять.
- Хранимые процедуры сложнее в написании, чем основные инструкции SQL, и их подготовка требует большей квалификации и опыта. Поэтому многие администраторы баз данных ограничивают права на создание хранимых процедур в качестве меры безопасности.

Несмотря на вышесказанное, хранимые процедуры весьма полезны и непременно должны применяться. В действительности многие СУБД располагают всевозможными хранимыми процедурами, которые предназначены для управления базами данных и таблицами. Обратитесь к документации своей СУБД, чтобы получить больше информации об этом.

> ПРИМЕЧАНИЕ: **не можете написать хранимые процедуры? Тогда просто используйте их**
>
> Во многих СУБД различаются меры безопасности и права доступа, необходимые для написания хранимых процедур и для их выполнения. И это хорошо. Если вы не намерены писать собственные хранимые процедуры, используйте готовые.

Выполнение хранимых процедур

Хранимые процедуры выполняются намного чаще, чем пишутся, поэтому мы начнем именно с их выполнения. Инструкция SQL для запуска хранимой процедуры — EXECUTE — принимает имя хранимой процедуры и передаваемые ей аргументы. Рассмотрим следующий пример.

Ввод ▼

```
EXECUTE AddNewProduct('JTS01',
                      'Stuffed Eiffel Tower',
                      6.49,
                      'Plush stuffed toy with the text
↳ La Tour Eiffel in red white and blue')
```

Анализ ▼

Здесь выполняется хранимая процедура AddNewProduct, которая добавляет новый товар в таблицу Products. Процедура принимает четыре аргумента: идентификатор поставщика (первичный ключ таблицы Vendors), название товара, цена и описание. Эти четыре параметра соответствуют четырем ожидаемым переменным хранимой процедуры (определенным в ней). Данная процедура добавляет новую строку в таблицу Products и распределяет полученные аргументы по соответствующим столбцам.

В таблице Products есть еще один столбец, нуждающийся в присвоении значения: prod_id, который является первичным ключом таблицы. Почему это значение не передается в хранимую процедуру в виде аргумента? Для того чтобы идентификаторы генерировались правильно, безопаснее сделать подобный процесс автоматизированным (не полагаясь на конечного пользователя). Именно поэтому хранимая процедура используется в данном примере. Она выполняет следующие действия:

- подтверждает правильность передаваемых данных, обеспечивая наличие значений у всех четырех аргументов;
- генерирует уникальный идентификатор, который будет использован в качестве первичного ключа;
- добавляет данные о новом товаре в таблицу Products, сохраняя созданный первичный ключ и занося данные в соответствующие столбцы.

Таков основной способ выполнения хранимой процедуры. В зависимости от СУБД могут быть доступны и другие варианты выполнения, включая следующие.

- Опциональные аргументы со значениями по умолчанию, присваиваемыми в случае, если аргумент не задан пользователем.
- Нестандартные параметры, указываемые в виде пар *параметр=значение*.
- Выходные параметры, позволяющие хранимой процедуре обновлять переменную, используемую вызывающим приложением.
- Данные, извлекаемые инструкцией SELECT.
- Возвращаемые коды, позволяющие хранимой процедуре передавать значение вызывающему приложению.

Создание хранимых процедур

Как уже говорилось, создание хранимой процедуры — задача не из тривиальных. Чтобы продемонстрировать это, рассмотрим простой пример: хранимую процедуру, которая подсчитывает в списке рассылки число клиентов, имеющих адрес электронной почты.

Ниже приведена версия для Oracle.

Ввод ▼

```
CREATE PROCEDURE MailingListCount (
   ListCount OUT INTEGER
)
IS
   v_rows INTEGER;
BEGIN
   SELECT COUNT(*) INTO v_rows
   FROM Customers
   WHERE NOT cust_email IS NULL;
   ListCount := v_rows;
END;
```

Анализ ▼

Эта хранимая процедура принимает один аргумент — `ListCount`. Вместо того чтобы передавать значение в хранимую процедуру, данный аргумент возвращает значение из нее. Ключевое слово `OUT` определяет подобное поведение аргумента. Oracle поддерживает аргументы типов `IN` (передаются в хранимые процедуры), `OUT` (передаются из хранимых процедур) и `INOUT` (передаются в обоих направлениях). Собственно код хранимой процедуры заключен между ключевыми словами `BEGIN` и `END`. Здесь для определения клиентов, имеющих адреса электронной почты, выполняется простая инструкция `SELECT`. После этого выходному аргументу `ListCount` присваивается значение, равное количеству строк в выборке.

Для запуска примера в Oracle необходимо выполнить следующий код.

Ввод ▼

```
var ReturnValue NUMBER
EXEC MailingListCount(:ReturnValue);
SELECT ReturnValue;
```

Анализ ▼

В этом коде объявляется переменная, которая будет хранить значение, возвращаемое процедурой. После этого запускается сама процедура и выполняется инструкция `SELECT` для отображения полученного значения.

Ниже приведена версия для Microsoft SQL Server.

Ввод ▼

```
CREATE PROCEDURE MailingListCount
AS
DECLARE @cnt INTEGER
SELECT @cnt = COUNT(*)
FROM Customers
WHERE NOT cust_email IS NULL;
RETURN @cnt;
```

Анализ ▼

Эта хранимая процедура вообще не принимает никаких аргументов. Вызывающее приложение получает нужное значение благодаря тому, что в SQL Server поддерживаются возвращаемые значения. Здесь посредством инструкции DECLARE объявлена локальная переменная @cnt (имена всех локальных переменных в SQL Server начинаются с символа @). Эта переменная затем используется в инструкции SELECT, принимая значение, возвращаемое функцией COUNT(*). Наконец, инструкция RETURN используется для передачи результатов подсчета в вызывающее приложение.

Для запуска примера в SQL Server необходимо выполнить следующий код.

Ввод ▼

```
DECLARE @ReturnValue INT
EXECUTE @ReturnValue=MailingListCount;
SELECT @ReturnValue;
```

Анализ ▼

В этом коде объявляется переменная, которая будет хранить значение, возвращаемое процедурой. Затем запускается сама процедура и выполняется инструкция SELECT для отображения полученного значения.

Приведем еще один пример, но на этот раз будем добавлять новый заказ в таблицу Orders. Данный пример подходит только для SQL Server, однако он хорошо иллюстрирует, как применять хранимые процедуры.

Ввод ▼

```
CREATE PROCEDURE NewOrder @cust_id CHAR(10)
AS
-- Объявление переменной для номера заказа
DECLARE @order_num INTEGER
-- Получение текущего наибольшего номера заказа
SELECT @order_num=MAX(order_num)
FROM Orders
-- Определение следующего номера заказа
SELECT @order_num=@order_num+1
-- Добавление нового заказа
INSERT INTO Orders(order_num, order_date, cust_id)
VALUES(@order_num, GETDATE(), @cust_id)
-- Возвращение номера заказа
RETURN @order_num;
```

Анализ ▼

Эта хранимая процедура создает новый заказ в таблице `Orders` и принимает один аргумент: идентификатор клиента, сделавшего заказ. Два других столбца таблицы — номер и дата заказа — генерируются автоматически в самой хранимой процедуре. Вначале в коде объявляется локальная переменная для хранения номера заказа. Затем запрашивается текущий наибольший номер заказа (посредством функции `MAX()`), который увеличивается на единицу (с помощью инструкции `SELECT`). После этого посредством инструкции `INSERT` добавляется новый заказ с использованием только что сгенерированного номера заказа, текущей системной даты (определяется с помощью функции `GETDATE()`) и полученного идентификатора клиента. Наконец, номер заказа (необходимый для обработки элементов заказа) возвращается с помощью инструкции `RETURN @order_num`. Обратите внимание на то, что код снабжен комментариями. Это всегда следует делать при написании хранимых процедур.

Ниже приведена несколько иная версия того же кода для SQL Server.

Ввод ▼

```
CREATE PROCEDURE NewOrder @cust_id CHAR(10)
AS
-- Добавление нового заказа
INSERT INTO Orders(cust_id)
VALUES(@cust_id)
-- Возвращение номера заказа
SELECT order_num = @@IDENTITY;
```

> СОВЕТ: **комментируйте свой код**
>
> Любой код должен быть снабжен комментариями, и хранимая процедура — не исключение. Добавление комментариев не окажет никакого влияния на производительность, так что здесь нет "обратной стороны медали" (время тратится только на написание комментариев). А преимущества очевидны, например облегчение понимания кода другими программистами (да и вами тоже), а также удобство его изменения спустя некоторое время.
>
> Как отмечалось на уроке 2, самый распространенный способ комментирования кода — поставить в начале строки символы -- (два дефиса). Некоторые СУБД поддерживают и альтернативный синтаксис комментариев, но синтаксис -- универсален, поэтому лучше использовать его.

Анализ ▼

Данная хранимая процедура также создает новый заказ в таблице `Orders`. Но на этот раз СУБД сама генерирует номер заказа. Большинство СУБД поддерживают такой тип функциональности (подобные столбцы называются полями автонумерации). Опять же, процедуре передается только один аргумент: идентификатор клиента, сделавшего заказ. Номер и дата заказа не указываются вообще — СУБД использует значение по умолчанию для даты (функция `GETDATE()`), а номер заказа генерируется автоматически. Как узнать, какой идентификатор был сгенерирован? В SQL Serer для этого предназначена глобальная переменная `@@IDENTITY`, возвращаемая в вызывающее приложение (на этот раз с использованием инструкции `SELECT`).

Как видите, хранимые процедуры очень часто позволяют решить одну и ту же задачу разными способами. Выбор во многом будет зависеть от особенностей СУБД, с которой вы работаете.

Резюме

На этом уроке вы узнали, что такое хранимые процедуры и для чего они нужны. Вы также ознакомились с базовым синтаксисом, применяемым для создания и выполнения хранимых процедур, и узнали о способах их применения. В каждой СУБД хранимые процедуры реализуются по-разному. Не исключено, что в вашей СУБД способ выполнения хранимых процедур будет несколько иным, и вы получите возможности, не упомянутые в данной книге. За дополнительной информацией обратитесь к документации своей СУБД.

УРОК 20

Обработка транзакций

На этом уроке вы узнаете, что такое транзакции и как применять инструкции `COMMIT` *и* `ROLLBACK` *для их обработки.*

Что такое транзакции

Обработка транзакций обеспечивает сохранение целостности базы данных за счет того, что пакеты SQL-запросов выполняются полностью или не выполняются вовсе.

Как объяснялось на уроке 12, реляционные базы данных организованы таким образом, что информация в них хранится во многих таблицах. Благодаря этому облегчается управление данными, а также их повторное использование. Не вдаваясь в подробности, почему реляционные базы данных устроены именно так, следует отметить, что схемы всех удачно спроектированных баз данных можно в какой-то степени отнести к реляционным.

Хороший пример — таблицы заказов, с которыми мы работали на последних 18-ти уроках. Заказы хранятся в двух таблицах: `Orders` содержит описание самих заказов, а `OrderItems` — информацию об отдельных элементах заказов. Эти две таблицы связаны между собой с помощью уникальных идентификаторов, которые называются *первичными ключами* (см. урок 1). Кроме того, эти таблицы связаны и с другими таблицами, содержащими информацию о клиентах и товарах.

1. Процесс добавления нового заказа заключается в выполнении следующих действий.
2. Проверка, содержится ли информация о клиенте в базе данных. Если нет, такая информация добавляется.
3. Получение идентификатора клиента.
4. Добавление строки в таблицу `Orders` и связывание ее с идентификатором клиента.
5. Получение идентификатора нового заказа, присвоенного ему в таблице `Orders`.

Добавление одной строки в таблицу `OrderItems` для каждого заказанного товара и соотнесение его с таблицей `Orders` посредством полученного идентификатора заказа (а также с таблицей `Products` посредством идентификатора товара).

Теперь предположим, что какая-то ошибка в базе данных (например, нехватка места на диске, ограничения, связанные с безопасностью, блокировка таблицы) помешала завершить эту последовательность действий. Что случится с данными?

Хорошо, если ошибка произойдет после добавления информации о клиенте в таблицу, но до того, как она будет добавлена в таблицу `Orders`, — в таком случае проблем не возникнет. Разрешается хранить данные о клиентах без заказов. При повторном выполнении приведенной выше последовательности действий добавленная запись о клиенте будет возвращена и использована. Вы сможете продолжить работу с того места, на котором остановились.

Но что если ошибка произойдет после того, как была добавлена строка в таблицу `Orders`, но до того, как будут добавлены строки в таблицу `OrderItems`? Теперь в базе данных будет присутствовать пустой заказ.

Или еще более плохой сценарий: что если система сделает ошибку в процессе добавления строк в таблицу `OrderItems`? В таком случае заказ будет внесен в базу данных лишь частично, и вы даже не будете знать об этом.

Как решить эту проблему? Именно здесь в игру вступают *транзакции*. Обработка транзакций — это механизм, применяемый для управления наборами SQL-запросов, которые должны быть выполнены только целиком, т.е. таким образом, чтобы в базу данных не могли попасть результаты частичного выполнения запросов. При обработке транзакций можно быть уверенным в том, что выполнение набора запросов не было прервано посередине — они или были выполнены все, или не был выполнен ни один из них. Если никаких ошибок не произошло, результаты работы всего набора фиксируются (записываются) в таблицах базы данных. Если произошла ошибка, должны быть отменены все операции, чтобы вернуть базу данных в прежнее согласованное состояние.

Итак, если обратиться к нашему примеру, то вот как на самом деле должен выглядеть процесс.

1. Проверка, содержится ли информация о клиенте в базе данных. Если нет, такая информация добавляется.

2. Фиксация информации о клиенте.
3. Получение идентификатора клиента.
4. Добавление строки в таблицу `Orders`.
5. Если во время добавления строки в таблицу `Orders` происходит ошибка, операция отменяется.
6. Получение идентификатора нового заказа, присвоенного ему в таблице `Orders`
7. Добавление одной строки в таблицу `OrderItems` для каждого заказанного товара.
8. Если в процессе добавления строк в таблицу `OrderItems` происходит ошибка, добавление всех строк в таблицу `OrderItems` отменяется.

При работе с транзакциями вы часто будете сталкиваться с одними и теми же терминами.

- **Транзакция**. Единый набор SQL-запросов.
- **Откат**. Процесс отмены указанных инструкций SQL.
- **Фиксация**. Запись несохраненных инструкций SQL в таблицы базы данных.
- **Точка сохранения**. Временное состояние в ходе выполнения транзакции, в которое можно вернуться после отмены части инструкций набора (в отличие от отмены всей транзакции).

> СОВЕТ: **действие каких инструкций можно отменить?**
>
> Обработка транзакций задействуется в ходе выполнения инструкций `INSERT`, `UPDATE` и `DELETE`. Нельзя отменить действие инструкции `SELECT` (в этом нет смысла.) Нельзя также отменить запросы `CREATE` или `DROP`. Их можно задействовать в транзакциях, но если понадобится выполнить откат, действие этих инструкций аннулировано не будет.

Управление транзакциями

Теперь, когда вы знаете, что такое обработка транзакций, перейдем к управлению транзакциями.

