Бен Форта

SQL

5-е издание

за 10 минут

Sams Teach Yourself

SQL

in 10 minutes

Fifth Edition

Ben Forta

SAMS
221 River Street, Hoboken, NJ 07030

SQL

за 10 минут

5-е издание

Бен Форта

Москва ◆ Санкт-Петербург
2021

ББК 32.973.26-018.2.75
Ф80
УДК 004.655.3 (075.8)

ООО "Диалектика"

Зав. редакцией *В.Р. Гинзбург*

Перевод с английского и редакция *В.Р. Гинзбурга*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info.dialektika@gmail.com, http://www.dialektika.com

Форта, Бен.

Ф80 SQL за 10 минут, 5-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2021. — 352 с. : ил. — Парал. тит. англ.

ISBN 978-5-907365-67-4 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Бен Форта

SQL за 10 минут
5-е издание

Подписано в печать 28.07.2021. Формат 84x108/32
Усл. печ. л. 18,5. Уч.-изд. л. 11,8
Тираж 1000 экз. Заказ № 5665

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург,
Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907365-67-4 (рус.)

ISBN 978-0-13-518279-6 (англ.)

Оглавление

Об авторе 13
Благодарности 14
Введение 15
Урок 1. Основы SQL 21
Урок 2. Извлечение данных из таблиц 33
Урок 3. Сортировка полученных данных 49
Урок 4. Фильтрация данных 59
Урок 5. Расширенная фильтрация данных 69
Урок 6. Фильтрация с использованием метасимволов 81
Урок 7. Создание вычисляемых полей 93
Урок 8. Функции обработки данных 105
Урок 9. Итоговые вычисления 119
Урок 10. Группирование данных 133
Урок 11. Подзапросы 147
Урок 12. Соединение таблиц 159
Урок 13. Создание расширенных соединений 175
Урок 14. Комбинированные запросы 189
Урок 15. Добавление данных 201
Урок 16. Обновление и удаление данных 213
Урок 17. Создание таблиц и работа с ними 221
Урок 18. Представления 235
Урок 19. Хранимые процедуры 251
Урок 20. Обработка транзакций 263
Урок 21. Курсоры 273
Урок 22. Расширенные возможности SQL 281
Приложение А. Сценарии демонстрационных таблиц 299
Приложение Б. Синтаксис инструкций SQL 309
Приложение В. Типы данных в SQL 315
Приложение Г. Зарезервированные слова SQL 323
Приложение Д. Ответы на упражнения 329
Предметный указатель 345

Содержание

Об авторе 13

Благодарности 14

Введение 15

Для кого предназначена эта книга 16
СУБД, рассмотренные в книге 16
Файлы примеров 17
Условные обозначения 17
Ждем ваших отзывов! 19

Урок 1. Основы SQL 21

Терминология баз данных 21
Что такое SQL 28
Попробуйте сами 29
Резюме 31

Урок 2. Извлечение данных из таблиц 33

Инструкция `SELECT` 33
Извлечение отдельных столбцов 34
Извлечение нескольких столбцов 37
Извлечение всех столбцов 38
Извлечение уникальных строк 39
Ограничение результатов запроса 41
Использование комментариев 45
Резюме 47
Упражнения 48

Урок 3. Сортировка полученных данных 49

Сортировка записей 49
Сортировка по нескольким столбцам 51
Сортировка по положению столбца 53

Указание направления сортировки 54
Резюме 57
Упражнения 58

Урок 4. Фильтрация данных 59

Предложение WHERE 59
Операторы в предложении WHERE 61
Резюме 67
Упражнения 68

Урок 5. Расширенная фильтрация данных 69

Комбинирование предложений WHERE 69
Оператор IN 75
Оператор NOT 77
Резюме 79
Упражнения 80

Урок 6. Фильтрация с использованием метасимволов 81

Оператор LIKE 81
Советы по использованию метасимволов 89
Резюме 90
Упражнения 91

Урок 7. Создание вычисляемых полей 93

Что такое вычисляемые поля 93
Конкатенация полей 95
Выполнение арифметических вычислений 101
Резюме 104
Упражнения 104

Урок 8. Функции обработки данных 105

Что такое функция 105
Применение функций 107
Резюме 116
Упражнения 117

Урок 9. Итоговые вычисления 119

Итоговые функции 119
Итоговые вычисления для уникальных значений 128
Комбинирование итоговых функций 130
Резюме 131
Упражнения 131

Урок 10. Группирование данных 133

Принципы группирования данных 133
Создание групп 134
Фильтрация по группам 137
Группирование и сортировка 141
Порядок предложений в инструкции SELECT 143
Резюме 144
Упражнения 145

Урок 11. Подзапросы 147

Что такое подзапросы 147
Фильтрация с помощью подзапросов 147
Использование подзапросов в качестве вычисляемых полей 152
Резюме 156
Упражнения 157

Урок 12. Соединение таблиц 159

Что такое соединение 159
Создание соединения 163
Резюме 172
Упражнения 173

Урок 13. Создание расширенных соединений 175

Использование псевдонимов таблиц 175
Другие виды соединений 177

Использование соединений совместно с итоговыми функциями 184
Правила создания соединений 186
Резюме 187
Упражнения 188

Урок 14. Комбинированные запросы 189

Что такое комбинированные запросы 189
Создание комбинированных запросов 190
Резюме 198
Упражнения 199

Урок 15. Добавление данных 201

Способы добавления данных 201
Копирование данных из одной таблицы в другую 210
Резюме 212
Упражнения 212

Урок 16. Обновление и удаление данных 213

Обновление данных 213
Удаление данных 216
Советы по обновлению и удалению данных 218
Резюме 220
Упражнения 220

Урок 17. Создание таблиц и работа с ними 221

Создание таблиц 221
Обновление таблиц 228
Удаление таблиц 231
Переименование таблиц 232
Резюме 233
Упражнения 233

Урок 18. Представления 235

Что такое представления 235
Создание представлений 240
Резюме 248
Упражнения 249

Урок 19. Хранимые процедуры 251

Что такое хранимые процедуры 251
Зачем нужны хранимые процедуры 252
Выполнение хранимых процедур 255
Создание хранимых процедур 256
Резюме 262

Урок 20. Обработка транзакций 263

Что такое транзакции 263
Управление транзакциями 266
Резюме 272

Урок 21. Курсоры 273

Что такое курсоры 273
Работа с курсорами 275
Резюме 280

Урок 22. Расширенные возможности SQL 281

Что такое ограничения 281
Что такое индексы 290
Что такое триггеры 294
Безопасность баз данных 296
Резюме 297

Приложение А. Сценарии демонстрационных таблиц 299

Демонстрационные таблицы 299
Получение демонстрационных таблиц 304

Приложение Б. Синтаксис инструкций SQL 309

ALTER TABLE 309
COMMIT 310
CREATE INDEX 310
CREATE PROCEDURE 310
CREATE TABLE 311
CREATE VIEW 311
DELETE 312
DROP 312
INSERT 312
INSERT SELECT 312
ROLLBACK 313
SELECT 313
UPDATE 314

Приложение В. Типы данных в SQL 315

Строковые типы данных 316
Числовые типы данных 318
Типы данных даты и времени 320
Бинарные типы данных 321

Приложение Г. Зарезервированные слова SQL 323

Приложение Д. Ответы на упражнения 329

Предметный указатель 345

Об авторе

Бен Форта — директор департамента обучающих решений в компании Adobe Systems. За его плечами более 30 лет работы в компьютерной индустрии, включая разработку продуктов, их поддержку и распространение, а также обучение пользователей. Бен — автор множества бестселлеров, включая книги по MySQL, SQL Server T-SQL и Oracle PL/SQL, а также книги по регулярным выражениям, Java, Windows и других темам. Имеет большой опыт проектирования баз данных, часто читает лекции и пишет статьи, посвященные веб-технологиям. Живет в г. Оук-Парк, штат Мичиган, со своей женой Марси и детьми. Можете написать Бену письмо по адресу `ben@forta.com` или же посетить его сайт `http://forta.com`.