> ПРЕДУПРЕЖДЕНИЕ: **различия в реализациях**
>
> Точный синтаксис, используемый для обработки транзакций, зависит от СУБД. Прежде чем применять описываемые далее инструкции, обратитесь к документации своей СУБД.

Ключ к управлению транзакциями заключается в том, чтобы сгруппировать SQL-запросы в логические блоки и явно указать, когда может быть выполнен откат, а когда — нет.

В некоторых СУБД требуется, чтобы пользователь явно пометил начало и конец каждой транзакции. Например, в SQL Server нужно сделать следующее.

Ввод ▼

```
BEGIN TRANSACTION
...
COMMIT TRANSACTION
```

Анализ ▼

В этом примере все инструкции, заключенные между фразами `BEGIN TRANSACTION` и `COMMIT TRANSACTION`, должны быть или выполнены, или не выполнены.

Эквивалентный код для MariaDB и MySQL приведен ниже.

Ввод ▼

```
START TRANSACTION
...
```

В Oracle синтаксис будет таким.

Ввод ▼

```
SET TRANSACTION
...
```

Ввод ▼

В PostgreSQL используется синтаксис ANSI SQL.

Ввод ▼

```
BEGIN;
...
```

В других СУБД применяются схожие варианты синтаксиса. Вы заметите, что в большинстве реализаций не требуется явного завершения транзакции. Вместо этого транзакция продолжается до тех пор, пока что-то не прервет ее. Обычно это либо инструкция COMMIT для сохранения изменений, либо инструкция ROLLBACK для их отмены.

Инструкция ROLLBACK

Инструкция ROLLBACK предназначена для отката (отмены) SQL-запросов, как показано ниже.

Ввод ▼

```
DELETE FROM Orders;
ROLLBACK;
```

Анализ ▼

В этом примере выполняется и сразу же, посредством инструкции ROLLBACK, аннулируется запрос DELETE. Пусть это и не самый полезный пример, он все равно показывает, что, будучи включенными в транзакцию, операции DELETE (а также INSERT и UPDATE) не являются окончательными.

Инструкция COMMIT

Обычно после выполнения инструкций SQL результаты записываются непосредственно в таблицы баз данных. Это называется *неявная фиксация* — операция сохранения (или записи) выполняется автоматически.

Однако в транзакции неявная фиксация может и не применяться. Это зависит от того, с какой СУБД вы работаете. Некоторые СУБД трактуют завершение транзакции как неявную фиксацию.

Для принудительной фиксации изменений предназначена инструкция COMMIT. Вот соответствующий пример для SQL Server.

Ввод ▼

```
BEGIN TRANSACTION
DELETE OrderItems WHERE order_num = 12345
DELETE Orders WHERE order_num = 12345
COMMIT TRANSACTION
```

Анализ ▼

В этом примере заказ номер 12345 полностью удаляется из базы данных. Поскольку это приводит к обновлению двух таблиц, Orders и OrderItems, транзакция применяется для того, чтобы не допустить частичного удаления заказа. Конечная инструкция COMMIT фиксирует изменения только в том случае, если не произошло никаких ошибок. Если первая инструкция будет выполнена, а вторая из-за ошибки — нет, удаление не будет зафиксировано.

Чтобы выполнить то же самое в Oracle, воспользуйтесь следующим кодом.

Ввод ▼

```
SET TRANSACTION
DELETE OrderItems WHERE order_num = 12345;
DELETE Orders WHERE order_num = 12345;
COMMIT;
```

Точки сохранения

Простые инструкции COMMIT и ROLLBACK позволяют фиксировать или отменять транзакции в целом. Это вполне оправданно по отношению к коротким транзакциям, но для более сложных могут понадобиться частичные фиксации или откаты.

Например, описанный выше процесс добавления заказа представляет собой одну транзакцию. Если произойдет ошибка, необходимо вернуться в состояние, когда строка еще не была добавлена в таблицу Orders. Но вы вряд ли захотите отменить добавление данных в таблицу Customers (если оно было сделано).

Для отмены части транзакции нужно иметь возможность размещения меток в стратегически важных точках блока инструкций. Тогда, если понадобится сделать частичный откат, вы сможете вернуть базу данных в состояние, соответствующее одной из меток.

В SQL подобные метки называются *точками сохранения*. Для создания такой точки в MariaDB, MySQL и Oracle применяется инструкция SAVEPOINT.

Ввод ▼

```
SAVEPOINT delete1;
```

В SQL Server нужно сделать следующее.

Ввод ▼

```
SAVE TRANSACTION delete1;
```

Каждая точка сохранения должна обладать уникальным именем, однозначно идентифицирующим ее, чтобы, когда вы выполняете откат, СУБД "знала", в какую точку она должна вернуться. Для отмены всех инструкций после этой точки в SQL Server нужно выполнить следующее.

Ввод ▼

```
ROLLBACK TRANSACTION delete1;
```

В MariaDB, MySQL и Oracle необходимо поступить так.

Ввод ▼

```
ROLLBACK TO delete1;
```

А вот полный пример для SQL Server.

Ввод ▼

```
BEGIN TRANSACTION
INSERT INTO Customers(cust_id, cust_name)
VALUES('1000000010', 'Toys Emporium');
SAVE TRANSACTION StartOrder;
INSERT INTO Orders(order_num, order_date, cust_id)
VALUES(20100,'2001/12/1','1000000010');
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
INSERT INTO OrderItems(order_num, order_item, prod_id,
```

```
↳quantity, item_price)
VALUES(20100, 1, 'BR01', 100, 5.49);
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
INSERT INTO OrderItems(order_num, order_item, prod_id,
↳quantity, item_price)
VALUES(20100, 2, 'BR03', 100, 10.99);
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
COMMIT TRANSACTION
```

Анализ ▼

Здесь выполняется набор из четырех инструкций INSERT, объединенных в транзакцию. Точка сохранения определена после первой инструкции INSERT, так что если один из последующих запросов INSERT закончится неудачей, отмена транзакции произойдет лишь до этой точки. В SQL Server для контроля успешности запроса можно использовать системную переменную @@ERROR. (В других СУБД применяются иные функции или переменные.) Если переменная @@ERROR содержит ненулевое значение, значит, произошла ошибка и транзакция отменяется до точки сохранения. Если транзакция в целом завершается успешно, для сохранения данных выполняется инструкция COMMIT.

> СОВЕТ: **чем больше точек сохранения, тем лучше**
>
> Можно создавать сколько угодно точек сохранения в SQL-коде, и чем больше, тем лучше. Почему? Потому что чем больше у вас точек сохранения, тем более гибко можно управлять откатами.

Резюме

Транзакции представляют собой блоки инструкций SQL, которые должны выполняться в пакетном режиме (все вместе). Вы ознакомились с инструкциями COMMIT и ROLLBACK, которые предназначены для явного управления процессами записи и отмены результатов транзакций. Вы также узнали, как применять точки сохранения для обеспечения более гибкого контроля за отменой запросов. Разумеется, обработка транзакций — очень обширная тема, которую невозможно охватить за один урок. Кроме того, механизмы обработки транзакций реализованы по-разному в каждой СУБД. Поэтому обратитесь к документации своей СУБД за дополнительной информацией.

УРОК 21

Курсоры

На этом уроке вы узнаете, что такое курсоры и как их применять.

Что такое курсоры

SQL-запросы, связанные с извлечением данных, работают с наборами строк, которые называются *результирующими*. Все возвращаемые строки соответствуют условию отбора, указанному в инструкции SQL; их может быть ноль или больше. При использовании простых инструкций SELECT невозможно получить первую, следующую строку или предыдущие десять строк. Это объясняется особенностями функционирования реляционной СУБД.

> **Результирующий набор**
> Результаты, возвращаемые SQL-запросом.

Но иногда бывает необходимо просмотреть строки в прямом или обратном порядке по одной или по нескольку строк за раз. Именно для этого и нужны курсоры. *Курсор* представляет собой запрос к базе данных, хранящийся на сервере СУБД, — это не инструкция SELECT, но результирующий набор, выборка, полученная в результате выполнения инструкции SELECT. После того как курсор сохранен, приложения могут "прокручивать" (просматривать) строки в прямом или обратном порядке, когда возникает такая необходимость.

Различные СУБД поддерживают разные параметры курсоров. Чаще всего предоставляются следующие возможности.

- Возможность помечать курсор как предназначенный только для чтения, в результате чего данные можно считывать, но нельзя обновлять или удалять.
- Возможность задавать направление выполняемых операций (вперед, назад, первая, последняя, абсолютное положение, относительное положение и т.п.).

ПРИМЕЧАНИЕ: **поддержка в различных СУБД**

В Microsoft Access не поддерживаются курсоры. Таким образом, приводимые далее примеры не применимы в этой СУБД.

Поддержка курсоров была добавлена в MySQL версии 5. В более ранних версиях СУБД примеры данного урока работать не будут. В SQLite поддерживается разновидность курсоров, называемая *шагами*. Основные концепции, рассматриваемые на этом уроке, применимы к шагам, но синтаксис будет совершенно иным.

- Возможность помечать одни столбцы как редактируемые, а другие — как нередактируемые.
- Указание области видимости, благодаря чему курсор может быть доступен только для запроса, посредством которого он был создан (например, для хранимой процедуры), или для всех запросов.
- Указание СУБД создать копию полученных данных (в противоположность работе с "живыми" данными в таблицах), чтобы они не изменялись в промежуток времени между открытием курсора и обращением к нему.

Курсоры используются главным образом интерактивными приложениями, которые позволяют пользователям прокручивать отображаемые на экране записи вперед и назад, просматривать их или изменять.

ПРИМЕЧАНИЕ: **курсоры и веб-приложения**

Курсоры практически бесполезны в веб-приложениях (написанных, к примеру, на ASP, ASP.NET, ColdFusion, PHP, Python, Ruby и JSP). Курсоры предназначены для использования в течение сеанса связи между клиентским приложением и сервером, но такая модель не годится для веб-приложений, потому что сервер приложений является клиентом базы данных, а не конечным пользователем. А раз так, то большинство разработчиков приложений избегают использования курсоров и реализуют нужную функциональность самостоятельно в случае необходимости.

Работа с курсорами

Работу с курсором можно разделить на несколько этапов.

- Прежде чем курсор может быть использован, его следует объявить (определить). В ходе этого процесса не происходит извлечение данных, а просто определяется соответствующая инструкция SELECT и задаются параметры курсора.
- После объявления курсор нужно открыть для получения данных. В ходе этого процесса уже происходит извлечение строк согласно предварительно заданной инструкции SELECT.
- После того как курсор заполнен данными, из него могут быть извлечены необходимые строки.
- По окончании работы курсор должен быть закрыт, и, возможно, должны быть освобождены занятые им ресурсы (в зависимости от СУБД).

После того как курсор объявлен, его можно открывать и закрывать столько раз, сколько необходимо. Если курсор открыт, извлекать из него строки можно произвольное число раз.

Создание курсоров

Курсоры создаются с помощью инструкции DECLARE, синтаксис которой зависит от СУБД. Инструкция DECLARE присваивает курсору имя и определяет инструкцию SELECT, дополненную по необходимости предложением WHERE и другими. Чтобы показать, как это работает, мы создадим курсор, который будет извлекать список всех клиентов, не имеющих адресов электронной почты. Такой курсор является частью приложения, позволяющего менеджеру вводить недостающие адреса.

Приведенная ниже версия подходит для DB2, MariaDB, MySQL и SQL Server.

Ввод ▼

```
DECLARE CustCursor CURSOR
FOR
SELECT * FROM Customers
WHERE cust_email IS NULL
```

А вот версия для Oracle и PostgreSQL.

Ввод ▼

```
DECLARE CURSOR CustCursor
IS
SELECT * FROM Customers
WHERE cust_email IS NULL
```

Анализ ▼

В обеих версиях для указания имени курсора применяется инструкция DECLARE — в данном случае это будет имя CustCursor. Инструкция SELECT определяет курсор, содержащий имена всех клиентов, у которых нет адреса электронной почты (соответствующее значение равно NULL).

Теперь, после того как курсор определен, его можно открыть.

Управление курсорами

Курсоры открываются с помощью инструкции OPEN CURSOR, синтаксис которой настолько прост, что его поддерживает большинство СУБД.

Ввод ▼

```
OPEN CURSOR CustCursor
```

При обработке инструкции OPEN CURSOR выполняется заданный запрос, и полученные строки сохраняются для последующего просмотра.

Доступ к содержимому курсора можно получить с помощью инструкции FETCH. Она задает, какие строки должны быть извлечены, откуда они должны быть извлечены и где их следует сохранить (имя переменной, например). В первом примере применяется синтаксис Oracle для извлечения одной строки курсора (первой).

Ввод ▼

```
DECLARE TYPE CustCursor IS REF CURSOR
   RETURN Customers%ROWTYPE;
DECLARE CustRecord Customers%ROWTYPE
BEGIN
```

```
   OPEN CustCursor;
   FETCH CustCursor INTO CustRecord;
   CLOSE CustCursor;
END;
```

Анализ ▼

В данном примере инструкция FETCH извлекает текущую строку (считывание автоматически начинается с первой строки) и записывает ее в переменную CustRecord. С полученными данными ничего не делается.

В следующем примере (в нем вновь применяется синтаксис Oracle) полученные данные подвергаются циклической обработке от первой строки до последней.

Ввод ▼

```
DECLARE TYPE CustCursor IS REF CURSOR
   RETURN Customers%ROWTYPE;
DECLARE CustRecord Customers%ROWTYPE
BEGIN
   OPEN CustCursor;
   LOOP
   FETCH CustCursor INTO CustRecord;
   EXIT WHEN CustCursor%NOTFOUND;
      ...
   END LOOP;
   CLOSE CustCursor;
END;
```

Анализ ▼

Как и в предыдущем примере, здесь используется инструкция FETCH для записи текущей строки в переменную CustRecord. Однако в данном случае инструкция FETCH находится в цикле LOOP, поэтому она выполняется снова и снова. Строка EXIT WHEN CustCursor%NOTFOUND означает, что цикл должен быть завершен, когда больше не останется строк для извлечения. Сам код обработки здесь не показан. В реальном примере необходимо заменить ... собственным кодом.

Рассмотрим другой пример, на этот раз с использованием синтаксиса Microsoft SQL Server.

Ввод ▼

```
DECLARE @cust_id CHAR(10),
        @cust_name CHAR(50),
        @cust_address CHAR(50),
        @cust_city CHAR(50),
        @cust_state CHAR(5),
        @cust_zip CHAR(10),
        @cust_country CHAR(50),
        @cust_contact CHAR(50),
        @cust_email CHAR(255),
OPEN CustCursor
FETCH NEXT FROM CustCursor
   INTO @cust_id, @cust_name, @cust_address,
        @cust_city, @cust_state, @cust_zip,
        @cust_country, @cust_contact, @cust_email
WHILE @@FETCH_STATUS = 0
BEGIN
...
FETCH NEXT FROM CustCursor
   INTO @cust_id, @cust_name, @cust_address,
        @cust_city, @cust_state, @cust_zip,
        @cust_country, @cust_contact, @cust_email
END
CLOSE CustCursor
```

Анализ ▼

В данном примере переменные объявляются для каждого извлекаемого столбца, а инструкции FETCH осуществляют выборку строк и сохраняют их значения в этих переменных. Цикл WHILE нужен для последовательной обработки каждой строки, а условие WHILE @@FETCH_STATUS = 0 обеспечивает завершение обработки (выход из цикла) после того, как все строки будут извлечены. Сам код обработки здесь тоже не показан. В реальном примере нужно заменить ... собственным кодом.