Благодарности

Хочу поблагодарить сотрудников издательства Sams за многолетнюю помощь и поддержку моих проектов. За последние 20 лет мы выпустили более 40 книг, но именно этот карманный справочник стал моей любимой книгой, и я хочу сказать спасибо всем тем, кто дал мне творческую свободу в реализации задуманного.

Спасибо пользователям Amazon.com, которые в отзывах на книгу предложили добавить в нее упражнения, что и было сделано в данном издании.

Я благодарен тысячам читателей, приславших отзывы на предыдущие издания книги. К счастью, в основном это были положительные отклики, что для меня очень приятно. Все изменения, внесенные в новое издание, стали прямым следствием пожеланий читателей, которые я стараюсь учитывать.

Не могу не поблагодарить десятки колледжей и университетов, которые используют мою книгу в своих учебных программах. Доверие, оказанное профессорами и преподавателями, вдохновляет и воодушевляет меня.

Наконец, хочу поблагодарить полмиллиона читателей, купивших предыдущие издания книги, что сделало ее не только моим главным бестселлером, но и самой продаваемой книгой по базам данных в мире. Ваша поддержка — самая большая награда за мои труды.

Бен Форта

Введение

SQL — самый популярный язык баз данных. Не важно, кто вы — разработчик приложений, системный администратор, веб-дизайнер или пользователь Microsoft Office. Хорошее 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 хорошими. 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, авторы большинства книг излагали 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 студентам, я вложил весь свой опыт преподавания в книгу, которую вы держите в руках. Она поможет вам быстро освоить SQL. Мы начнем с простых запросов на выборку данных, постепенно переходя к более сложным темам, таким как соединения, подзапросы, хранимые процедуры, курсоры, триггеры и ограничения. Обучение будет происходить методично, систематично и просто — на каждый урок вам потребуется не более 10 минут.

На сегодняшний день предыдущие издания книги проданы суммарным тиражом более полумиллиона экземпляров. Они переведены на десятки языков и помогают пользователям по всему миру.

В новом издании добавились упражнения к урокам 2–18. Они дают возможность закрепить полученные знания и применить их для решения задач, не рассмотренных на уроке. Все ответы на упражнения приведены в конце книги.

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 посторонней помощи.

СУБД, рассмотренные в книге

В большинстве случаев SQL-запросы, рассматриваемые в книге, можно выполнять в любой СУБД (система управления базами данных). Но поскольку не все реализации языка идентичны, особое внимание в книге будет уделено следующим СУБД (по мере необходимости будут даваться инструкции и примечания для каждой из них):

- IBM Db2 (включая Db2 on Cloud);
- Microsoft SQL Server (включая Microsoft SQL Server Express);
- MariaDB;

- MySQL;
- Oracle (включая Oracle Express);
- PostgreSQL;
- SQLite.

Файлы примеров

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

`http://forta.com/books/0135182794`

Также архив примеров книги доступен на сайте издательства "Диалектика":

`http://go.dialektika.com/SQL5`

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.
```

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

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 даются определения основных терминов.

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 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.dialektika@gmail.com`

WWW: `http://www.williamspublishing.com`

УРОК 1

Основы SQL

На этом уроке вы узнаете, что такое SQL и что такое запросы к базам данных.

Терминология баз данных

Раз вы читаете книгу по SQL, значит, вам так или иначе необходимо работать с базами данных. SQL предназначен для создания запросов к базам данных, поэтому, прежде чем перейти к рассмотрению самого 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 выполняете поиск 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 непонимание 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.

Основные концепции

Ниже дан очень краткий обзор основных понятий баз данных. Он предназначен для того, чтобы освежить ваши знания либо дать 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 применительно к SQL мы будем считать базу данных набором записей, 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), предназначенный для хранения упорядоченных данных.

Неправильное употребление термина приводит к путанице

Люди часто используют термин *база данных* для обозначения программного обеспечения, управляющего базой данных. Это ведет к путанице. В действительности 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 сваливать их в кучу. Вместо этого вы раскладываете их по соответствующим папкам.

В базах 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 называться таким именем.

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 ячеек, как в Excel. В каждом столбце сетки находится определенная часть информации. Например, в таблице клиентов в одном столбце находится номер клиента, в другом — его имя. Адрес, город, штат, почтовый индекс — все это хранится в отдельных столбцах.

Распределение данных по столбцам

Очень важно правильно распределить данные по нескольким столбцам. Например, название города и штата, а также почтовый индекс всегда должны находиться в отдельных столбцах. Это позволяет сортировать и фильтровать данные по столбцам (например, для поиска всех клиентов из определенного штата или города). Если названия города и штата будут храниться в одном столбце, то это сильно затруднит сортировку и фильтрацию данных по штатам.

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 определяет, какие данные могут храниться в этом столбце. Например, если в столбце должны содержаться числа (допустим, количество товаров в заказе), то тип данных будет числовым. Если в столбце необходимо хранить даты, заметки, денежные суммы и т.п., то для всех этих данных предусмотрены соответствующие 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 сортировать инфор-

мацию и играют важную роль в оптимизации использования места на диске. Таким образом, выбору типов данных для столбцов создаваемой таблицы необходимо уделить особое внимание.

Совместимость типов данных

Типы данных и их названия служат 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 содержится в своей строке. Возвращаясь к сравнению с сеткой ячеек Excel, можно сказать, что ее вертикальные ряды представляют собой столбцы таблицы, а горизонтальные ряды — это строки.

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.

Записи или строки?

Часто пользователи баз данных говорят о *записях*, имея в виду *строки*. Эти два термина взаимозаменяемы.

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, а в таблице со список книг уникальным идентификатором будет служить ISBN.

Первичный ключ

Столбец (или набор столбцов), значения которого уникальным образом 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.

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 содержать значения NULL).
- Значения в столбце первичного ключа не могут быть изменены.
- Значения первичного ключа нельзя использовать дважды. (Если строка удалена из таблицы, то 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 приведенные выше правила 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 — *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 (таких, как Java, C или Python), SQL состоит из относительно небольшого количества слов английского языка. Так было изначально задумано. SQL разрабатывался для решения одной конкретной задачи — предоставлять простой и эффективный способ чтения и записи информации из баз данных.

Каковы же преимущества SQL?

- SQL не относится к числу проприетарных языков, используемых поставщиками конкретных СУБД. Почти все ведущие СУБД поддерживают SQL, по-

этому 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 — это очень мощный язык. Умело пользуясь его инструкциями, можно выполнять очень сложные операции с базами данных.

Вот почему стоит изучить SQL.

Расширения SQL

Многие поставщики СУБД расширили возможности SQL, добавив в язык дополнительные операторы или инструкции. Эти расширения необходимы для обеспечения дополнительной функциональности или для упрощения определенных операций. И хотя часто они бывают очень полезными, 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. Отдельным реализациям присвоены собственные имена (PL-SQL в Oracle, Transact-SQL в Microsoft SQL Server 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.

Какую СУБД выбрать?

Для работы с примерами книги требуется СУБД. Но какую из них выбрать?

К счастью, приводимый в книге код SQL может выполняться практически в любой СУБД. Соответственно, вы вправе выбирать то, что для вас удобнее и проще.

Тут есть два варианта. Прежде всего, можете инсталлировать СУБД на свой компьютер, чтобы иметь полный доступ к базам данных. Впрочем, для многих именно установка и настройка СУБД — самая сложная часть в изучении SQL. Альтернативный вариант — получить доступ к серверной (или облачной) СУБД, чтобы ничего не требовалось инсталлировать и конфигурировать.

Для выбора доступно множество СУБД. Вот несколько рекомендаций.

- MySQL (как и ее ответвление MariaDB) — самое лучшее решение, потому что это легко инсталлируемая бесплатная СУБД с поддержкой для всех операционных систем. Имеется утилита командной строки, позволяющая вводить SQL-код, но есть и графическая утилита MySQL Workbench, которую нужно загружать и инсталлировать отдельно.
- Для пользователей Windows доступно приложение Microsoft SQL Server Express. Это бесплатная версия популярной СУБД SQL Server, включающая клиентский модуль SQL Server Management Studio.

Теперь что касается выбора серверной (или облачной) СУБД.

- Если вы изучаете SQL для применения на работе, то, скорее всего, у вас на фирме уже используется какая-то СУБД. Попросите администратора создать для вас учетную запись и предоставить вам утилиту, с помощью которой можно вводить и тестировать SQL-запросы.
- Облачные СУБД — это экземпляры, выполняющиеся на виртуальных серверах, благодаря чему вам не нужно ничего инсталлировать у себя на компьютере. Такие СУБД предлагаются всеми ведущими постав-

щиками облачных служб (включая Google, Amazon и Microsoft). К сожалению, конфигурирование этих СУБД (включая настройку удаленного доступа) — нетривиальная задача, зачастую требующая большего объема работы, чем при установке локальной СУБД. Исключение составляют разве что Oracle Live SQL и IBM Db2 on Cloud, предлагающие бесплатную версию с веб-интерфейсом.

Ссылки на все упомянутые СУБД доступны на сайте книги по адресу `http://forta.com/books/0135182794`. По мере обновления СУБД эта страница тоже будет обновляться.

Во всех упражнениях книги используются реальные инструкции SQL и полноценные таблицы базы данных. В приложении А описываются все демонстрационные таблицы и рассказывается о том, как их получить (или создать самостоятельно), чтобы можно было выполнять примеры каждого урока.

Кроме того, начиная с урока 2 в конце каждой главы приводятся упражнения. Это даст вам возможность закрепить полученные знания и применить их для решения задач, не рассмотренных в самой главе. Ответы на упражнения приводятся в приложении Д.

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, мы также рассмотрели основную терминологию 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 слов английского языка. Эти слова называются *ключевыми*, и каждая инструкция SQL содержит одно или несколько таких слов. Чаще всего вы будете иметь дело с инструкцией SELECT, которая предназначена для извлечения информации из одной или нескольких таблиц.

Ключевое слово

Зарезервированное слово, составляющее часть лексикона SQL. Никогда не называйте таблицу или столбец таким словом. В приложении Г 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.

Выполняйте примеры по ходу чтения книги

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

Используйте отдельную базу данных

Все СУБД позволяют работать с несколькими базами данных. Для выполнения примеров с демонстрационными таблицами (они описаны в приложении А) рекомендуется создать отдельную базу данных. Если вы вдруг столкнетесь с непонятными ошибками при выполнении запросов, убедитесь в том, что выбрана правильная база данных.

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 в предложении FROM указывается имя таблицы, из которой извлекаются данные. Результат выполнения инструкции будет следующим.

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
```

В зависимости от используемой СУБД вы можете получить сообщение о том, сколько строк было выбрано и за какое время. Например, в командной строке MySQL будет получено примерно такое сообщение:

```
9 rows in set (0.01 sec)
```

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 мы поговорим 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, что набор возвращаемых строк будет одним и тем же.

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

Завершение инструкций

Несколько инструкций SQL должны быть разделены точкой с запятой. 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 считается обязательной.

Инструкции SQL и регистр символов

Важно подчеркнуть, что инструкции SQL нечувствительны к регистру символов, поэтому ключевые слова `SELECT`, `select` и `Select` равнозначны. 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: в отличие от инструкций SQL имена таблиц, столбцов и значений могут оказаться чувствительными к регистру (это зависит от СУБД и ее конфигурации).

Использование пробелов

Все лишние пробелы в инструкции SQL пропускаются при обработке запроса. Поэтому инструкция 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 тестировать.

Извлечение нескольких столбцов

Для извлечения нескольких столбцов из таблицы применяется уже рассмотренная нами инструкция SELECT. Отличие состоит в том, что после ключевого слова SELECT через запятую перечисляется несколько имен столбцов.

Будьте внимательны с запятыми

При перечислении нескольких столбцов вставляйте запятые только между ними, но не после завершающего столбца в списке, так как 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 через запятую перечислены три имени столбца. Результат выполнения инструкции показан ниже.

Вывод ▼

```
prod_id     prod_name              prod_price
---------   --------------------   ----------
BNBG01      Fish bean bag toy            3.49
BNBG02      Bird bean bag toy            3.49
BNBG03      Rabbit bean bag toy          3.49
BR01        8 inch teddy bear            5.99
BR02        12 inch teddy bear           8.99
```

```
BR03        18 inch teddy bear      11.99
RGAN01      Raggedy Ann              4.99
RYL01       King doll                9.49
RYL02       Queen doll               9.49
```

Вывод результатов запроса

Как видно из показанного примера, инструкции SQL обычно возвращают неформатированные данные, и в разных СУБД и приложениях эти данные могут отображаться по-разному (в частности, с разным количеством пробелов между столбцами и знаков после десятичной точки). Форматирование данных — задача вывода результатов на экран, а не выборки значений из базы данных. Поэтому представление записей на экране зависит от конкретного приложения, применяемого для выполнения запросов.

Извлечение всех столбцов

Помимо извлечения конкретных столбцов (одного или нескольких), с помощью инструкции SELECT можно запросить все столбцы, не перечисляя каждый из них. Для этого вместо имен столбцов следует указать символ "звездочка" (*), как в показанном ниже примере.

Ввод ▼

```
SELECT *
FROM Products;
```

Анализ ▼

При наличии символа подстановки (*) возвращаются все столбцы. Обычно (но не всегда) столбцы возвращаются в том порядке, в котором они перечислены в определении таблицы. Впрочем, табличные данные редко вы-

водятся в том виде, в котором они хранятся в базе данных. (Обычно они возвращаются в приложение, которое должным образом форматирует их.)

Использование символов подстановки

Лучше не использовать символ * (кроме тех случаев, когда вам действительно необходимы все столбцы таблицы). Несмотря на то что это позволяет сэкономить время и усилия, затрачиваемые на перечисление требуемых столбцов, извлечение ненужных столбцов обычно приводит к снижению производительности запроса и приложения в целом.

Извлечение неизвестных столбцов

У символа подстановки есть лишь одно преимущество: поскольку вы не указываете точные имена столбцов (при наличии символа * возвращаются все столбцы), появляется возможность извлечь столбцы, имена которых вам 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 стоять непосредственно перед списком имен столбцов.

Вывод ▼

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

Строки не могут быть частично уникальными

Ключевое слово DISTINCT применяется ко всем столбцам, а не только к тому, перед которым оно стоит. Если задать предложение SELECT DISTINCT vend_id, prod_price, то будут извлечены шесть строк из девяти, поскольку для данной пары столбцов возможны шесть уникальных комбинаций. Чтобы увидеть разницу, введите следующие две инструкции и сравните результаты.

```
SELECT DISTINCT vend_id, prod_price
FROM Products;
SELECT vend_id, prod_price FROM Products;
```

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. К сожалению, это именно та ситуация, в которой разные СУБД ведут себя по-разному.