Закрытие курсоров

Как было показано в предыдущих примерах, по окончании работы с курсорами их нужно закрывать. Кроме того, в некоторых СУБД (например, в SQL Server) требуется, чтобы ресурсы, занятые курсором, были освобождены явным образом. Соответствующий синтаксис для DB2, Oracle и PostgreSQL приведен ниже.

Ввод ▼

```
CLOSE CustCursor
```

А вот синтаксис для Microsoft SQL Server.

Ввод ▼

```
CLOSE CustCursor
DEALLOCATE CURSOR CustCursor
```

Для закрытия курсора предназначена инструкция `CLOSE`. После того как курсор закрыт, к нему нельзя обратиться, не открыв перед этим вновь. Однако его не нужно объявлять заново при повторном использовании, достаточно лишь выполнить инструкцию `OPEN`.

Резюме

На этом уроке вы узнали, что такое курсоры и как их применять. В вашей СУБД, возможно, поддерживается несколько иной синтаксис, а также доступны параметры, не упомянутые в книге. За дополнительной информацией обратитесь к документации своей СУБД.

УРОК 22

Расширенные возможности SQL

На этом уроке мы рассмотрим несколько расширенных средств обработки данных в SQL: ограничения, индексы и триггеры.

Что такое ограничения

SQL прошел целый ряд этапов развития, прежде чем стать полноценным и мощным языком. В итоге он обогатился эффективными инструментами обработки данных, в том числе такими, как ограничения.

Мы уже неоднократно говорили о реляционных таблицах и ссылочной целостности на предыдущих уроках. В частности, подчеркивалось, что реляционные базы данных хранят информацию во многих таблицах, каждая из которых содержит поля, связанные с полями из других таблиц. Для создания ссылок из одной таблицы на другие используются *ключи*.

Чтобы реляционная база данных работала должным образом, необходимо убедиться в том, что данные в ее таблицах введены правильно. Например, если в таблице `Orders` хранится информация о заказах, а в таблице `OrderItems` — их детальные описания, вы должны быть уверены, что все идентификаторы заказов, упомянутые в таблице `OrderItems`, существуют и в таблице `Orders`. Аналогично каждый клиент, упомянутый в таблице `Orders`, не должен быть забыт и в таблице `Customers`.

Несмотря на то что можно проводить соответствующие проверки перед вводом новых строк (выполняя инструкцию `SELECT` для другой таблицы, дабы удостовериться в том, что нужные значения правильны), лучше избегать этого по следующим причинам.

- Если правила, обеспечивающие целостность базы данных, навязываются на клиентском уровне, их придется соблюдать каждому клиенту (некоторые из клиентов наверняка не захотят этого делать).

- Вам придется принудительно ввести правила для выполнения запросов UPDATE и DELETE.
- Выполнение проверок на стороне клиента — процесс, отнимающий много времени. Заставить СУБД выполнять такие проверки — намного более эффективный подход.

Ограничения

Правила, регламентирующие ввод и обработку информации в базе данных.

СУБД принудительно обеспечивает ссылочную целостность за счет ограничений, налагаемых на таблицы базы данных. Большинство ограничений задается в определениях таблиц (с помощью инструкций CREATE TABLE или ALTER TABLE; см. урок 17).

ПРЕДУПРЕЖДЕНИЕ: **ограничения зависят от СУБД**

Существуют различные типы ограничений, и каждая СУБД обеспечивает собственный уровень их поддержки. Следовательно, примеры данного урока могут работать не так, как вы предполагаете. Обратитесь к документации своей СУБД, прежде чем выполнять их.

Первичные ключи

О первичных ключах говорилось на уроке 1. Первичный ключ — это особое ограничение, применяемое для того, чтобы значения в столбце (или наборе столбцов) были уникальными и никогда не изменялись. Другими словами, это столбец (или столбцы) таблицы, значения которого однозначно идентифицируют каждую строку таблицы. Это облегчает обработку отдельных строк и доступ к ним. Без первичных ключей было бы очень трудно обновлять или удалять определенные строки, не затрагивая при этом другие.

Любой столбец таблицы может быть назначен на роль первичного ключа, но только если он удовлетворяет следующим условиям.

- Никакие две строки не могут иметь одно и то же значение первичного ключа.

- Каждая строка должна иметь какое-то значение первичного ключа (в таких столбцах не должны допускаться значения NULL).
- Столбец, содержащий значения первичного ключа, не может быть модифицирован или обновлен.
- Значения первичного ключа ни при каких обстоятельствах не могут быть использованы повторно. Если какая-то строка удаляется из таблицы, ее первичный ключ не может быть назначен новой строке.

Один из способов определить первичный ключ — указать соответствующее ограничение в процессе создания таблицы.

Ввод ▼

```
CREATE TABLE Vendors
(
   vend_id          CHAR(10)   NOT NULL PRIMARY KEY,
   vend_name        CHAR(50)   NOT NULL,
   vend_address     CHAR(50)   NULL,
   vend_city        CHAR(50)   NULL,
   vend_state       CHAR(5)    NULL,
   vend_zip         CHAR(10)   NULL,
   vend_country     CHAR(50)   NULL
);
```

Анализ ▼

В данном примере в определение таблицы добавлена спецификация PRIMARY KEY, благодаря которой столбец vend_id становится первичным ключом.

Ввод ▼

```
ALTER TABLE Vendors
ADD CONSTRAINT PRIMARY KEY (vend_id);
```

Анализ ▼

Здесь в качестве первичного ключа назначен тот же самый столбец, но с использованием ключевого слова CONSTRAINT. Оно допустимо в инструкциях CREATE TABLE и ALTER TABLE.

ПРИМЕЧАНИЕ: **ключи в SQLite**

В SQLite нельзя определять ключи с помощью инструкции ALTER TABLE. Это можно делать только в первоначальной инструкции CREATE TABLE.

Внешние ключи

Внешний ключ — это столбец одной таблицы, значения которого совпадают со значениями столбца, являющегося первичным ключом другой таблицы. Внешние ключи — очень важная часть механизма обеспечения ссылочной целостности. Чтобы разобраться в том, что собой представляют внешние ключи, рассмотрим следующий пример.

Таблица Orders содержит единственную строку для каждого заказа, зафиксированного в базе данных. Информация о клиенте хранится в таблице Customers. Заказы в таблице Orders связаны с определенными строками в таблице Customers за счет идентификатора клиента, который является первичным ключом в таблице Customers. Каждый клиент имеет уникальный идентификатор. Номер заказа является первичным ключом в таблице Orders, и каждый заказ имеет свой уникальный номер.

Значения в столбце таблицы Orders, содержащем идентификаторы клиентов, не обязательно уникальные. Если клиент сделал несколько заказов, могут существовать несколько строк с тем же самым идентификатором клиента (хотя каждая из них будет иметь свой номер заказа). В то же время единственные значения, которые могут появиться в столбце идентификаторов клиента в таблице Orders, — это идентификаторы клиентов из таблицы Customers.

Именно так и образуются внешние ключи. В нашем примере внешний ключ определен как столбец идентификаторов клиентов в таблице Orders, который может принимать только значения, содержащиеся в первичном ключе таблицы Customers.

Вот один из способов определения внешнего ключа.

Ввод ▼

```
CREATE TABLE Orders
(
   order_num     INTEGER     NOT NULL PRIMARY KEY,
```

```
   order_date    DATETIME    NOT NULL,
   cust_id       CHAR(10)    NOT NULL REFERENCES
↳Customers(cust_id)
);
```

Анализ ▼

В этом определении таблицы используется ключевое слово REFERENCES, указывающее на то, что любое значение в столбце cust_id должно также находиться в столбце cust_id таблицы Customers.

Аналогичного результата можно добиться с помощью ключевого слова CONSTRAINT в инструкции ALTER TABLE.

Ввод ▼

```
ALTER TABLE Orders
ADD CONSTRAINT
FOREIGN KEY (cust_id) REFERENCES Customers (cust_id)
```

СОВЕТ: **внешние ключи могут воспрепятствовать случайному удалению данных**

Внешние ключи не только помогают обеспечивать ссылочную целостность, но также служат другой важной цели. После того как внешний ключ определен, СУБД не позволит удалять строки, связанные со строками в других таблицах. Например, вы не сможете удалить информацию о клиенте, у которого есть заказы. Единственный способ это сделать состоит в предварительном удалении связанных с клиентом заказов (для чего, в свою очередь, нужно удалить информацию об элементах этих заказов). Поскольку требуется столь методичное и целенаправленное удаление, внешние ключи могут оказать помощь в предотвращении случайного удаления данных.

Однако в некоторых СУБД поддерживается возможность *каскадного удаления*. Если такая функция реализована, можно удалять все связанные со строкой данные при удалении ее из таблицы. Например, если разрешено каскадное удаление и имя клиента удаляется из таблицы Customers, все связанные с его заказами строки удаляются автоматически.

Ограничения уникальности

Ограничения уникальности обеспечивают неповторяемость всех данных в столбце (или в наборе столбцов). Такие столбцы напоминают первичные ключи, однако имеются и важные отличия.

- Таблица может содержать множество ограничений уникальности, но у нее должен быть только один первичный ключ.
- Столбцы с ограничением уникальности могут содержать значения NULL.
- Столбцы с ограничением уникальности можно модифицировать и обновлять.
- Значения столбцов с ограничением уникальности можно использовать повторно.
- В отличие от первичных ключей, столбцы с ограничениями уникальности не могут быть использованы для определения внешних ключей.

Примером такого ограничения может служить таблица с данными о сотрудниках. Каждый из них имеет свой уникальный номер карточки социального страхования, но вы вряд ли будете использовать его в качестве первичного ключа, поскольку он слишком длинный (и, кроме того, вы вряд ли захотите сделать эту информацию легко доступной). Поэтому каждому сотруднику присваивается уникальный идентификатор (первичный ключ) в дополнение к его номеру карточки социального страхования.

Поскольку идентификатор сотрудника является первичным ключом, можно быть уверенным в том, что он уникален. А для того чтобы СУБД проверила уникальность каждого номера карточки социального страхования (исключив вероятность опечатки при вводе, когда для одного сотрудника указывается номер карточки другого), необходимо задать ограничение UNIQUE для столбца, в котором содержатся номера карточек социального страхования.

Синтаксис ограничения уникальности напоминает синтаксис других ограничений: в определении таблицы указывается ключевое слово UNIQUE или отдельно задается ограничение CONSTRAINT.

Ограничения на значения столбца

Ограничения на значения столбца нужны для того, чтобы данные в столбце (или наборе столбцов) соответствовали указанным вами критериям. Наиболее распространенными ограничениями являются следующие.

- Ограничение максимального и минимального значений — например, для предотвращения появления заказов на 0 (нуль) товаров (хотя 0 и является допустимым числом).
- Указание диапазонов — например, ограничение на то, чтобы дата отгрузки наступала позже или соответствовала текущей дате и не отстояла от нее больше, чем на год.
- Разрешение только определенных значений — например, разрешение вводить в поле "пол" только букву `M` или `F`.

Типы данных (см. урок 1) сами по себе ограничивают данные, которые могут храниться в столбце, а ограничения на значения столбца предъявляют дополнительные требования к данным определенного типа, чтобы в базу данных можно было вводить только конкретные значения. Вместо того чтобы полагаться на клиентское приложение или добросовестность пользователя, СУБД самостоятельно отвергает некорректные данные.

В следующем примере налагается ограничение на значения столбца `quantity` таблицы `OrderItems`, с тем чтобы для всех товаров указывалось количество, большее `0`.

Ввод ▼

```
CREATE TABLE OrderItems
(
   order_num     INTEGER     NOT NULL,
   order_item    INTEGER     NOT NULL,
   prod_id       CHAR(10)    NOT NULL,
   quantity      INTEGER     NOT NULL CHECK (quantity > 0),
   item_price    MONEY       NOT NULL
);
```

Анализ ▼

После применения этого ограничения каждая добавляемая (или обновляемая) строка будет проверяться на предмет того, чтобы количество товаров было больше нуля.

Дабы проконтролировать тот факт, что в столбце с обозначением пола может содержаться только буква M или F, добавьте следующую строку в инструкцию ALTER TABLE.

Ввод ▼

```
ADD CONSTRAINT CHECK (gender LIKE '[MF]')
```

> СОВЕТ: **пользовательские типы данных**
>
> Пользователи некоторых СУБД могут создавать собственные типы данных. Обычно это базовые типы, но с дополнительными ограничениями на значения. Например, можно определить собственный тип данных, назвав его gender (пол). Он будет представлять значения, состоящие из одной буквы, с ограничением, допускающим только два варианта: M или F (и, возможно, NULL, если пол служащего неизвестен). Такой тип данных можно указывать в определениях таблиц. Преимущество пользовательских типов данных состоит в том, что подобные ограничения можно задать всего один раз (в определении типа данных), после чего они будут автоматически применяться всякий раз, когда задействуется пользовательский тип данных. Посмотрите в документации своей СУБД, поддерживает ли она пользовательские типы данных.

Что такое индексы

Индексы предназначены для логической сортировки хранимых данных, что позволяет повысить скорость поиска и сортировки строк в запросах. Лучший способ понять, что такое индекс, — взглянуть на предметный указатель в конце книги.

Предположим, вы хотите найти вхождения слова *индекс* в книге. "Лобовым" способом решения этой задачи было бы вернуться на первую страницу и просмотреть каждую строку каждой страницы в поисках совпадений. Такой вариант, конечно, допустим, но очевидно,

что он нереален. Просмотреть несколько страниц текста еще можно, но просматривать подобным образом всю книгу — плохая затея. Чем больше объем текста, в котором нужно провести поиск, тем больше времени требуется на выявление мест вхождения нужных данных.

Именно поэтому книги снабжают предметными указателями. Предметный указатель — это список ключевых слов и терминологических словосочетаний, расположенных в алфавитном порядке, со ссылками на страницы, на которых искомые слова упоминаются в книге. Чтобы найти термин *индекс*, необходимо посмотреть в предметном указателе, на каких страницах он встречается.

Что делает предметный указатель столь эффективным средством поиска? Попросту говоря, тот факт, что он правильно отсортирован. Трудность поиска слов в книге обусловлена не тем, что ее объем слишком велик, а тем, что слова в ней не отсортированы в алфавитном порядке. Если бы они были отсортированы подобно тому, как это делается в словарях, в предметном указателе не было бы необходимости (именно поэтому словари не снабжаются предметными указателями).

Индексы баз данных работают схожим образом. Данные первичного ключа всегда отсортированы — СУБД делает это за вас. Таким образом, извлечение указанных строк по первичному ключу всегда осуществляется быстро и эффективно.