В Microsoft SQL Server 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` позволяет извлечь из таблицы первые пять 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` в данном случае не требует 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 ▼

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
```

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

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.

Краткая форма записи для MySQL, MariaDB и SQLite

MySQL, MariaDB и SQLite поддерживают краткую форму предложения `LIMIT 4 OFFSET 3`, позволяя записать его как `LIMIT 3, 4`. В данной форме число перед запятой задает начальную точку, а после запятой — количество строк (порядок чисел обратный, так что будьте внимательны).

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 портируемые, но более сложные всегда нужно проверять. Помните об этом при разработке приложений, работающих с базами данных.

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 программистов). 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*, рассмотренные в приложении А.
- То же самое справедливо для заголовков в начале SQL-сценария, содержащих контактные данные программиста, а также различные описания и примечания. Примеры этого также можно увидеть в вышеуказанных файлах.
- Другое важное применение комментариев — временный запрет на выполнение 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. Сначала рассмотрим встроенные комментарии.

Ввод ▼

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

Анализ ▼

Для встраивания комментариев предназначено выражение -- (два дефиса). Все, что идет далее до конца строки, 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 отключения 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 или 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. На следующем уроке мы рассмотрим, как сортировать результаты запроса.

Упражнения

1. Напишите инструкцию SQL для извлечения идентификаторов всех клиентов (`cust_id`) из таблицы `Customers`.
2. В таблице `OrderItems` хранится список всех заказанных товаров, причем некоторые товары были заказаны неоднократно. Напишите инструкцию SQL, которая возвращает перечень заказанных товаров (`prod_id`). *Подсказка:* вы должны получить семь уникальных строк.
3. Напишите инструкцию SQL, которая извлекает все столбцы из таблицы `Customers`, и альтернативную инструкцию `SELECT`, которая извлекает только идентификаторы клиентов. Используйте комментарии, чтобы выполнять только одну из инструкций (но протестируйте обе).

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 явно заданной 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 аргументы запроса. Примером может служить предложение `FROM` инструкции `SELECT` из предыдущего примера.

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 по столбцу prod_name, как показано 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
```

Местоположение предложения ORDER BY

При использовании предложения ORDER BY убедитесь в том, что оно стоит последним в инструкции SELECT. В противном случае будет выдано сообщение об ошибке.

Сортировка по невыбранным столбцам

Чаще всего столбцы, указываемые в предложении ORDER BY, отображаются на экране, хотя это не обязательно. Вполне допускается сортировать данные по столбцу, который не извлекается в самом запросе.

Сортировка по нескольким столбцам

Часто бывает необходимо отсортировать данные по нескольким столбцам. Например, если выводится список сотрудников, может понадобиться отсортировать его

сначала по фамилии сотрудника, а затем по имени. Это будет полезным, если в компании есть однофамильцы.

Чтобы выполнить сортировку по нескольким столбцам, перечислите их имена через запятую в предложении ORDER BY. В следующем примере извлекаются три столбца, а результат сортируется по двум из них: сначала по цене, затем 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 показанном 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, то 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 в таблице результатов запроса). 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`. В нем не указаны имена столбцов, а лишь задано их относительное положение в предложении `SELECT`. Выражение `ORDER BY 2` означает сортировку по второму столбцу списка, а именно по столбцу `prod_price`. Пред-

ложение 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 есть и недостатки. Во-первых, отсутствие явных имен столбцов повышает вероятность того, что вы случайно укажете не тот столбец. Во-вторых, можно случайно нарушить порядок столбцов при изменении предложения 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 предложении SELECT.

Сортировка по невыбранным столбцам

Рассмотренный метод нельзя применять для сортировки по столбцам, не указанным в предложении SELECT. Но допускается в одной инструкции задавать как имена столбцов, так и их номера.

Указание направления сортировки

Данные можно сортировать не только по возрастанию (от 'A' до 'Z'). Такой порядок применяется по умолчанию, но в предложении ORDER BY можно также задавать сортировку по убыванию (от 'Z' до 'A'). Для этого предназначено ключевое слово DESC.

В следующем примере товары сортируются по убыванию цены (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 по убыванию цены (вначале идут самые дорогие), а также по названию.

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 данном примере ключевое слово DESC указано только для столбца prod_price. Таким образом, столбец prod_price сортируется по убыванию, а столбец prod_name (в пределах каждого ценового блока) — как обычно, по возрастанию.

Сортировка по убыванию для нескольких столбцов

Если необходимо отсортировать данные в порядке убывания по нескольким столбцам, укажите для каждого из них ключевое слово DESC.

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' идентично 'a', и такое поведение типично для большинства СУБД. Но в некоторых СУБД администратор может в случае необходимости изменить данную настройку. (Это будет 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.

Следует уточнить, что DESC — это сокращение от DESCENDING; допускается использовать оба ключевых слова. Его антоним — ключевое слово ASC (ASCENDING), означающее сортировку по возрастанию. На практике оно обычно не применяется, поскольку такой порядок задается по умолчанию (если не указано ни ASC, ни DESC).

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.

Упражнения

1. Напишите инструкцию SQL, которая извлекает имена всех клиентов (`cust_name`) из таблицы `Customers` и сортирует результаты по убыванию.
2. Напишите инструкцию SQL, которая извлекает идентификаторы клиентов (`cust_id`) и номера заказов (`order_num`) из таблицы `Orders` и сортирует результаты сначала по идентификатору, а затем по дате заказа в обратном хронологическом порядке.
3. В нашем вымышленном магазине активно продаются дорогие товары. Напишите инструкцию SQL, которая извлекает информацию о количестве и цене (`item_price`) товаров из таблицы `OrderItems`, сортируя результаты сначала по убыванию количества, а затем по убыванию цены.
4. Что неправильного в следующей инструкции SQL? (Постарайтесь понять это, не выполняя саму инструкцию).

```
SELECT vend_name,
FROM Vendors
ORDER vend_name DESC;
```

УРОК 4

Фильтрация данных

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

Предложение 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 должно стоять сразу после названия таблицы (в предложении FROM), как показано ниже.

Ввод ▼

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

Анализ ▼

Данная инструкция извлекает два столбца из таблицы Products, но возвращает не все строки, а только те из них, 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 этом примере используется простая проверка на равенство: сначала проверяется, содержится ли в столбце указанное значение, а затем данные фильтруются соответствующим образом. Но в SQL можно выполнять не только проверку на равенство.

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 типы 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.

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 брать на себя функции базы данных, вы значительно ухудшаете его производительность и масштабируемость. Кроме того, если данные фильтруются на стороне клиента, то сервер пересылает по сети ненужные данные, тем самым увеличивая сетевой трафик.

Положение предложения WHERE

При использовании обоих предложений, ORDER BY и WHERE, убедитесь в том, что предложение ORDER BY стоит после предложения WHERE, иначе возникнет ошибка (см. урок 3).

Операторы в предложении WHERE

В предыдущем примере выполнялась проверка на равенство, т.е. определялось, содержится ли в столбце указанное значение. В SQL поддерживается целый набор операторов сравнения, перечисленных 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	Равенство значению NULL

Совместимость операторов

Некоторые из операторов, приведенных в табл. 4.1, избыточны. Например, <> — это то же самое, что и !=, а 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 операторы сравнения она поддерживает.

Сравнение с одиночным значением

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

В следующем примере выводятся названия товаров, стоимость которых меньше 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
```

Следующая инструкция извлекает все товары, которые стоят не более 10 долларов (результат будет таким

же, как и в предыдущем случае, поскольку в таблице 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 товары от всех поставщиков, кроме 'DLL01'.

Ввод ▼

```
SELECT vend_id, prod_name
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
```

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_name
FROM Products
WHERE vend_id != 'DLL01';
```

Оператор != или <>

Операторы != и <> обычно взаимозаменяемы. Однако не во всех СУБД поддерживаются обе формы оператора. Если у вас возникают 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. В частности, данный оператор можно использовать для поиска товаров, цена которых находится в диапазоне от 5 до 10 долларов, или всех дат, которые попадают в интервал между 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, в операторе 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, которые равны граничным значениям.

Проверка на отсутствие значения

При создании таблицы разработчик может указать, допустимо ли, чтобы в отдельных ее столбцах не содержалось никаких значений. 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. В предложении 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 (т.е. поле prod_price пустое, а не содержит цену 0), а поскольку таковых у нас нет, никаких данных мы не получим. Зато 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
```

Операторы, специфичные для конкретной СУБД

Во многих СУБД набор операторов расширен дополнительными фильтрами. За информацией обратитесь к документации своей СУБД.

Значения NULL и проверка 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).

Упражнения

1. Напишите инструкцию SQL, которая извлекает идентификатор товара (`prod_id`) и название товара (`prod_name`) из таблицы `Products`, возвращая только товары с ценой `9.49`.
2. Напишите инструкцию SQL, которая извлекает идентификатор товара (`prod_id`) и название товара (`prod_name`) из таблицы `Products`, возвращая только товары с ценой не ниже `9`.
3. Напишите инструкцию SQL, которая извлекает из таблицы `OrderItems` список уникальных номеров заказов (`order_num`), содержащих не менее 100 единиц какого-либо товара.
4. Напишите инструкцию SQL, извлекающую название товара (`prod_name`) и цену товара (`prod_price`) из таблицы `Products` для всех товаров, цена на которые находится в диапазоне от `3` до `6`. Результаты должны быть отсортированы по цене. (Существует несколько вариантов решения, как будет показано в следующей главе. Используйте те возможности языка, которые изучили на данный момент.)

УРОК 5

Расширенная фильтрация данных

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

Комбинирование предложений WHERE

Все предложения WHERE, рассмотренные на уроке 4, фильтровали данные с использованием одного критерия. Для создания более сложных фильтров можно использовать несколько условий WHERE, объединяемых с помощью операторов 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. Рассмотрим пример.

Ввод ▼

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

Анализ ▼

С помощью данного запроса извлекаются идентификаторы, цены и названия всех товаров, предлагаемых поставщиком `'DLL01'` по цене не выше 4 долларов. Предложение `WHERE` 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, то он не попадет в результаты запроса. Аналогичным образом товары, которые стоят меньше 4 долларов, но предлагаются иными поставщиками, тоже не будут выведены. Результаты запроса будут следующими.

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 данном примере использовались два условия фильтрации, объединенные оператором `AND`. Можно задавать

и большее количество условий. Каждое из них должно отделяться оператором AND.

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 приведенных в книге. Количество возвращаемых строк всегда будет одинаковым, а вот их порядок может оказаться иным. Добавляйте 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 соблюдается 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 ▼

С помощью этого запроса извлекаются названия и цены всех товаров, изготовленных одним из двух поставщиков. Оператор OR заставляет СУБД применять какое-то одно условие, но не оба сразу. Если бы здесь использо-

вался оператор AND, то мы не получили бы никаких данных. Результаты запроса будут следующими.

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 операторов AND и OR. Комбинируя их, можно создавать сложные фильтры.

Впрочем, при комбинировании операторов AND и OR возникает одна проблема. Предположим, необходимо вывести список всех предлагаемых поставщиками 'DLL01' и 'BRS01' товаров по цене не ниже 10 долларов. В следующей инструкции SELECT используется комбинация операторов AND и OR для формирования критерия 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, что-то пошло не так. В чем же причина? Причина заключается в порядке обработки операторов. В SQL (как и в большинстве других языков) вначале обрабатываются операторы `AND` и только потом — операторы `OR`. Когда СУБД встречает такое предложение `WHERE`, она интерпретирует его следующим образом: *"Извлечь все товары поставщика `'BRS01'`, которые стоят не менее 10 долларов, а также все товары поставщика `'DLL01'`, независимо от их цены"*. Другими словами, в связи с тем, что приоритет у оператора `AND` выше, чем у оператора `OR`, были объединены не те условия.

Решение проблемы состоит в использовании скобок для правильного группирования необходимых 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 ▼

Единственное отличие от предыдущего запроса — скобки, в которые заключены первые два условия предложения WHERE. Поскольку скобки имеют еще большее высокий приоритет, чем операторы AND и OR, СУБД вначале обрабатывает условие OR в скобках. Соответственно, запрос будет интерпретирован так: *"Извлечь все товары, которые стоят не менее 10 долларов и предлагаются либо поставщиком 'DLL01', либо поставщиком 'BRS01'"*. Это именно то, что нам нужно.

Использование скобок в предложении WHERE

Используя операторы AND и OR в предложении WHERE, всегда добавляйте скобки, чтобы правильно сгруппировать условия. Не полагайтесь на порядок обработки, заданный по умолчанию, даже если он подразумевает нужный вам результат. Тем самым вы всегда будете застрахованы от неожиданностей.

Оператор IN

Оператор IN предназначен для сравнения со списком допустимых значений, указанных в скобках. Значения 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 ▼

В данном случае инструкция SELECT возвращает все товары, предлагаемые поставщиками 'DLL01' и 'BRS01'. После оператора IN указан список допустимых значений, разделенных запятыми, и весь список заключен в скобки.

Если вам кажется, что оператор IN равнозначен оператору OR, то вы совершенно правы! Следующий запрос возвращает точно такие же результаты.

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 допустимых значений синтаксис оператора IN намного понятнее.
- При использовании оператора IN совместно с операторами AND и OR проще управлять порядком обработки.
- Операторы IN почти всегда обрабатываются быстрее, чем списки операторов OR (впрочем, это сложно заметить в случае коротких списков).
- Самое большое преимущество оператора IN заключается в том, что в нем может содержаться еще одна инструкция SELECT, а это позволяет создавать очень гибкие фильтры. (Подробнее о вложенных запросах рассказывается на уроке 11.)

IN

Ключевое слово, используемое в предложении WHERE для указания списка допустимых значений, обрабатываемых так же, как и в случае применения оператора OR.

Оператор NOT

Оператор NOT в предложении WHERE позволяет инвертировать последующее условие. Поскольку он никогда не используется сам по себе (а только вместе с другими операторами), его синтаксис немного отличается: этот оператор должен стоять перед названием столбца, значения которого нужно отфильтровать, а не после.

NOT

Ключевое слово, применяемое в предложении WHERE для инверсии имеющегося условия.

В следующем примере демонстрируется применение оператора NOT. В этом запросе 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 ▼

Оператор NOT инвертирует условие, следующее за ним, поэтому СУБД извлекает не те значения vend_id, которые совпадают с 'DLL01', а все остальные.

Предыдущий запрос можно переписать с использованием 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 оператор NOT? Конечно же, в таких простых предложениях WHERE он не обязателен. Его преимущества проявляются в более сложных фильтрах. Например, для нахождения всех строк, не содержащих одно из допустимых значений, можно использовать оператор NOT в связке с оператором IN.

Оператор NOT в MariaDB

В MariaDB оператор NOT используется только для инверсии 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 операторы AND и OR в предложениях WHERE. Было также показано, как управлять порядком обработки операторов и как применять операторы IN и NOT.

Упражнения

1. Напишите инструкцию SQL, которая извлекает имя поставщика (vend_name) из таблицы Vendors, возвращая только поставщиков из Калифорнии. (Для этого потребуется применить фильтр как по стране ['USA'], так и по штату ['CA']. Кто знает, может, Калифорния существует и за пределами США?). *Подсказка:* в предложении фильтрации придется сравнивать строки.
2. Напишите инструкцию SQL для поиска всех заказов, содержащих как минимум 100 элементов 'BR01', 'BR02' или 'BR03'. Инструкция должна возвращать номер заказа (order_num), идентификатор товара (prod_id) и количество товара из таблицы OrderItems, фильтруя результаты как по идентификатору товара, так и по количеству. *Подсказка:* при составлении фильтра учитывайте порядок проверки значений.
3. Вернемся к упражнению из предыдущего урока. Напишите инструкцию SQL, извлекающую название товара (prod_name) и цену товара (prod_price) из таблицы Products для всех товаров, цена на которые находится в диапазоне от 3 до 6. Используйте оператор AND и отсортируйте результаты по цене.
4. Что неправильного в следующей инструкции SQL? (Постарайтесь понять это, не выполняя саму инструкцию.)