Однако поиск значений в других столбцах выполняется уже не столь эффективно. Что произойдет, например, если попытаться получить список всех клиентов, проживающих в определенном штате? Поскольку таблица не отсортирована по названиям штатов, СУБД придется просматривать каждую строку таблицы (начиная с самой первой), отыскивая совпадения точно так же, как это сделали бы вы в поисках вхождений слов в книге, не имеющей предметного указателя.

Решение указанной проблемы состоит в использовании индекса. В качестве индекса можно назначить один или несколько столбцов, чтобы СУБД хранила отсортированный список содержимого для внутренних целей. После того как индекс определен, СУБД применяет его точно так же, как вы работаете с предметным указателем книги. Она проводит поиск в отсортированном индексе, чтобы найти местоположения всех совпадений и затем извлечь соответствующие строки.

Но прежде чем создавать множество индексов, примите во внимание следующее.

- Индексы повышают производительность запросов, связанных с извлечением данных, но ухудшают производительность

операций добавления, модификации и удаления строк. Это связано с тем, что при выполнении подобных операций СУБД должна еще и динамически обновлять индекс.

- Для хранения индекса требуется дополнительное место на диске.
- Не все данные подходят для индексации. Данные, которые не являются достаточно уникальными (как, например, названия штатов в столбце `cust_state`), не дадут такого выигрыша от индексации, как данные, которые имеют больше возможных значений (как, например, имя и фамилия).
- Индексы применяются для фильтрации и сортировки данных. Если вы часто сортируете данные одинаковым образом, эти данные могут быть кандидатом на индексацию.
- В качестве индекса можно определить несколько столбцов (например, с названием штата и названием города). Такой индекс будет полезен, только если данные сортируются в порядке "штат плюс город" (если вы захотите отсортировать данные лишь по названию города, индекс окажется бесполезен).

Не существует твердых правил относительно того, что и когда следует индексировать. В большинстве СУБД предлагаются утилиты, которые можно применять для определения эффективности индексов, и ими следует регулярно пользоваться.

Индексы создаются с помощью инструкции `CREATE INDEX`, синтаксис которой зависит от СУБД. Следующая инструкция создает простой индекс для столбца с наименованиями товаров в таблице `Products`.

Ввод ▼

```
CREATE INDEX prod_name_ind
ON Products (prod_name);
```

Анализ ▼

Каждый индекс должен обладать уникальным именем. В данном случае оно определено как `prod_name_ind`. Ключевое слово `ON` служит для указания таблицы, которая должна быть проиндексирована, а столбцы, включаемые в индекс (в данном примере он один), указываются в круглых скобках после имени таблицы.

СОВЕТ: **пересмотр индексов**

Эффективность индексов снижается, если в таблицу добавляются данные или происходит их обновление. Многие администраторы баз данных считают так: то, что когда-то было идеальным набором индексов, может перестать быть таковым после нескольких месяцев работы с базой данных. Целесообразно регулярно пересматривать индексы и, в случае необходимости, настраивать их.

Что такое триггеры

Триггеры — это особые хранимые процедуры, автоматически выполняемые при наступлении определенных событий в базе данных. Триггеры могут быть связаны с выполнением инструкций `INSERT`, `UPDATE` и `DELETE` по отношению к указанным таблицам.

В отличие от хранимых процедур (которые представляют собой заранее записанные инструкции SQL), триггеры связаны с отдельными таблицами. Триггер, ассоциированный с инструкциями `INSERT` по отношению к таблице `Orders`, будет выполняться только в том случае, если в эту таблицу добавляется строка. Аналогично, триггер, относящийся к инструкциям `INSERT` и `UPDATE` для таблицы `Customers`, будет выполняться только в случае применения указанных операций по отношению к данной таблице.

Код триггера может иметь доступ к следующим данным:

- все новые данные в инструкциях `INSERT`;
- все новые и старые данные в инструкциях `UPDATE`;
- удаляемые данные в инструкциях `DELETE`.

В зависимости от СУБД, с которой вы работаете, триггер может выполняться до или после связанной с ним операции.

Чаще всего триггеры применяются для следующих целей:

- обеспечение непротиворечивости данных (например, для преобразования всех названий штатов в верхний регистр при выполнении инструкций `INSERT` или `UPDATE`);
- выполнение действий по отношению к другим таблицам на основе изменений, которые были сделаны в какой-то таблице

(например, для внесения записи в контрольный журнал с целью регистрации каждого случая обновления или удаления строки);

- дополнительная проверка и, в случае необходимости, отмена ввода данных (например, дабы удостовериться в том, что разрешенная для клиента сумма кредита не превышена, в противном случае операция блокируется);
- подсчет значений вычисляемых полей или обновление меток даты/времени.

Как вы, наверное, уже догадываетесь, синтаксис создания триггеров зависит от СУБД. За подробностями обратитесь к документации своей СУБД.

В следующем примере создается триггер, преобразующий значения столбца `cust_state` в таблице `Customers` в верхний регистр при выполнении любых инструкций `INSERT` и `UPDATE`.

Вот версия для SQL Server.

Ввод ▼

```
CREATE TRIGGER customer_state
ON Customers
FOR INSERT, UPDATE
AS
UPDATE Customers
SET cust_state = Upper(cust_state)
WHERE Customers.cust_id = inserted.cust_id;
```

Ниже приведена версия для Oracle и PostgreSQL.

Ввод ▼

```
CREATE TRIGGER customer_state
AFTER INSERT OR UPDATE
FOR EACH ROW
BEGIN
UPDATE Customers
SET cust_state = Upper(cust_state)
WHERE Customers.cust_id = :OLD.cust_id
END;
```

СОВЕТ: **ограничения работают быстрее, чем триггеры**

Как правило, ограничения обрабатываются быстрее, чем триггеры, поэтому старайтесь по возможности использовать именно их.

Безопасность баз данных

Нет ничего более ценного для организации, чем ее данные, поэтому они всегда должны быть защищены от кражи или случайного просмотра. В то же время данные должны быть всегда доступны для определенных пользователей, поэтому большинство СУБД предоставляет в распоряжение администраторов инструменты, посредством которых можно разрешать или ограничивать доступ к данным.

В основе любой системы безопасности лежит авторизация и аутентификация пользователей. Так называется процесс, в ходе которого пользователь подтверждает, что он — это именно он и что ему разрешено проводить операции, которые он собирается выполнить. Одни СУБД применяют для этого средства безопасности операционной системы, другие хранят свои собственные списки пользователей и паролей, третьи интегрируются с внешними серверами службы каталогов.

Чаще всего применяются следующие ограничения безопасности:

- ограничение доступа к административным функциям (создание таблиц, изменение или удаление существующих таблиц и т.п.);
- ограничение доступа к отдельным базам данных или таблицам;
- ограничение типа доступа (только для чтения, доступ к отдельным столбцам и т.п.);
- организация доступа к таблицам только через представления или хранимые процедуры;
- создание нескольких уровней безопасности, вследствие чего обеспечивается различная степень доступа и контроля на основе учетных записей пользователей.
- ограничение возможности управлять учетными записями пользователей.

Управление безопасностью осуществляется посредством инструкций GRANT и REVOKE, хотя большинство СУБД предлагает интерактивные утилиты администрирования, в которых применяются те же самые инструкции.

Резюме

На этом уроке вы узнали, как применять некоторые расширенные средства SQL. Ограничения — важная часть системы обеспечения ссылочной целостности; индексы помогут улучшить производительность запросов, связанных с извлечением данных; триггеры можно использовать для обработки данных перед началом или сразу после завершения определенных операций, а параметры системы безопасности можно применять для управления доступом к данным. Наверняка ваша СУБД в той или иной форме обеспечивает указанные возможности. Обратитесь к ее документации, чтобы подробнее узнать об этом.

ПРИЛОЖЕНИЕ А

Сценарии демонстрационных таблиц

Процесс написания инструкций SQL требует хорошего понимания структуры базы данных. Без знания того, какая информация в какой таблице хранится, как таблицы связаны одна с другой и как распределены данные в строках, невозможно написать эффективный SQL-код.

Настоятельно рекомендую проверить на практике каждый пример каждого урока книги. Во всех уроках используется один и тот же набор файлов данных. Чтобы вам было легче разбираться в примерах и выполнять их по мере изучения книги, в этом приложении описываются применяемые таблицы, отношения между ними и способы построения таблиц (или их получения в готовом виде).

Демонстрационные таблицы

Таблицы, используемые на протяжении всей книги, являются частью системы регистрации заказов воображаемого дистрибьютора игрушек. Эти таблицы служат для решения нескольких задач:

- взаимодействие с поставщиками;
- работа с каталогами товаров;
- управления списками клиентов;
- прием заказов от клиентов.

Всего требуется пять таблиц (они тесно связаны между собой в рамках схемы реляционной базы данных). В следующих разделах описана каждая из таблиц.

ПРИМЕЧАНИЕ: **упрощенные примеры**

Таблицы, используемые в книге, нельзя назвать полными. Реальная система регистрации заказов хранила бы множество других данных, не включенных в представленные таблицы (например, платежные реквизиты, номера инвойсов, контрольные номера поставок и многое другое). В то же время с помощью этих таблиц будет наглядно показано, как структурируются базы данных и какие отношения между таблицами существуют на практике. Вы сможете применить полученные знания по отношению к своим собственным базам данных.

Описания таблиц

Далее будут рассмотрены все пять демонстрационных таблиц с указанием имен столбцов каждой таблицы и их описаниями.

Таблица `Vendors`

В таблице `Vendors` (табл. А.1) хранятся данные о поставщиках, товары которых продаются. Для каждого поставщика в этой таблице имеется отдельная запись, а столбец с идентификаторами поставщиков (`vend_id`) используется для указания соответствия между товарами и поставщиками.

ТАБЛИЦА А.1. **Столбцы таблицы `Vendors`**

Столбец	Описание
`vend_id`	Уникальный идентификатор поставщика
`vend_name`	Имя поставщика
`vend_address`	Адрес поставщика
`vend_city`	Город поставщика
`vend_state`	Штат поставщика
`vend_zip`	ZIP-код поставщика
`vend_country`	Страна поставщика

- Для всех таблиц должны быть определены первичные ключи. В данной таблице в качестве первичного ключа следует использовать столбец `vend_id`.

Таблица Products

Таблица Products (табл. А.2) содержит каталог товаров, по одному товару в строке. Каждый товар имеет уникальный идентификатор (столбец prod_id) и связан с соответствующим поставщиком через столбец vend_id (уникальный идентификатор поставщика).

ТАБЛИЦА А.2. **Столбцы таблицы Products**

Столбец	Описание
prod_id	Уникальный идентификатор товара
vend_id	Идентификатор поставщика товара (связан со столбцом vend_id таблицы Vendors)
prod_name	Название товара
prod_price	Цена товара
prod_desc	Описание товара

- Для всех таблиц должны быть определены первичные ключи. В данной таблице в качестве первичного ключа следует использовать столбец prod_id.
- Для обеспечения ссылочной целостности следует определить внешний ключ на основе столбца vend_id, связав его со столбцом vend_id таблицы Vendors.

Таблица Customers

В таблице Customers (табл. А.3) хранится информация обо всех клиентах. Каждый из них имеет уникальный идентификатор (столбец cust_id).

ТАБЛИЦА А.3. **Столбцы таблицы Customers**

Столбец	Описание
cust_id	Уникальный идентификатор клиента
cust_name	Имя клиента
cust_address	Адрес клиента
cust_city	Город клиента
cust_state	Штат клиента
cust_zip	ZIP-код клиента

Окончание табл. А.3

Столбец	Описание
cust_country	Страна клиента
cust_contact	Контактное лицо клиента
cust_email	Контактный адрес электронной почты клиента

- Для всех таблиц должны быть определены первичные ключи. В данной таблице в качестве первичного ключа следует использовать столбец cust_id.

Таблица Orders

В таблице Orders (табл. А.4) хранится информация о заказах клиентов (без подробностей). Каждый заказ имеет уникальный номер (столбец order_num). Заказы связаны с соответствующими клиентами через столбец cust_id (который связан с уникальным идентификатором клиента в таблице Customers).

ТАБЛИЦА А.4. **Столбцы таблицы Orders**

Столбец	Описание
order_num	Уникальный номер заказа
order_date	Дата заказа
cust_id	Идентификатор клиента, сделавшего заказ (связан со столбцом cust_id таблицы Customers)

- Для всех таблиц должны быть определены первичные ключи. В данной таблице в качестве первичного ключа следует использовать столбец order_num.
- Для обеспечения ссылочной целостности следует определить внешний ключ на основе столбца cust_id, связав его со столбцом cust_id таблицы Customers.

Таблица OrderItems

В таблице OrderItems (табл. А.5) хранятся элементы всех заказов, и для каждого элемента каждого заказа выделено по одной

строке. Каждой строке таблицы Orders соответствует одна или несколько строк в таблице OrderItems. Каждый элемент заказа уникальным образом идентифицирован посредством номера заказа в совокупности с номером элемента в заказе (первый элемент заказа, второй и т.д.). Элемент заказа связан с соответствующим ему заказом через столбец order_num (который соотносит его с уникальным идентификатором заказа в таблице Orders). Кроме того, каждая запись об элементе заказа содержит идентификатор товара (который связывает товар с таблицей Products).

ТАБЛИЦА А.5. **Столбцы таблицы OrderItems**

Столбец	Описание
order_num	Номер заказа (связан со столбцом order_num таблицы Orders)
order_item	Номер элемента заказа (последовательно присваиваемый в заказе)
prod_id	Идентификатор товара (связан со столбцом prod_id таблицы Products)
quantity	Количество заказанных товаров
item_price	Цена за единицу товара

- Для всех таблиц должны быть определены первичные ключи. В данной таблице в качестве первичного ключа следует использовать связку столбцов order_num и order_item.
- Для обеспечения ссылочной целостности следует определить внешние ключи на основе столбца order_num, связав его с полем order_num таблицы Orders, и столбца prod_id, связав его с полем prod_id таблицы Products.

Администраторы СУБД часто используют диаграммы отношений, демонстрирующие, как связаны между собой таблицы базы данных. Помните, что отношения определяются внешними ключами, описанными в табл. А.1–А.5. На рис. А.1 приведена диаграмма отношений для пяти таблиц, рассмотренных в данном приложении.