```
SELECT vend_name
FROM Vendors
ORDER BY vend_name
WHERE vend_country = 'USA' AND
      vend_state = 'CA';
```

УРОК 6

Фильтрация с использованием метасимволов

На этом уроке вы узнаете, что такое метасимволы и как их использовать в операторе `LIKE` *для фильтрации извлекаемых данных.*

Оператор `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 искалось известное значение.

Однако подобный способ фильтрации данных работает не всегда. Например, как бы вы искали товары, в названии которых содержится текст `'bean bag'`? Этого нельзя сделать с помощью простых операторов сравнения, и здесь на помощь приходит поиск с использованием метасимволов. Благодаря метасимволам можно создавать расширенные критерии отбора строк. В данном случае, для того чтобы найти все товары, в названии которых содержится текст `'bean bag'`, необходимо составить шаблон поиска, позволяющий найти строку `'bean bag'` в любом месте названия товара.

Метасимволы

Специальные символы, применяемые для поиска части значения.

Шаблон поиска

Условие фильтрации строк, состоящее из текста и метасимволов.

Метасимволы представляют собой специальные знаки, которые особым образом трактуются в предложении `WHERE`. В SQL поддерживается несколько видов метасимволов.

Чтобы использовать метасимволы в критериях 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 считается оператором? Когда это *предикат*, задающий размер множества проверяемых значений. Технически `LIKE` — это предикат, а не оператор. Результат остается тем же, просто не пугайтесь данного термина, если встретите его в документации по SQL.

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

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 товары, содержащие текст `'bean bag'` в любом месте названия, независимо от количества символов перед указанным текстом или после него.*

Метасимвол может стоять внутри шаблона поиска, хотя это применяется нечасто. 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';
```

Поиск фрагментов адресов электронной почты

Это как раз та ситуация, в которой метасимволы оказываются очень полезными внутри 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.

Следите за замыкающими пробелами

Некоторые СУБД заполняют содержимое поля пробелами. Например, если столбец рассчитан на 50 символов, а в него вставлен текст `'Fish bean bag toy'` (17 символов), то для заполнения столбца в него могут быть добавлены еще 33 пробела. Обычно это не влияет на вывод 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, то оно не будет заканчиваться на `'y'`, и в результате строка `'Fish bean bag toy'` не будет извлечена. Одним из решений может стать добавление второго символа `%` в шаблон поиска: `'F%y%'`. Теперь будут учитываться любые символы (в том числе пробелы) после буквы `'y'`. Но лучше "отсечь" пробелы с помощью строковых функций, которые рассматриваются на уроке 8.

Следите за значениями NULL

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

Метасимвол "знак подчеркивания" (_)

Еще один полезный метасимвол — знак подчеркивания (_). Он используется так же, как и знак %, но при этом ищется совпадение не со всем множеством символов, а всего с одним символом.

Метасимволы в Db2

Метасимвол _ не поддерживается в Db2.

Рассмотрим пример.

Ввод ▼

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

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 предложения WHERE использованы два метасимвола _, после которых следует текст. В результате были отобраны только те строки, которые

удовлетворяют заданному 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" ([])

Квадратные скобки ([]) служат для указания набора допустимых символов, с которыми ищется совпадение в позиции метасимвола.

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
```

Наборы символов не всегда поддерживаются

В отличие от описанных ранее метасимволов, использование квадратных скобок для создания проверочных наборов поддерживается далеко не всеми СУБД. Наборы поддерживаются в Microsoft SQL Server, но не в MySQL, Oracle, Db2 и SQLite. Обратитесь к документации своей СУБД, чтобы узнать, поддерживаются ли в ней поисковые наборы.

Анализ ▼

Условие фильтрации в этой инструкции выглядит так: `'[JM]%'`. В данном шаблоне поиска используются два разных метасимвола. По критерию `'[JM]'` осуществляется поиск всех клиентов, имена которых начинаются на одну из указанных в квадратных скобках букв, при этом проверяется только один символ. Благодаря метасимволу `%`, следующему после `'[JM]'`, выполняется поиск любого количества символов после первой буквы, что и приводит к нужному результату.

Существует метасимвол, обозначающий инверсию поискового шаблона: ^. 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
```

Конечно, аналогичного результата можно достичь с помощью оператора NOT. Единственное преимущество символа ^ — 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 все 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 поискового шаблона. Шаблоны, начинающиеся с метасимволов, обрабатываются медленнее всего.
- Внимательно следите за позицией метасимволов. Если они находятся не на своем месте, то будут извлечены не те данные.

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

Резюме

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

Упражнения

1. Напишите инструкцию SQL, которая извлекает название товара (`prod_name`) и его описание (`prod_desc`) из таблицы `Products`, возвращая только товары со словом `'toy'` в описании.
2. Теперь сделаем противоположное. Напишите инструкцию SQL, которая извлекает название товара (`prod_name`) и его описание (`prod_desc`) из таблицы `Products`, возвращая только товары без слова `'toy'` в описании. На этот раз отсортируйте результаты по названию товара.
3. Напишите инструкцию SQL, которая извлекает название товара (`prod_name`) и его описание (`prod_desc`) из таблицы `Products`, возвращая только товары со словами `'toy'` и `'carrots'` в описании. Это можно сделать несколькими способами, но в данном случае используйте оператор `AND` и два предиката `LIKE`.
4. Следующее упражнение посложнее. Соответствующий синтаксис не был рассмотрен в данной главе, поэтому постарайтесь определить его самостоятельно. Напишите инструкцию SQL, которая извлекает название товара (`prod_name`) и его описание (`prod_desc`) из таблицы `Products`, возвращая только товары со словами `'toy'` и `'carrots'` в описании, причем слова должны встречаться именно в таком порядке (слово `'toy'` должно предшествовать слову `'carrots'`). *Подсказка:* вам понадобится один предикат `LIKE` с тремя символами `%`.

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 `OrderItems` хранится цена каждого товара и его количество, но не полная стоимость (цена, умноженная на количество). Чтобы распечатать инвойс, требуется 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 записи 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 представляют собой реальные столбцы таблицы, а какие — вычисляемые поля. С точки зрения клиента (например, пользовательского приложения) данные вычисляемого поля возвращаются точно так же, как и данные из любого другого столбца.

Клиентское или серверное форматирование?