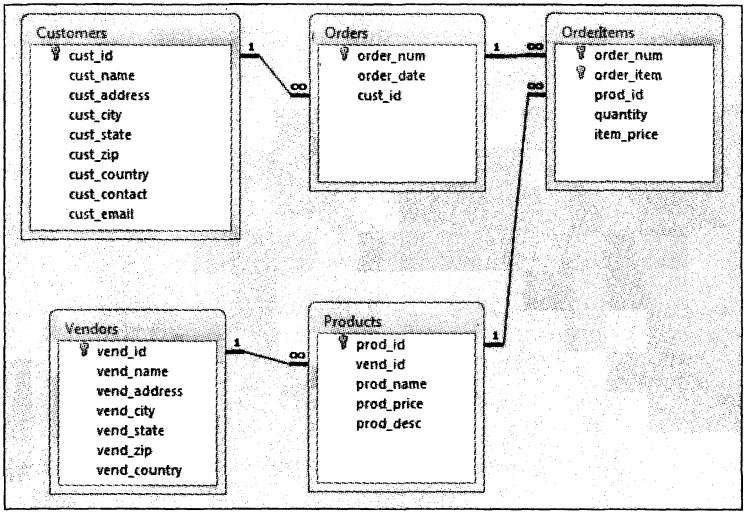


РИС. А.1. Диаграмма отношений между демонстрационными таблицами

Получение демонстрационных таблиц

Чтобы попрактиковаться в выполнении представленных в книге примеров, вам понадобится набор заполненных таблиц. Все необходимое можно найти на сайте книги по следующим адресам:

```
http://www.forta.com/books/0672336073/
http://www.williamspublishing.com/
       Books/978-5-8459-1858-1.html
```

Загрузка готовых баз данных

Перейдите по одному из указанных выше адресов, чтобы получить готовые базы данных в следующих форматах:

- Apache OpenOffice Base;
- Microsoft Access (2000 и 2007);
- SQLite.

Если вы воспользуетесь этими файлами, то вам не придется выполнять никаких сценариев создания и заполнения таблиц.

Загрузка SQL-сценариев для различных СУБД

Большинство СУБД хранят данные в форматах, которые не позволяют распространять файлы баз данных (в отличие от Access, OpenOffice Base и SQLite). Для таких СУБД по указанным выше адресам можно загрузить SQL-сценарии, включающие два файла:

- файл `create.txt`, содержащий инструкции SQL, которые необходимы для создания пяти демонстрационных таблиц базы данных (включая определения всех первичных и внешних ключей);
- файл `populate.txt`, содержащий инструкции `INSERT`, используемые для заполнения демонстрационных таблиц.

Инструкции SQL, содержащиеся в этих файлах, зависят от СУБД, поэтому выполняйте только те сценарии, которые соответствуют вашей СУБД. Эти сценарии предназначены лишь для удобства читателей, и никакая ответственность за возможные проблемы при их использовании не предполагается.

К тому моменту, когда книга уходила в печать, были доступны сценарии для следующих СУБД:

- IBM DB2;
- Microsoft SQL Server (включая Microsoft SQL Server Express);
- MariaDB
- MySQL;
- Oracle (включая Oracle Express);
- PostgreSQL.

По мере необходимости этот список может быть дополнен другими СУБД.

В приложении Б приведены инструкции по выполнению сценариев в популярных средах разработки.

ПРИМЕЧАНИЕ: **вначале создать, потом заполнять**

Вначале следует выполнить сценарий создания таблиц, и только потом — сценарий их заполнения. Убедитесь в том, что сценарии не возвращают никаких сообщений об ошибках. Если сценарий создания таблиц потерпит неудачу, выявите и устраните возникшие проблемы, а потом уже заполняйте таблицы.

ПРИМЕЧАНИЕ: **инструкции по настройке для конкретных СУБД**

Действия по настройке конкретных СУБД существенно различаются. Загрузив SQL-сценарии на сайте книги, вы найдете в каждом архиве файл README с описанием действий, которые необходимо выполнить в каждом конкретном случае.

ПРИЛОЖЕНИЕ Б

Работа с популярными программами

Как объяснялось на уроке 1, SQL — это язык, а не программа. Для того чтобы работать с примерами книги, необходима программа, поддерживающая выполнение инструкций SQL.

В данном приложении описывается порядок выполнения SQL-запросов в наиболее популярных СУБД и средах разработки баз данных.

Для проверки SQL-кодов можно использовать любую из рассмотренных здесь систем. На чем же остановиться?

- Многие СУБД поставляются со своими собственными клиентскими утилитами. С ними вполне можно начинать работу. Однако для них не всегда характерен интуитивно понятный пользовательский интерфейс.
- Если вы веб-разработчик, можете воспользоваться любым языком разработки серверных приложений, включая ASP.NET, ColdFusion, Java, JSP, PHP, Python и Ruby.

Apache OpenOffice Base

Apache OpenOffice Base — это открытое клиентское приложение на основе Java, предназначенное для работы с базами данных. Запросы для него можно писать непосредственно на SQL. Выполните следующие действия.

1. Откройте свою базу данных в Apache OpenOffice Base.
2. Выберите раздел Queries на панели Database в левой части окна.
3. На панели Tasks щелкните на кнопке Create Query In SQL View, чтобы отобразить окно Query Design.
4. Введите свой SQL-запрос в большом текстовом поле (все окно представляет собой текстовое поле).

5. Чтобы выполнить SQL-запрос, щелкните на кнопке Run Query (на ней изображены документы и зеленая метка). Можно также нажать клавишу <F5> или выбрать команду Run Query в меню Edit.

Adobe ColdFusion

Adobe ColdFusion представляет собой платформу для разработки веб-приложений. В ColdFusion для создания сценариев применяется язык, основанный на дескрипторах (тегах). Чтобы протестировать SQL-код, создайте простую страницу, которую можно будет отобразить, открыв ее в браузере. Для этого выполните следующие действия.

1. Прежде чем обращаться к каким-либо базам данных из ColdFusion, должен быть определен источник данных. Программа ColdFusion Administrator предлагает веб-интерфейс для создания источников данных (в случае необходимости обратитесь к документации ColdFusion).
2. Создайте новую страницу ColdFusion (с расширением CFM).
3. Используйте дескрипторы CFML <CFQUERY> и </CFQUERY> для создания блока запроса. Присвойте ему имя, используя атрибут NAME, и укажите источник данных в атрибуте DATASOURCE.
4. Введите свою инструкцию SQL между дескрипторами <CFQUERY> и </CFQUERY>.
5. Используйте цикл <CFDUMP> или <CFOUTPUT> для отображения результатов запроса.
6. Сохраните страницу в любом каталоге исполняемых файлов, находящемся в структуре корневого каталога веб-сервера.
7. Отобразите страницу, открыв ее в браузере.

IBM DB2

DB2 компании IBM — это мощная высокопроизводительная кроссплатформенная СУБД. Она поставляется с целым набором клиентских инструментов, которые могут быть использованы для выполнения SQL-запросов. Приведенные ниже инструкции

предназначены для Java-утилиты Control Center — одной из самых простых и универсальных среди всех утилит СУБД.

1. Запустите Control Center.
2. На панели Object View в левой части окна будет приведен список всех доступных баз данных. Раскройте раздел All Databases и выберите требуемую базу данных.
3. Чтобы открыть окно Query, щелкните правой кнопкой мыши на названии своей базы данных и выберите пункт Query либо (пока база данных активизирована) пункт Query в меню Selected.
4. Введите инструкцию SQL в верхнее поле.
5. Щелкните на кнопке Execute (с изображением зеленой стрелки, направленной вправо), чтобы выполнить запрос.
6. В нижнем окне отобразится статусная информация. Щелкните на вкладке Query Results, чтобы просмотреть результаты запроса в виде таблицы.

MariaDB

В MariaDB нет собственной клиентской утилиты, поэтому применяются клиенты MySQL (обе СУБД полностью совместимы). Обратитесь к разделу, посвященному MySQL.

Microsoft Access

Microsoft Access обычно используется интерактивно для создания баз данных и управления таблицами. В программе имеется конструктор запросов, который можно применять для интерактивного построения инструкций SQL. Но многие не учитывают, что конструктор запросов позволяет также вводить SQL-код напрямую. Выполните следующие действия.

1. Запустите Microsoft Access. Программа предложит открыть (или создать) базу данных. Откройте базу данных, с которой собираетесь работать.
2. Выберите меню Запросы в окне базы данных, а затем дважды щелкните на ссылке Создание запроса в режиме конструктора (либо щелкните на кнопке Создать и выберите в

появившемся окне пункт Конструктор). При работе с лентой перейдите на вкладку Создание и щелкните на кнопке Конструктор запросов.

3. Появится диалоговое окно Добавление таблицы. Закройте его, не выбрав ни одну из таблиц.
4. Выберите команду Режим SQL в меню Вид. При работе с лентой раскройте меню кнопки Режим на контекстной вкладке Конструктор и выберите пункт Режим SQL.
5. Введите свою инструкцию SQL в окне запроса.
6. Чтобы выполнить SQL-запрос, щелкните на кнопке Запуск (она помечена красным восклицательным знаком). Результаты запроса отобразятся в том же окне, но в режиме таблицы.
7. Можно переключаться с режима ввода запросов (вам нужно будет повторно выполнить команду Режим SQL для изменения SQL-запроса) на режим отображения их результатов. Можно также выбрать вариант Создание запроса с помощью мастера для интерактивного построения инструкций SQL.

Microsoft ASP

Microsoft ASP — это платформа разработки сценариев, ориентированная на создание веб-приложений. Для того чтобы протестировать инструкции SQL на странице ASP, необходимо создать страницу, которую можно будет отобразить на экране, открыв ее в браузере. Выполните следующие действия.

1. ASP для взаимодействия с базами данных использует ODBC, поэтому соответствующий источник данных ODBC должен быть создан заранее (об этом рассказывается в конце приложения).
2. Создайте новую страницу ASP (с расширением ASP) в любом текстовом редакторе.
3. Используйте метод `Server.CreateObject` для создания экземпляра объекта `ADODB.Connection`.
4. Используйте метод `Open` для открытия нужного источника данных ODBC.
5. Передайте свою инструкцию SQL методу `Execute` в качестве аргумента. Метод `Execute` возвращает результаты запроса. Используйте команду `Set` для сохранения полученных данных.

6. Чтобы отобразить результаты запроса, воспользуйтесь циклом <% Do While NOT EOF %>.
7. Сохраните страницу в любом каталоге исполняемых файлов, находящемся в структуре корневого каталога веб-сервера.
8. Откройте страницу в браузере.

Microsoft ASP.NET

Microsoft ASP.NET — это платформа разработки сценариев для создания веб-приложений с использованием технологий .NET. Чтобы протестировать инструкции SQL на странице ASP.NET, создайте страницу, которую можно будет отобразить на экране, открыв ее в браузере. Это можно сделать разными способами, ниже описан один из них.

1. Создайте новый файл с расширением `.aspx`.
2. Создайте подключение к базе данных, используя функцию `SqlConnection()` или `OleDbConnection()`.
3. Используйте функцию `SqlCommand()` или `OleDbCommand()` для передачи инструкции в СУБД.
4. Создайте объект `DataReader`, используя метод `ExecuteReader`.
5. Последовательно обработайте все записи, содержащиеся в объекте, чтобы получить возвращаемые значения.
6. Сохраните страницу в любом каталоге исполняемых файлов, находящемся в структуре корневого каталога веб-сервера.
7. Откройте страницу в браузере.

Microsoft Query

Microsoft Query — это отдельная утилита создания SQL-запросов, которая является удобным средством тестирования инструкций SQL с использованием источников данных ODBC. Microsoft Query больше не поставляется в составе Windows, но может опционально устанавливаться вместе с другими продуктами Microsoft, а также с приложениями сторонних разработчиков.

СОВЕТ: **поиск Microsoft Query**

Microsoft Query часто устанавливается в системе вместе с другими программными пакетами Microsoft (например, Office), но обычно это происходит только при полной установке пакета. Если утилита не отображается в меню Пуск, попробуйте осуществить поиск файла MSQRY32.EXE или MSQUERY.EXE.

Для работы с Microsoft Query необходимо выполнить следующие действия.

1. Microsoft Query использует ODBC для взаимодействия с базами данных, поэтому в системе предварительно должен быть создан источник данных ODBC (об этом рассказывается в конце приложения).
2. Прежде чем начинать использовать утилиту Microsoft Query, убедитесь в том, что она установлена в системе. Просмотрите список программ, открывающийся после щелчка на кнопке Пуск, и найдите утилиту.
3. В меню Файл выберите команду Выполнить запрос SQL. Откроется окно Выполнение запроса SQL.
4. Щелкните на кнопке Источники, чтобы выбрать источник данных ODBC. Если нужный вам источник отсутствует в списке, щелкните на кнопке Обзор, чтобы найти его. После того как будет выбран нужный источник данных, щелкните на кнопке OK.
5. Введите свою инструкцию в поле Инструкция SQL.
6. Щелкните на кнопке Выполнить, чтобы выполнить SQL-запрос и отобразить полученные данные.

Microsoft SQL Server (включая Microsoft SQL Server Express)

Microsoft SQL Server содержит мощную административную утилиту, называемую SQL Server Management Studio. Она позволяет решать множество задач: администрировать базы данных, управлять безопасностью, создавать отчеты и многое другое. Она также предоставляет удобную среду для создания и тестирования SQL-запросов. Вот как работать с SQL Server Management Studio.

1. Запустите SQL Server Management Studio.
2. В случае необходимости укажите информацию о сервере и введите свои учетные данные.
3. Появится окно SQL Server Management Studio, разделенное на несколько панелей. На панели Object Explorer в левой части окна приведен список всех баз данных с описанием их параметров. На панелях инструментов в верхней части окна доступны кнопки для выполнения конкретных действий. Основную часть окна занимает большое текстовое поле, предназначенное для ввода инструкций SQL.
4. В левой части нижней панели инструментов находится раскрывающийся список, в котором указана текущая база данных. Если необходимо, выберите в списке другую базу данных (это эквивалентно выполнению инструкции `USE`).
5. Введите свой SQL-запрос в большом текстовом окне, после чего щелкните на кнопке Execute Query (с изображением красного восклицательного знака), чтобы выполнить его. (Можно также нажать клавишу <F5> или выбрать команду Execute в меню Query.)
6. Результаты отобразятся на отдельной панели под окном SQL-кода.
7. Щелкайте на вкладках внизу окна запроса для переключения между режимами просмотра данных и просмотра полученных сообщений.

MySQL

С MySQL можно работать двумя способами. СУБД поставляется вместе с утилитой командной строки, называемой `mysql`. Это сугубо текстовое средство создания запросов, которое может применяться для выполнения любых инструкций SQL. Кроме того, разработчики выпустили интерактивную утилиту MySQL Workbench. Ее обычно приходится загружать и устанавливать отдельно, поэтому она может иметься не во всех инсталляциях СУБД. В то же время настоятельно рекомендуется работать с ней при изучении MySQL.

Чтобы воспользоваться утилитой `mysql`, выполните следующие действия.

1. Введите `mysql`, чтобы запустить эту утилиту. В зависимости от ограничений, налагаемых системой безопасности, вам

может понадобиться указать параметры `-u` и `-p`, чтобы ввести имя пользователя и пароль.

2. В ответ на приглашение `mysql>` введите `USE` *`база_данных`*, указав тем самым имя рабочей базы данных.
3. Введите свой SQL-запрос после приглашения `mysql>`, проверив, чтобы каждая инструкция заканчивалась точкой с запятой (`;`). Результаты будут отображены на экране.
4. Введите `\h` для получения списка доступных команд или `\s` для получения информации о статусе (включая информацию о версии MySQL).
5. Введите `\q`, чтобы выйти из утилиты `mysql`.

Чтобы воспользоваться утилитой MySQL Workbench, выполните следующее.