Многие операции преобразования и форматирования, которые могут быть выполнены посредством инструкций SQL, могут быть также реализованы и клиентским приложением. Но, 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 имя поставщика вместе с информацией об адресе. Предположим, необходимо создать отчет по поставщику и указать его страну как часть его имени в виде *имя (страна)*.

В отчете должно быть одно поле, 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. Инструкцию `SELECT`, которая возвращает имена поставщиков вместе с названиями стран, составить несложно, но как получить комбинированное значение?

Конкатенация

Комбинирование строк (путем присоединения их друг к другу) для получения одной длинной строки.

Для этого необходимо конкатенировать два столбца. 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.

Оператор + или ||?

В SQL Server для конкатенации используется оператор +. В Db2, Oracle, PostgreSQL и SQLite поддерживается оператор ||. За более подробной информацией обратитесь к документации своей СУБД.

Ниже приведен пример использования 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     )
```

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 этих инструкциях SELECT выполнялась 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.

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

Как видите, инструкция SELECT, которая использовалась для конкатенации полей адреса, 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, поскольку 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(RTrim(vend_name), ' (',
              RTrim(vend_country), ')')
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

Анализ ▼

Сама по себе эта инструкция SELECT ничем не отличается от предыдущего примера, 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 сможет обращаться к нему по имени, как если бы это был реальный столбец таблицы.

Ключевое слово AS обычно необязательное

Ключевое слово AS необязательное во многих СУБД, но его применение считается общепринятой практикой.

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.

Имена псевдонимов

Псевдонимом может служить как одно слово, так и целая строка. Если используется строка, то она должна быть заключена в кавычки, но, вообще говоря, поступать подобным образом не рекомендуется. Многословные имена, конечно, удобнее читать, однако они создают множество проблем для клиентских приложений. Более того, чаще всего псевдонимы применяются именно для переименования многословных названий столбцов в однословные.

Выполнение арифметических вычислений

Еще один способ использования вычисляемых полей — выполнение арифметических операций над извлеченными данными. Рассмотрим пример. В таблице `Orders` хранятся все полученные заказы, а в таблице `OrderItems` — списки товаров по каждому заказу. 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 представляет собой вычисляемое поле. Формула здесь простая: `quantity*item_price`. Теперь клиентское приложение сможет использовать новый вычисляемый столбец подобно любому другому столбцу в таблице.

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

Таблица 7.1. Арифметические операторы в SQL

Операция	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление

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 Curdate();` использует функцию `Curdate()` (поддерживается в MySQL и MariaDB) 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 арифметических операций. Кроме того, было показано, как создавать и применять псевдонимы, чтобы клиентское приложение могло обращаться к вычисляемым полям.

Упражнения

1. Псевдонимы обычно применяются для переименования полей в результатах запроса. Напишите инструкцию SQL, которая извлекает поля `vend_id`, `vend_name`, `vend_address` и `vend_city` из таблицы `Vendors`, переименовывая `vend_name` в `vname`, `vend_city` в `vcity` и `vend_address` в `vaddress`. Отсортируйте результаты по имени поставщика (можете использовать как оригинальное, так и переименованное поле).
2. В нашем магазине проводится распродажа, и все товары подешевели на 10%. Напишите инструкцию SQL, которая возвращает поля `prod_id`, `prod_price` и `sale_price` из таблицы `Products`, где `sale_price` — это вычисляемое поле скидочной цены. *Подсказка:* оригинальную цену можно умножить на `0.9` для получения скидки 10%.

УРОК 8

Функции обработки данных

На этом уроке вы узнаете, что такое функции, какие разновидности 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.

Совместимость функций

Прежде чем переходить к примерам, следует обратить внимание на то, что использование SQL-функций может оказаться проблематичным.

В отличие от инструкций SQL (таких, как `SELECT`), которые в основном поддерживаются всеми СУБД одинаково, в разных СУБД могут быть реализованы разные функции. Лишь некоторые функции универсально поддерживаются во всех ведущих СУБД. Общая функциональность обычно доступна в каждой СУБД, но названия функций и их синтаксис могут существенно отличаться. Для наглядности в табл. 8.1 приведены три наиболее

часто возникающие задачи и названия соответствующих функций в различных СУБД.

Таблица 8.1. Различия в именах функций

Задача	Синтаксис
Извлечение части строки	В Db2, Oracle, PostgreSQL и SQLite — `SUBSTR()`, в MariaDB, MySQL и SQL Server — `SUBSTRING()`
Преобразование типа данных	В Oracle имеется 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 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 СУБД, может не работать в другой.

Переносимый код

Код, который может работать в разных системах.

Ставя перед собой цель обеспечить переносимость кода, многие разработчики баз данных 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.

Стоит ли использовать функции?

Решать вам, 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 СУБД писался данный код.

Применение функций

В большинстве СУБД поддерживаются следующие группы функций.

- *Строковые функции.* Используются для обработки текстовых строк (например, для отсечения пробелов, заполнения строк пробелами или преобразования символов в другой регистр).
- *Числовые функции.* Используются для выполнения арифметических операций над числовыми данными (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).
- *Функции форматирования.* Используются для генерирования отформатированного текста (например, строк даты или денежных обозначений в соответствии с региональными настройками).
- *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. Их 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).

Строковые функции

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.

Регистр символов

Как уже должно быть понятно, функции SQL не различают регистр символов, поэтому имена функций можно записывать как `upper()`, `UPPER()`, `Upper()` или `substr()`, `SUBSTR()`, `SubStr()` и т.п. Выбор за вами. Главное, будьте последовательны и соблюдайте единый стиль именования функций, чтобы SQL-код было легче читать.

В табл. 8.2 приведены наиболее часто используемые строковые функции.

Таблица 8.2. Наиболее часто используемые строковые функции

Функция	Описание
`LEFT()`	Возвращает символы из левой части строки
`LENGTH` (а также `DATALENGTH()` или `LEN()`)	Возвращает длину строки
`LOWER()`	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
`SUBSTR()` или `SUBSTRING()`	Возвращает подстроку
`SOUNDEX()`	Возвращает SOUNDEX-индекс строки
`UPPER()`	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()` учитывает 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 функция SOUNDEX() не соответствует основным концепциям SQL, большинство СУБД поддерживает ее.

Поддержка функции SOUNDEX()

Функция SOUNDEX() не поддерживается в PostgreSQL, поэтому следующий пример не будет работать в этой СУБД. Кроме того, она будет доступна в SQLite, только если при установке дистрибутива 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 контактное лицо — 'Michelle Green'. Но что если это опечатка, и в действительности 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 подобно '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-индексы. Поскольку `'Michael Green'` и `'Michelle Green'` звучат похоже, их SOUNDEX-индексы 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.

Внутренний 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 работы с ними. Как следствие, такие функции считаются одними из наиболее важных в SQL. К сожалению, они

также наименее согласованы и наименее совместимы в различных СУБД.

Рассмотрим применение этих функций на простом примере. В таблице Orders для каждого заказа указана дата. Чтобы извлечь все заказы за определенный год, необходимо выполнить фильтрацию не по всей дате, а только по ее фрагменту, который соответствует году.

Вот как получить список всех заказов, сделанных в 2020 году, в SQL Server.

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE DATEPART(yy, order_date) = 2020;
```

Вывод ▼

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

Анализ ▼

В этом примере применяется функция DATEPART(), которая, как следует 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. Сравнивая полученное значение с констан-

той 2020, 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) = 2020;
```

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

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE EXTRACT(year FROM order_date) = 2020;
```

Анализ ▼

В этом примере функция EXTRACT() используется для извлечения компонента даты, соответствующего году, чтобы его можно было сравнить со значением 2020.