1. Запустите утилиту.
2. В самом левом столбце будут перечислены доступные подключения к базам данных MySQL. Щелкните на любом подключении, чтобы открыть его, или выберите команду New Connection, если нужной базы данных нет в списке.
3. После подключения к базе данных появится окно с панелями. На панели Object Browser в левой части окна перечислены доступные базы данных, большая текстовая панель в центре предназначена для ввода инструкций SQL, а результаты запросов и сообщения отображаются в нижней панели.
4. Щелкните на кнопке +SQL, чтобы открыть новое окно SQL-запроса.
5. После ввода запроса щелкните на кнопке Execute (с изображением молнии), чтобы выполнить его. Результаты запроса отобразятся внизу.

Oracle

В Oracle имеется большой набор административных и клиентских утилит. При изучении SQL лучше всего пользоваться утилитой Oracle SQL Developer. Она может инсталлироваться вместе с самой СУБД либо загружаться и инсталлироваться отдельно. Ниже описано, как работать с ней.

1. Запустите Oracle SQL Developer. (В Windows это нужно сделать через специальный файл сценария, а не через саму СУБД.)
2. Прежде чем начать работать с какой-либо базой данных, необходимо создать подключение к ней. Для этого воспользуйтесь командами, доступными на панели Connections в левой части окна.
3. После подключения к базе данных можно воспользоваться вкладкой SQL Worksheet в окне Query Builder для ввода инструкций SQL.
4. Чтобы выполнить SQL-запрос, щелкните на кнопке Execute (с изображением молнии). Результаты запроса отобразятся на нижней панели.

Oracle Express

Oracle Express — это мощная, но в то же время довольно простая в применении СУБД с очень удобным веб-интерфейсом. После инсталляции СУБД в вашем распоряжении окажется ссылка Getting Started для запуска административной веб-страницы, на которой можно вводить инструкции SQL. Выполните следующие действия.

1. Откройте административную веб-страницу Oracle Express.
2. В ответ на запрос введите свои имя пользователя и пароль, которые были заданы на этапе инсталляции СУБД.
3. После регистрации в системе вы увидите ряд пиктограмм, включая значок со словом SQL. Щелкните на нем, чтобы получить доступ к параметрам запросов.
4. Первая из пиктограмм называется SQL Commands. Ею можно воспользоваться для ввода инструкций SQL. (Вторая, SQL Scripts, удобна для выполнения готовых сценариев, например сценариев создания и заполнения демонстрационных таблиц, предлагаемых на сайте книги.) Щелкните на ней, чтобы открыть окно SQL Commands.
5. Введите свой SQL-запрос в верхней части окна.
6. Чтобы выполнить запрос, щелкните на кнопке Run в правом верхнем углу. Результаты отобразятся под текстом запроса.

PHP

PHP — это популярный язык написания веб-сценариев. PHP предлагает функции и библиотеки, предназначенные для подключения к различным базам данных, поэтому код, используемый для выполнения инструкций SQL, может меняться в зависимости от СУБД (и способа доступа к ней). А раз так, то невозможно предложить пошаговые инструкции, которые годились бы для любой ситуации. Ниже приведен типичный пример для MySQL. Обратитесь к документации PHP за инструкциями по подключению к своей СУБД.

1. Создайте новую PHP-страницу (с помощью одного из расширений PHP).
2. Подключитесь к своей базе данных с помощью соответствующей функции. Для MySQL функция называется `mysql_connect()`.
3. Передайте свой SQL-запрос соответствующей функции обработки запросов. Для MySQL такая функция называется `mysql_query()`.
4. В ответ будет получен массив результатов запроса, который необходимо обработать в цикле, чтобы вывести результаты на экран.
5. Сохраните страницу в любом каталоге исполняемых файлов, находящемся в структуре корневого каталога веб-сервера.
6. Откройте страницу в браузере.

PostgreSQL

С PostgreSQL можно работать двумя способами. СУБД поставляется с утилитой командной строки, которая называется `psql`. Это сугубо текстовое средство создания запросов, которое может служить для выполнения любых инструкций SQL. Кроме того, имеется интерактивная утилита pgAdmin, которая предназначена в основном для административных целей, но может также применяться и для тестирования инструкций SQL.

Чтобы воспользоваться утилитой `psql`, выполните следующие действия.

1. Введите `psql`, чтобы запустить утилиту. Чтобы загрузить конкретную базу данных, укажите ее в командной строке в

виде `psql` *`база_данных`*. (PostgreSQL не поддерживает инструкцию `USE`.)

2. Введите свой SQL-запрос в ответ на приглашение =>, убедившись в том, что каждая инструкция заканчивается точкой с запятой (`;`). Результаты будут отображены на экране.
3. Введите `\?`, чтобы отобразить список доступных команд.
4. Введите `\h`, чтобы получить справку по SQL, или `\h` *`инструкция`*, чтобы получить справку относительно конкретной инструкции SQL (например, `\h SELECT`).
5. Введите `\q`, чтобы выйти из утилиты `psql`.

Чтобы воспользоваться утилитой pgAdmin, выполните следующее.

1. Запустите утилиту (она может называться pgAdmin III).
2. В случае необходимости введите имя пользователя и пароль.
3. Утилита отобразит список серверов баз данных в левой части окна. Выберите нужный сервер, после чего утилита подключится к нему и выведет информацию о сервере.
4. Далее необходимо выбрать конкретную базу данных (в результате на панели инструментов станут доступны соответствующие кнопки).
5. Найдите на панели кнопку Execute Arbitrary SQL Queries (на ней изображена лупа со словом SQL) и щелкните на ней.
6. Появится новое окно. Убедитесь в том, что требуемая база данных выбрана в раскрывающемся списке в правом верхнем углу окна.
7. Теперь можете вводить инструкции SQL в большом текстовом поле в верхней части окна.
8. Щелкните на кнопке Execute Button (с изображением зеленой стрелки, направленной вправо), чтобы выполнить SQL-запрос. Результаты запроса отобразятся внизу.

SQLite

SQLite предназначена для встраивания в другие приложения и обычно не используется как автономная база данных. Однако в состав библиотеки входит утилита командной строки, с помощью которой можно выполнять SQL-запросы к базе данных SQLite.

Эта утилита называется `sqlite3` (или `sqlite3.exe` в Windows). Для работы с ней выполните следующие действия.

1. В идеальном случае утилита `sqlite3` и база данных должны находиться в одной и той же папке. (Базы данных SQLite хранятся в отдельном файле, чаще всего с расширением `.sqlite` или `.db`, но в принципе расширение может быть любым или вообще отсутствовать.)
2. Введите `sqlite3` *`база_данных`*`.sqlite` (заменив аргумент командной строки реальным именем файла базы данных).
3. Появится приглашение `sqlite>`, после которого можно вводить любые инструкции SQL. Все инструкции должны завершаться символом `;` (точка с запятой).
4. По умолчанию `sqlite3` отображает результаты запроса с символами `|` в качестве разделителей столбцов и без заголовков. Чтобы изменить это поведение, введите `.mode column` и нажмите клавишу <Enter>, после чего введите `.header on` и снова нажмите клавишу <Enter>.
5. Чтобы завершить сеанс работы с утилитой `sqlite3`, введите `.quite` и нажмите клавишу <Enter>.

Конфигурирование источников данных ODBC

Несколько приложений из числа вышеописанных используют для интеграции с базами данных протокол ODBC, поэтому необходимо дать краткий обзор ODBC, а также инструкции по конфигурированию источников данных ODBC.

ODBC — это стандарт, позволяющий клиентским приложениям взаимодействовать с различными серверами баз данных и различными СУБД. При наличии ODBC можно написать код с помощью одного клиента, и этот код будет взаимодействовать почти с любой базой данных или СУБД.

ODBC не является ни базой данных, ни СУБД. Скорее это системная оболочка, позволяющая всем базам данных вести себя непротиворечивым и согласованным образом. Это достигается за счет программных драйверов, выполняющих две основные функции. Во-первых, они инкапсулируют характерные для отдельных баз данных особенности, скрывая их от клиентов, а во-вторых, обеспечивают

общий язык для взаимодействия с этими базами данных (в случае необходимости выполняя нужные преобразования). Язык, используемый в ODBC, — это SQL.

Клиентские приложения ODBC не взаимодействуют с базами данных напрямую. Вместо этого они обращаются к источникам данных ODBC. Источник данных представляет собой логическую базу данных, которая включает в свой состав драйвер (база данных каждого типа имеет свой собственный драйвер) и информацию о том, как нужно подключаться к этой базе данных (пути к файлам, имена серверов и т.п.).

После того как источники данных ODBC определены, с ними может работать любое ODBC-совместимое приложение. Источники данных не специфичны для приложений, но специфичны для систем.

> ПРЕДУПРЕЖДЕНИЕ: **различия в реализации**
>
> Существует много версий системных модулей ODBC, поэтому невозможно дать четкие инструкции, применимые ко всем версиям. Обращайте внимание на подсказки, когда будете создавать свои источники данных.

Источники данных ODBC создаются с помощью ODBC-аплета панели управления Windows. Чтобы установить какой-либо источник данных, выполните следующие действия.

1. Откройте ODBC-аплет панели управления Windows (он может находиться в разделе Администрирование).
2. Большинство источников данных ODBC должны быть общесистемными (в противоположность пользовательским источникам данных), поэтому выберите вкладку Системный DSN, если она доступна.
3. Щелкните на кнопке Добавить, чтобы добавить новый источник данных.
4. Выберите драйвер, который будет применяться. Обычно по умолчанию доступен набор драйверов, обеспечивающих поддержку большинства продуктов компании Microsoft. В вашей системе могут быть установлены и другие драйверы. Необходимо выбрать драйвер, соответствующий типу базы данных, к которой вы собираетесь подключаться.

5. В зависимости от типа базы данных или СУБД вам будет предложено ввести имя сервера или путь к файлу и, возможно, регистрационную информацию. Введите запрашиваемые параметры и следуйте остальным инструкциям, чтобы создать источник данных.

ПРИЛОЖЕНИЕ В

Синтаксис инструкций SQL

Для того чтобы помочь вам быстро узнать нужный синтаксис, в этом приложении описывается синтаксис наиболее распространенных инструкций SQL. Каждый раздел начинается с краткого описания инструкции, а затем приводится ее синтаксис. Для удобства даются также ссылки на уроки, на которых рассматривались соответствующие инструкции.

При изучении синтаксиса инструкций помните следующее.

- Символ | означает выбор одного из нескольких вариантов, поэтому выражение `NULL | NOT NULL` означает, что нужно вводить либо `NULL`, либо `NOT NULL`.
- Ключевые слова или предложения, заключенные в квадратные скобки, `[например, так]`, являются необязательными.
- Рассматриваемый здесь синтаксис подходит почти для любой СУБД. Обратитесь к документации своей СУБД, чтобы узнать, нет ли каких-то изменений в синтаксисе.

ALTER TABLE

Инструкция `ALTER TABLE` предназначена для обновления схемы существующей таблицы. Чтобы создать новую таблицу, используйте инструкцию `CREATE TABLE`. За более детальной информацией обратитесь к уроку 17.

Ввод ▼

```
ALTER TABLE имя_таблицы
(
   ADD|DROP столбец тип_данных [NULL|NOT NULL]
[CONSTRAINTS],
   ADD|DROP столбец тип_данных [NULL|NOT NULL]
[CONSTRAINTS],
  ...
);
```

COMMIT

Инструкция COMMIT предназначена для сохранения результатов транзакции в базе данных (см. урок 20).

Ввод ▼

```
COMMIT [TRANSACTION];
```

CREATE INDEX

Инструкция CREATE INDEX предназначена для создания индекса одного или нескольких столбцов (см. урок 22).

Ввод ▼

```
CREATE INDEX название_индекса
ON имя_таблицы (столбец, ...);
```

CREATE PROCEDURE

Инструкция CREATE PROCEDURE предназначена для создания хранимых процедур (см. урок 19). В Oracle применяется иной синтаксис.

Ввод ▼

```
CREATE PROCEDURE имя_процедуры [аргументы] [опции]
AS
инструкция SQL;
```

CREATE TABLE

Инструкция CREATE TABLE предназначена для создания новых таблиц базы данных. Чтобы обновить схему уже существующей таблицы, используйте инструкцию ALTER TABLE. За более детальной информацией обратитесь к уроку 17.

Ввод ▼

```
CREATE TABLE имя_таблицы
(
   столбец тип_данных [NULL|NOT NULL] [CONSTRAINTS],
   столбец тип_данных [NULL|NOT NULL] [CONSTRAINTS],
 ...
);
```

CREATE VIEW

Инструкция CREATE VIEW предназначена для создания нового представления одной или нескольких таблиц (см. урок 18).

Ввод ▼

```
CREATE VIEW имя_представления AS
SELECT столбцы, ...
FROM таблицы, ...
[WHERE ...]
[GROUP BY ...]
[HAVING ...];
```

DELETE

Инструкция DELETE удаляет одну или несколько строк из таблицы (см. урок 16).

Ввод ▼

```
DELETE FROM имя_таблицы
[WHERE ...];
```

DROP

Инструкция DROP удаляет объекты из базы данных (таблицы, представления, индексы и т.п.). За более детальной информацией обратитесь к урокам 17 и 18.

Ввод ▼

```
DROP INDEX|PROCEDURE|TABLE|VIEW
имя_индекса|имя_процедуры|имя_таблицы|имя_представления;
```

INSERT

Инструкция INSERT добавляет в таблицу одну строку (см. урок 15).

Ввод ▼

```
INSERT INTO имя_таблицы [(столбцы, ...)]
VALUES(значения, ...);
```

INSERT SELECT

Инструкция INSERT SELECT добавляет результаты выполнения инструкции SELECT в таблицу (см. урок 15).

Ввод ▼

```
INSERT INTO имя_таблицы [(столбцы, ...)]
SELECT столбцы, ... FROM имя_таблицы, ...
[WHERE ...];
```

ROLLBACK

Инструкция ROLLBACK предназначена для отмены результатов транзакции (см. урок 20).

Ввод ▼

```
ROLLBACK [ TO точка_сохранения];
```

Ниже приведен альтернативный вариант.

Ввод ▼

```
ROLLBACK TRANSACTION;
```

SELECT

Инструкция SELECT предназначена для извлечения данных из одной или нескольких таблиц (или из представлений). За более детальной информацией обратитесь к урокам 2–4. (На всех уроках со 2-го по 14-й рассматриваются различные аспекты применения инструкции SELECT.)

Ввод ▼

```
SELECT имя_столбца, ...
FROM имя_таблицы, ...
[WHERE ...]
[UNION ...]
[GROUP BY ...]
[HAVING ...]
[ORDER BY ...];
```

UPDATE

Инструкция UPDATE обновляет одну или несколько строк в таблице (см. урок 16).

Ввод ▼

```
UPDATE имя_таблицы
SET имя_столбца = значение, ...
[WHERE ...];
```

ПРИЛОЖЕНИЕ Г

Типы данных в SQL

Как объяснялось на уроке 1, типы данных представляют собой основные правила, определяющие, какие данные могут храниться в столбцах и в каком виде эти данные в действительности хранятся.

Типы данных нужны по нескольким причинам.

- Они позволяют ограничить диапазон данных, которые могут храниться в столбце. Например, столбцы с данными числового типа будут принимать только числовые значения.
- Они позволяют более эффективно организовать хранение данных. Числовые значения и значения даты/времени могут храниться в более компактном виде, чем текстовые строки.
- Они позволяют изменять порядок сортировки. Если все данные трактуются как строки, то `1` предшествует `10`, а `10` предшествует `2`. (Строки сортируются в словарном порядке, по одному символу за раз, начиная слева.) Если выполняется числовая сортировка, то числа будут располагаться по возрастанию.

При разработке таблиц обращайте особое внимание на используемые в них типы данных. При наличии неправильных типов данных работа базы данных серьезно замедлится. Изменение типов данных для уже имеющихся и заполненных столбцов — задача нетривиальная. (Кроме того, при ее выполнении возможна потеря данных.)

В одном приложении невозможно дать исчерпывающую информацию по типам данных и способам их использования. Здесь рассмотрены лишь основные типы данных, рассказано, для чего они нужны, и указаны возможные проблемы совместимости.

ПРЕДУПРЕЖДЕНИЕ: **не существует двух одинаковых СУБД**

Об этом уже говорилось, но не лишним будет сказать еще раз. К сожалению, в разных СУБД используются отличающиеся типы данных. Даже если названия типов данных звучат одинаково, пониматься под одним и тем же типом данных в разных СУБД может не одно и то же. Обязательно обратитесь к документации своей СУБД и выясните, какие в точности типы данных она поддерживает и каким образом.

Строковые типы данных

Чаще всего используются данные строковых типов. К ним относятся хранимые в базах данных строки, например имена, адреса, номера телефонов и почтовые индексы. В основном строки бывают двух типов: фиксированной и переменной длины (табл. Г.1).

Строки фиксированной длины могут состоять из фиксированного числа символов, и это число определяется при создании таблицы. Например, можно разрешить ввод 30 символов в столбец, предназначенный для хранения имен, или 11 символов в столбец с номером карточки социального страхования. В столбцы для строк фиксированной длины нельзя вводить больше символов, чем разрешено. База данных выделяет для хранения ровно столько места, сколько указано. Так, если строка Иван сохраняется в столбце имен, рассчитанном на ввод 30 символов, будет сохранено ровно 30 символов (в случае необходимости текст дополняется пробелами или нулями).

В строках переменной длины можно хранить столько символов, сколько необходимо (максимальное значение ограничивается типом данных и конкретной СУБД). Некоторые типы данных переменной длины имеют ограничение снизу (фиксированное значение минимальной длины). Другие ограничений не имеют. В любом случае сохраняются только заданные символы (и никаких дополнительных).

Если строки переменной длины обладают такой гибкостью, то зачем нужны строки фиксированной длины? Ответ прост: для повышения производительности. СУБД способна сортировать столбцы с данными фиксированной длины и манипулировать ими намного быстрее, чем в случае данных переменной длины. Кроме того, многие СУБД не способны индексировать столбцы с данными переменой длины (или переменную часть столбца). (Индексы рассматривались на уроке 22.)

ТАБЛИЦА Г.1. **Строковые типы данных**

Тип данных	Описание
CHAR	Строка фиксированной длины, состоящая из 1–255 символов. Ее размер должен быть определен на этапе создания таблицы
NCHAR	Разновидность типа данных CHAR, разработанная с целью поддержки многобайтовых символов или символов Unicode (точная спецификация зависит от реализации)
NVARCHAR	Разновидность типа данных TEXT, разработанная с целью поддержки многобайтовых символов или символов Unicode (точная спецификация зависит от реализации)
TEXT (также называется LONG, MEMO или VARCHAR)	Текст переменной длины

СОВЕТ: **использование кавычек**

Независимо от типа строковых данных строка всегда должна быть заключена в одинарные кавычки.

ПРЕДУПРЕЖДЕНИЕ: **когда числовые значения не являются таковыми**

Может показаться, будто номера телефонов и почтовые индексы должны храниться в числовых полях (ведь они содержат только числовые данные), но поступать так нецелесообразно. Если вы сохраните почтовый индекс 01234 в числовом поле, будет сохранено число 1234, и вы потеряете одну цифру.

Основное правило таково: если число используется в вычислениях (итоговых сумм, средних значений и т.п.), его следует хранить в столбце, предназначенном для числовых данных. Если же оно используется в качестве строкового литерала (пусть он и состоит только из цифр), его место — в столбце с данными строкового типа.

Числовые типы данных

Числовые типы данных предназначены для хранения чисел. В большинстве СУБД поддерживаются различные числовые типы данных, каждый из которых рассчитан на хранение чисел определенного диапазона. Очевидно, что чем шире поддерживаемый диапазон, тем больше нужно места для хранения числа. Кроме того, некоторые числовые типы данных поддерживают использование десятичных чисел (и дробей), а другие — только целые числа. В табл. Г.2 представлены наиболее часто используемые числовые типы данных. Не все СУБД следуют соглашениям о наименовании и описаниям, приведенным в таблице.

ТАБЛИЦА Г.2. **Числовые типы данных**

Типы данных	Описание
`BIT`	Одноразрядное значение, `0` или `1`; используется в основном для битовых флагов
`DECIMAL` (также называется `NUMERIC`)	Значения с фиксированной или плавающей запятой различной степени точности
`FLOAT` (также называется `NUMBER`)	Значения с плавающей запятой
`INT` (также называется `INTEGER`)	4-байтовые целые значения; поддерживаются числа от -2147483648 до 2147483647
`REAL`	4-байтовые значения с плавающей запятой
`SMALLINT`	2-байтовые целые значения; поддерживаются числа от -32768 до 32767
`TINYINT`	1-байтовые целые значения; поддерживаются числа от 0 до 255

СОВЕТ: **кавычки не используются**

В отличие от строковых типов данных, числа никогда не заключаются в кавычки.

СОВЕТ: **денежные типы данных**

В большинстве СУБД поддерживается особый числовой тип данных для хранения денежных значений. Обычно он называется MONEY или CURRENCY. Как правило, такие типы данных относятся к типу DECIMAL, но со специфическими диапазонами, делающими их удобными для хранения денежных значений.

Типы данных даты и времени

Все СУБД поддерживают типы данных, предназначенные для хранения значений даты и времени (табл. Г.3). Аналогично числовым типам, в большинстве СУБД имеется несколько типов данных даты и времени, каждый со своим диапазоном и степенью точности.

ТАБЛИЦА Г.3. **Типы данных даты и времени**

Тип данных	Описание
DATE	Значения даты
DATETIME (также называется TIMESTAMP)	Значения даты и времени
SMALLDATETIME	Значения даты и времени с точностью до минуты (без значений секунд или миллисекунд)
TIME	Значения времени

ПРЕДУПРЕЖДЕНИЕ: **формат даты**

Не существует стандартного способа указания даты, который подходил бы для любой СУБД. В большинстве реализаций приемлем формат вида 2015-12-30 или Dec 30th, 2015, но даже эти значения могут оказаться проблемой для некоторых СУБД. Обязательно обратитесь к документации своей СУБД и узнайте список распознаваемых ею форматов даты.

СОВЕТ: **значения даты в ODBC**

Поскольку в каждой СУБД применяется свой формат представления даты, в ODBC введен собственный формат, который подходит для любой СУБД при работе с ODBC. Формат ODBC выглядит так: `{d '2015-12-30'}` для значений дат, `{t '21:46:29'}` для значений времени и `{ts '2015-12-30 21:46:29'}` для значений даты и времени. Если вы выполняете SQL-запросы через ODBC, убедитесь в том, что значения даты и времени отформатированы соответствующим образом.

Бинарные типы данных

Бинарные типы данных относятся к наименее совместимым (и реже всего используемым) типам данных. В отличие от всех других типов данных, рассмотренных нами до сих пор и предназначенных для весьма конкретного применения, бинарные типы могут содержать любые данные, даже информацию в двоичном виде, в частности графические изображения, мультимедийные объекты и документы текстового процессора (табл. Г.4).

ТАБЛИЦА Г.4. **Бинарные типы данных**

Тип данных	Описание
`BINARY`	Двоичные данные фиксированной длины (максимальная длина может быть от 255 до 8000 байт, в зависимости от реализации)
`LONG RAW`	Двоичные данные переменной длины (до 2 Гбайт)
`RAW` (в некоторых реализациях называется `BINARY`)	Двоичные данные фиксированной длины (до 255 байт)
`VARBINARY`	Двоичные данные переменной длины (обычно максимальная длина варьируется от 255 до 8000 байт, в зависимости от реализации)

ПРИМЕЧАНИЕ: **сравнение типов данных**

Чтобы увидеть реальный пример и провести сравнение типов данных в различных СУБД, воспользуйтесь сценариями создания демонстрационных таблиц (см. приложение А). Путем сравнения сценариев, предназначенных для различных СУБД, вы воочию убедитесь в том, насколько сложна задача согласования типов данных.

ПРИЛОЖЕНИЕ Д

Зарезервированные слова SQL

В инструкциях SQL широко применяются ключевые слова, которые считаются зарезервированными. Нужно внимательно следить за тем, чтобы они не использовались в качестве имен баз данных, таблиц, столбцов и других объектов.

В данном приложении содержится перечень зарезервированных слов, наиболее часто встречающихся в основных СУБД. Обратите внимание на следующие моменты.

- Ключевые слова сильно зависят от конкретной СУБД, поэтому не все приведенные ниже слова используются во всех СУБД.
- Во многих СУБД имеется расширенный перечень зарезервированных слов SQL, включающий термины, которые специфичны для конкретной реализации языка. Многие из таких слов не представлены ниже.
- Чтобы обеспечить совместимость и переносимость базы данных, следует избегать применения любых ключевых слов, даже тех, которые не являются зарезервированными в конкретной СУБД.

```
ABORT           ABSOLUTE        ACTION
ACTIVE          ADD             AFTER
ALL             ALLOCATE        ALTER
ANALYZE         AND             ANY
ARE             AS              ASC
ASCENDING       ASSERTION       AT
AUTHORIZATION   AUTO            AUTO-INCREMENT
AUTOINC         AVG             BACKUP
BEFORE          BEGIN           BETWEEN
```

BIGINT	BINARY	BIT
BLOB	BOOLEAN	BOTH
BREAK	BROWSE	BULK
BY	BYTES	CACHE
CALL	CASCADE	CASCADED
CASE	CAST	CATALOG
CHANGE	CHAR	CHARACTER
CHARACTER_LENGTH	CHECK	CHECKPOINT
CLOSE	CLUSTER	CLUSTERED
COALESCE	COLLATE	COLUMN
COLUMNS	COMMENT	COMMIT
COMMITTED	COMPUTE	COMPUTED
CONDITIONAL	CONFIRM	CONNECT
CONNECTION	CONSTRAINT	CONSTRAINTS
CONTAINING	CONTAINS	CONTAINSTABLE
CONTINUE	CONTROLROW	CONVERT
COPY	COUNT	CREATE
CROSS	CSTRING	CUBE
CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
DATABASE	DATABASES	DATE
DATETIME	DAY	DBCC
DEALLOCATE	DEBUG	DEC
DECIMAL	DECLARE	DEFAULT
DELETE	DENY	DESC
DESCENDING	DESCRIBE	DISCONNECT
DISK	DISTINCT	DISTRIBUTED
DIV	DO	DOMAIN
DOUBLE	DROP	DUMMY
DUMP	ELSE	ELSEIF
ENCLOSED	END	ERRLVL
ERROREXIT	ESCAPE	ESCAPED
EXCEPT	EXCEPTION	EXEC
EXECUTE	EXISTS	EXIT
EXPLAIN	EXTEND	EXTERNAL
EXTRACT	FALSE	FETCH
FIELD	FIELDS	FILE
FILLFACTOR	FILTER	FLOAT
FLOPPY	FOR	FORCE
FOREIGN	FOUND	FREETEXT

FREETEXTTABLE
FROM
FULL
FUNCTION
GENERATOR
GET
GLOBAL
GO
GOTO
GRANT
GROUP
HAVING
HOLDLOCK
HOUR
IDENTITY
IF
IN
INACTIVE
INDEX
INDICATOR
INFILE
INNER
INOUT
INPUT
INSENSITIVE
INSERT
INT
INTEGER
INTERSECT
INTERVAL
INTO
IS
ISOLATION
JOIN
KEY
KILL
LANGUAGE
LAST
LEADING
LEFT
LENGTH
LEVEL
LIKE
LIMIT
LINENO
LINES
LISTEN
LOAD
LOCAL
LOCK
LOGFILE
LONG
LOWER
MANUAL
MATCH
MAX
MERGE
MESSAGE
MIN
MINUTE
MIRROREXIT
MODULE
MONEY
MONTH
MOVE
NAMES
NATIONAL
NATURAL
NCHAR
NEXT
NEW
NO
NOCHECK
NONCLUSTERED
NONE
NOT
NULL
NULLIF
NUMERIC
OF
OFF
OFFSET
OFFSETS
ON
ONCE
ONLY
OPEN
OPTION
OR
ORDER
OUTER
OUTPUT
OVER
OVERFLOW
OVERLAPS
PAD
PAGE
PAGES
PARAMETER
PARTIAL
PASSWORD
PERCENT
PERM
PERMANENT
PIPE
PLAN
POSITION
PRECISION
PREPARE
PRIMARY
PRINT
PRIOR
PRIVILEGES
PROC
PROCEDURE
PROCESSEXIT
PROTECTED
PUBLIC
PURGE
RAISERROR

READ
READTEXT
REAL
REFERENCES
REGEXP
RELATIVE
RENAME
REPEAT
REPLACE
REPLICATION
REQUIRE
RESERV
RESERVING
RESET
RESTORE
RESTRICT
RETAIN
RETURN
RETURNS
REVOKE
RIGHT
ROLLBACK
ROLLUP
ROWCOUNT
RULE
SAVE
SAVEPOINT
SCHEMA
SECOND
SECTION
SEGMENT
SELECT
SENSITIVE
SEPARATOR
SEQUENCE
SESSION_USER
SET
SETUSER
SHADOW
SHARED
SHOW
SHUTDOWN
SINGULAR
SIZE
SMALLINT
SNAPSHOT
SOME
SORT
SPACE
SQL
SQLCODE
SQLERROR
STABILITY
STARTING
STARTS
STATISTICS
SUBSTRING
SUM
SUSPEND
TABLE
TABLES
TAPE
TEMP
TEMPORARY
TEXT
TEXTSIZE
THEN
TIME
TIMESTAMP
TO
TOP
TRAILING
TRAN
TRANSACTION
TRANSLATE
TRIGGER
TRIM
TRUE
TRUNCATE
UNCOMMITTED
UNION
UNIQUE
UNTIL
UPDATE
UPDATETEXT
UPPER
USAGE
USE
USER
USING
VALUE
VALUES
VARCHAR
VARIABLE
VARYING
VERBOSE
VIEW
VOLUME
WAIT
WAITFOR
WHEN
WHERE
WHILE
WITH
WORK
WRITE
WRITETEXT
XOR
YEAR
ZONE

Предметный указатель

A
Access, 247
ASP, 248
ASP.NET, 249

C
ColdFusion, 246

D
DB2, 246

M
MariaDB, 247
Microsoft Query, 249
MySQL, 251

N
NOT NULL, 176
NULL, 52; 176

O
ODBC, 256
OpenOffice Base, 245
Oracle, 252
Oracle Express, 253

P
PHP, 254
PostgreSQL, 254

S
SQL Server, 250
SQLite, 255

Б
База данных, 20
Безопасность, 235

В
Внешнее объединение, 140
Внешний ключ, 169; 226
Внутреннее объединение, 130
Вычисляемое поле, 73; 74; 194

Г
Группировка, 105
Групповой символ, 32

Д
Дата, 179
Декартово произведение, 128
Добавление
  извлеченных данных, 160
  нескольких строк, 162
  полных строк, 155
  части строки, 159

Е
Естественное объединение, 139

З
Запись, 24
Запрос, 115
  комбинированный, 147
Значение по умолчанию, 178

И
Индекс, 230
Инструкция
  ALTER TABLE, 179; 225; 259
  BEGIN TRANSACTION, 210
  CLOSE, 221
  COMMIT, 211; 260
  CREATE INDEX, 232; 260
  CREATE PROCEDURE, 201; 260
  CREATE TABLE, 173; 225; 261
  CREATE TRIGGER, 234
  CREATE VIEW, 189; 261
  DECLARE, 203; 217
  DELETE, 168; 261
  DROP, 262
  DROP TABLE, 182
  DROP VIEW, 189
  EXECUTE, 200
  FETCH, 218
  GRANT, 236

INSERT, 155; 262
INSERT SELECT, 160; 262
OPEN CURSOR, 218
RENAME, 183
REVOKE, 236
ROLLBACK, 211; 262
SAVE TRANSACTION, 213
SAVEPOINT, 213
SELECT, 27; 263
порядок предложений, 114
SELECT INTO, 162
SET TRANSACTION, 210
START TRANSACTION, 210
TRUNCATE TABLE, 170
UPDATE, 165; 263
Источник данных, 257
Итоговая функция, 94; 143

К

Кавычки, 51
Каскадное удаление, 227
Ключевое слово, 27; 273
ALL, 101; 107
AND, 55
AS, 78; 136
ASC, 45
BETWEEN, 51
CONSTRAINT, 225
DEFAULT, 178
DESC, 43
DISTINCT, 33; 101
IN, 60
INTO, 156
LIKE, 65
NOT, 61
OR, 57
REFERENCES, 227
TOP, 34
UNIQUE, 228
Ключ
внешний, 169; 226
первичный, 24; 124; 224
Комбинированный запрос, 147
Комментарий, 37; 204
Конкатенация, 75
Критерий отбора, 47
Курсор, 215
закрытие, 221
открытие, 218
создание, 217

М

Математическая операция, 80
Метасимвол, 65
знак процента, 66
квадратные скобки, 70
символ подчеркивания, 69

Н

Неявная фиксация, 211

О

Обновление
данных, 165
таблиц, 179
Объединение, 123
внешнее, 140
внутреннее, 130
естественное, 139
нескольких таблиц, 131
перекрестное, 130
полное внешнее, 142
Ограничение, 223
на значения столбца, 229
уникальности, 228
Оператор
EXCEPT, 154
INTERSECT, 154
UNION, 148
UNION ALL, 152
математический, 81
условный, 49
Откат транзакции, 209

П
Первичный ключ, 24; 124; 224
Переименование таблиц, 183
Перекрестное объединение, 130
Переносимый код, 84
Подзапрос, 115
  в качестве вычисляемого поля, 119
Поле, 74
Полное внешнее объединение, 142
Пользовательский тип данных, 230
Предикат, 66
Предложение, 40
  GROUP BY, 106
  HAVING, 108
  LIMIT, 35
  ON, 131
  ORDER BY, 40; 111
  SET, 166
  VALUES, 156
  WHERE, 47; 55
Представление, 185
  создание, 189
  удаление, 189
Псевдоним, 78; 135

Р
Реляционная таблица, 123

С
СУБД, 20
Самообъединение, 137
Скобки, 59
Соединение, 147
Сортировка, 39; 111
  в указанном направлении, 43
  по невыбранным столбцам, 41; 43
  по нескольким столбцам, 41
  по положению столбца, 42
Ссылочная целостность, 126
Столбец, 22; 74
  производный, 80
Строка, 23
Схема, 21

Т
Таблица, 20
  копирование, 163
  обновление, 179
  переименование, 183
  реляционная, 123
  создание, 173
  удаление, 182
Тип данных, 23; 265
  бинарный, 270
  даты и времени, 269
  денежный, 269
  пользовательский, 230
  строковый, 266
  числовой, 268
Точка сохранения, 209; 212
Транзакция, 207
  отмена, 211
Триггер, 233

У
Удаление
  данных, 168
  таблиц, 182
Условие фильтрации, 47

Ф
Фиксация транзакции, 209
  неявная, 211
Функция, 83
  AVG(), 94
  COUNT(), 96
  DATE_PART(), 89
  DATEPART(), 89
  LTRIM(), 78
  MAX(), 97
  MIN(), 98
  RTRIM(), 77

SOUNDEX(), 87
SUM(), 99
to_char(), 90
to_date(), 90
to_number(), 90
TRIM(), 78
UPPER(), 85
YEAR(), 90

Х

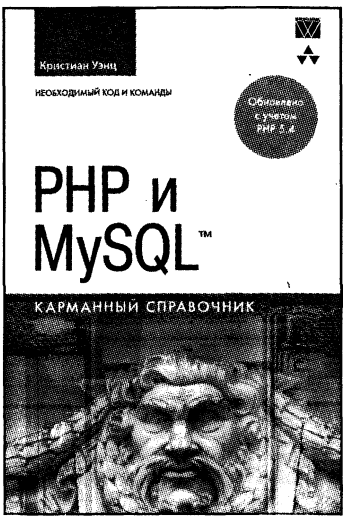
Хранимая процедура, 197
  создание, 201

Ш

Шаблон поиска, 66

PHP и MySQL™ Карманный справочник

Кристиан Уэнц



Эта книга, не претендуя на полноту описания всех возможностей, предоставляемых языком PHP, предлагает для рассмотрения темы, с которыми PHP-программист сталкивается практически ежедневно. Автор приложил максимум усилий, чтобы этот карманный справочник соответствовал последним стандартам языка PHP, и акцентировал внимание на новых возможностях версий PHP 5.3 и 5.4. Поскольку СУБД MySQL остается повсеместно принятым стандартом баз данных для приложений на PHP, она заслужила отдельной главы. Все листинги из книги доступны для загрузки, а их код протестирован на таких платформах, как Linux, Windows, Mac OS X и Solaris.

www.williamspublishing.com

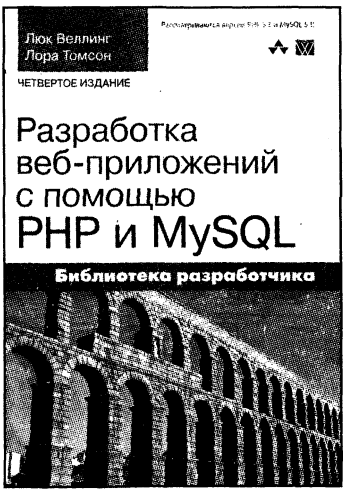
ISBN 978-5-8459-1866-6 **в продаже**

РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ PHP И MYSQL

БИБЛИОТЕКА РАЗРАБОТЧИКА

ЧЕТВЕРТОЕ ИЗДАНИЕ

Люк Веллинг
Лора Томсон



Эта книга предназначена для тех, кто знаком с основами HTML и ранее разрабатывал программы на современных языках программирования, но, возможно, не занимался программированием для веб или не использовал реляционные базы данных. В ней подробно описано применение последних версий PHP и MySQL для построения крупных коммерческих веб-сайтов. Основное внимание в книге уделено реальным приложениям. Здесь рассматриваются как простые интерактивные системы приема заказов, так и различные аспекты электронных систем продажи и безопасности во взаимосвязи с созданием реального веб-сайта. Подробно описаны все стадии разработки типовых проектов на PHP и MySQL, в числе которых служба веб-почты, приложение поддержки веб-форумов и электронный книжный магазин. Книга ориентирована на профессиональных разработчиков, но будет полезной и начинающим.

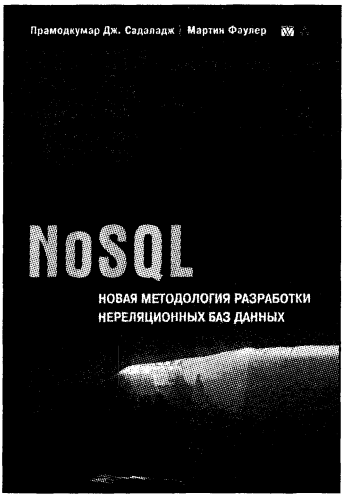
www.williamspublishing.com

ISBN 978-5-8459-1574-0

в продаже

NoSQL: новая методология разработки нереляционных баз данных

Прамодкумар Дж. Садаладж
Мартин Фаулер



Необходимость обрабатывать все большие объемы данных является одним из факторов, стимулирующих появление альтернатив реляционным базам данных, использующим язык SQL. Одной из таких альтернатив является технология NoSQL.

В книге Фаулера рассматриваются основные концепции NoSQL, включая неструктурированные модели данных, агрегаты, новые модели распределения, теорему CAP и отображение–свертку, а также исследованы архитектурные и проектные вопросы, связанные с реализацией баз данных NoSQL.

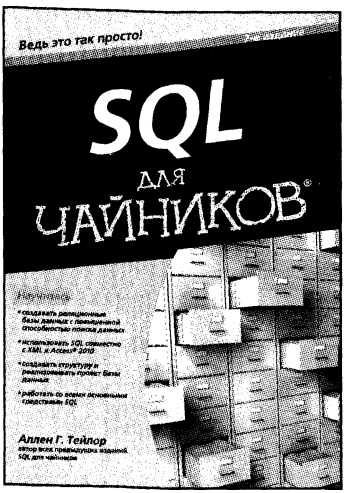
Книга предназначена для разработчиков баз данных, корпоративных приложений, а также для студентов.

www.williamspublishing.com

ISBN 978-5-8459-1829-1 **в продаже**

SQL ДЛЯ ЧАЙНИКОВ
7-е издание

Аллен Г. Тейлор



Эта книга предназначена для тех, кто хочет повысить свой уровень работы с базами данных с помощью языка структурированных запросов — SQL. Вы освоите основы реляционных баз данных и языка SQL, научитесь проектировать базы данных, заполнять их информацией и извлекать ее, используя расширенные возможности языка. Отдельные части книги посвящены вопросам защиты информации в базах данных и обработки ошибок.

Даже если вам никогда не приходилось ранее разрабатывать системы хранения данных, эта книга поможет при работе с информацией использовать самые современные технологии.

www.dialektika.com

ISBN 978-5-8459-1673-0 **в продаже**

C# 5.0 И ПЛАТФОРМА .NET 4.5
ДЛЯ ПРОФЕССИОНАЛОВ

К. Нейгел, Б. Ивьен,
Д. Глинн, К. Уотсон,
М. Скиннер



www.dialektika.com

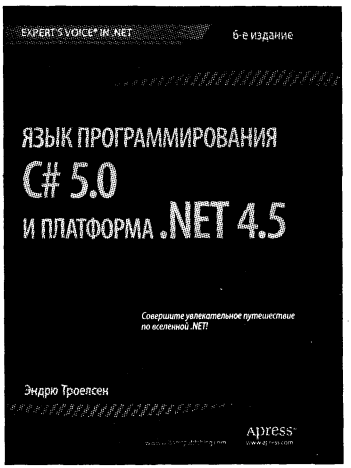
Книга известных специалистов в области разработки приложений с использованием .NET Framework посвящена программированию на языке C# 5.0 в среде .NET Framework 4.5. Ее отличает простой и доступный стиль изложения, изобилие примеров и множество рекомендаций по написанию высококачественных программ. Подробно рассматриваются такие вопросы, как основы языка программирования C#, организация среды .NET, работа с данными, написание Windows- и веб-приложений, взаимодействие через сеть, создание веб-служб и многое другое. Немалое внимание уделено проблемам безопасности и сопровождения кода. Тщательно подобранный материал позволит без труда разобраться с тонкостями использования ASP.NET и построения веб-страниц, а также научиться разрабатывать приложения для Windows 8 и WinRT. Читатели ознакомятся с работой в Visual Studio, а также с применением различных технологий, встроенных в .NET.

Книга рассчитана на программистов разной квалификации, а также будет полезна для студентов и преподавателей дисциплин, связанных с программированием и разработкой для .NET.

ISBN 978-5-8459-1850-5 **в продаже**

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 5.0 И ПЛАТФОРМА .NET 4.5
6-Е ИЗДАНИЕ

Эндрю Троелсен



www.williamspublishing.com

Новое издание этой книги было полностью пересмотрено и переписано с учетом последних изменений в спецификации языка C# и дополнений платформы .NET Framework. Отдельные главы посвящены важным новым средствам, которые превращают .NET Framework 4.5 в самое передовое решение для корпоративных приложений. Помимо этого, рассмотрены все ключевые возможности языка C#, как старые, так и новые, что позволило обрести популярность предыдущим изданиям этой книги (материал покрывает все темы, начиная с обобщений и кончая pLINQ). Основное назначение книги — служить исчерпывающим руководством по языку программирования C# и ключевым аспектам платформы .NET (сборкам, удаленному взаимодействию, Windows Forms, Web Forms, ADO.NET, веб-службам XML и т.д.).

ISBN 978-5-8459-1814-7 **в продаже**

Освой самостоятельно

SQL

4-е издание

за 10 минут

В книге предлагаются простые и практичные решения для тех, кто хочет быстро получить результат.

Проработав все 22 урока, на каждый из которых придется затратить не более 10 минут, вы узнаете обо всем, что необходимо для практического применения SQL.

Приведенные в книге примеры подходят для IBM DB2, Microsoft Access, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, MariaDB и Apache OpenOffice Base.

Наглядные примеры помогут понять, как структурируются инструкции SQL.

Советы подскажут короткие пути к решениям.

Предупреждения помогут избежать распространенных ошибок.

Примечания предоставят дополнительные разъяснения.

Бен Форта — директор департамента разработки в компании Adobe Systems. Автор множества бестселлеров, включая книги по базам данных, SQL и ColdFusion. Имеет большой опыт в проектировании баз данных и разработке приложений.



www.williamspublishing.com

SAMS
informit.com/sams

Что можно узнать за 10 минут:

- Основные инструкции SQL
- Создание сложных SQL-запросов с множеством предложений и операторов
- Извлечение, сортировка и форматирование данных
- Получение конкретных данных с помощью различных методов фильтрации
- Применение итоговых функций для получения сводных данных
- Объединение реляционных таблиц
- Добавление, обновление и удаление данных
- Создание и изменение таблиц
- Работа с представлениями, хранимыми процедурами и многое другое

Категория: программирование/базы данных
Предмет рассмотрения: SQL
Уровень: начальный — средний

ISBN 978-5-8459-1858-1