Поддержка функции Extract() в PostgreSQL

В PostgreSQL тоже имеется функция Extract(), поэтому данный пример можно считать альтернативным решением (по отношению к примеру с функцией DATE_PART()).

Аналогичных результатов можно добиться с помощью оператора BETWEEN.

Ввод

```
SELECT order_num
FROM Orders
WHERE order_date BETWEEN to_date('2020-01-01',
                                 'yyyy-mm-dd')
                     AND to_date('2020-12-31',
                                 'yyyy-mm-dd');
```

Анализ ▼

В этом примере функция Oracle to_date() используется для преобразования двух строк в даты. В одной строке задается дата 1 января 2020 года, а в другой — 31 декабря 2020 года. Стандартный оператор BETWEEN позволяет осуществить поиск всех заказов, сделанных в период между этими двумя датами. Такой код не будет работать в SQL Server, поскольку в этой СУБД не поддерживается функция to_date(). Но если заменить функцию to_date() функцией DATEPART(), то данный синтаксис можно будет применять.

В Db2, 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) = 2020;
```

В SQLite формат инструкции чуть сложнее.

Ввод ▼

```
SELECT order_num
FROM Orders
WHERE strftime('%Y', order_date) = 2020;
```

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, выполнения арифметических операций с датами, форматирования дат и т.п. Но, как уже было сказано, функции даты и времени сильно зависят от конкретной СУБД. Обратитесь к документации своей СУБД и уточните, какие функции работы с датой и временем поддерживаются в ней.

Числовые функции

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 распространенные 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 в ней поддерживаются.

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 обычно по-разному реализованы в разных СУБД.

Упражнения

1. У нашего магазина появился сайт, и нужно создать учетные записи клиентов. Имя пользователя, назначаемое по умолчанию, будет представлять собой комбинацию клиентского имени и названия города. Напишите инструкцию SQL, возвращающую идентификатор клиента (`cust_id`), имя клиента (`cust_name`) и поле `user_login`, которое должно быть записано в верхнем регистре и содержать первые два символа контактного имени (`cust_contact`) и первые три символа названия города (`cust_city`). Например, в моем случае (клиент `'Ben Forta'` из города `'Oak Park'`) именем пользователя будет `BEOAK`. *Подсказка:* в этом упражнении потребуется применить несколько функций, операцию конкатенации и псевдоним.
2. Напишите инструкцию SQL, которая возвращает номер заказа (`order_num`) и дату заказа (`order_date`) для всех заказов, сделанных в январе 2020 года, причем список должен быть отсортирован по дате заказа. Имеющихся у вас знаний должно быть достаточно, но в случае необходимости загляните в документацию к своей СУБД.

УРОК 9

Итоговые вычисления

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

Итоговые функции

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. Вот несколько примеров подобных запросов:

- 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 достаточно согласованно поддерживаются в большинстве СУБД.

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 ▼

Данная 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 следущем 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 выбираются только те товары, значение vend_id для которых равно 'DLL01', поэтому значение, полученное в столбце с псевдонимом avg_price, представляет собой среднее только для товаров данного поставщика.

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().

Значения NULL

Строки столбца, содержащие значения NULL, игнорируются функцией AVG().

Функция COUNT()

Функция COUNT() подсчитывает количество строк. С ее помощью можно узнать общее число строк в таблице или количество 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 непустое значение в указанном столбце (значения NULL игнорируются).

В первом примере возвращается общее количество 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, для которых известен адрес 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 непустое значение в столбце `cust_email`. В данном случае количество строк равно 3 (т.е. только 3 из 5 клиентов имеют адрес электронной почты).

Значения NULL

Строки со значениями NULL игнорируются функцией COUNT(), если указано имя столбца, и учитываются, если используется метасимвол-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.

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.

Значения NULL

Строки со значениями NULL игнорируются функцией MAX().

Функция MIN()

Функция MIN() выполняет противоположное по отношению к функции 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 ▼

В данном случае функция MIN() возвращает цену самого дешевого товара в таблице Products.

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.

Значения NULL

Строки со значениями NULL игнорируются функцией MIN().

Функция SUM()

Функция SUM() возвращает сумму значений в указанном столбце.

Рассмотрим пример. В таблице OrderItems содержится перечень заказанных товаров, причем каждому товару соответствует определенное количество, указанное в заказе. Общее количество 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 заказанных товаров, а предложение WHERE гарантирует, что будут учитываться только товары из заказа с нужным номером.

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

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 заказа с нужным номером.

Вычисления с несколькими столбцами

Все итоговые функции позволяют выполнять вычисления над несколькими столбцами с использованием стандартных арифметических операций, как было показано в данном примере.

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 предиката (так как ALL — это поведение по умолчанию);
- для выполнения вычислений по уникальным значениям при наличии ключевого слова DISTINCT.

Предикат ALL задан по умолчанию

Ключевое слово ALL не обязательно указывать, так как оно подразумевается по умолчанию. Если не задано ключевое слово DISTINCT, то подразумевается предикат ALL.

В следующем примере функция AVG() используется для определения средней цены товаров, предлагаемых указанным поставщиком. Это такая же инструкция SELECT, как и та, что была рассмотрена ранее, но теперь в ней указано ключевое слово DISTINCT, и в результате 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.

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(*). Кроме того, ключевое слово DISTINCT должно стоять перед именем столбца, а не перед вычисляемым полем или выражением.

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 одними и теми теми же независимо от того, учитываются уникальные значения или нет.

Дополнительные аргументы итоговых функций

Помимо ключевых слов DISTINCT и ALL, некоторые СУБД поддерживают дополнительные предикаты, такие как TOP и TOP PERCENT, позволяющие выполнять итоговые вычисления 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, рассмотренных 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 цена товаров).

Имена псевдонимов

При указании псевдонимов для хранения результатов итоговой функции старайтесь не использовать реальные названия столбцов в таблице, поскольку во многих СУБД такое поведение не приветствуется, и вы получите сообщение об ошибке.

Резюме

Итоговые функции предназначены для вычисления базовых статистических показателей. В SQL поддерживаются пять таких функций, каждая из которых может использоваться несколькими способами для получения требуемых результатов. Эти функции достаточно эффективны и обычно возвращают результат гораздо быстрее, чем если бы аналогичные вычисления выполнялись в клиентском приложении.

Упражнения

1. Напишите инструкцию SQL, которая возвращает общее количество заказанных товаров (используйте столбец `quantity` таблицы `OrderItems`).
2. Модифицируйте предыдущую инструкцию, чтобы она возвращала общее количество заказов товара `'BR01'` (столбец `prod_id`).
3. Напишите инструкцию SQL, возвращающую стоимость (`prod_price`) самого дорогого товара в таблице `Products`, цена на который не превышает `10`. Назовите вычисляемое поле `max_price`.

УРОК 10

Группирование 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 группирования 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 в столбце, не прибегая к извлечению всех данных.

До этого 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 требуется 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*. Это позволяет разделить все записи 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`. Продемонстрируем это 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 вычисляемое поле `num_prods` (создается с помощью функции `COUNT(*)`). Предложение `GROUP BY` заставляет СУБД отсортировать данные и сгруппировать их по столбцу `vend_id`. В результате значение `num_prods` бу-

дет вычисляться по одному разу для каждой группы записей vend_id, а не один раз для всей таблицы products. В итоге оказывается, что поставщик 'BRS01' предлагает три товара, поставщик 'DLL01' — четыре, а поставщик 'FNG01' — два.

Благодаря предложению 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.

При использовании предложения GROUP BY необходимо руководствоваться следующими правилами.

- В предложении GROUP BY можно указывать произвольное число столбцов. Это позволяет вкладывать группы одна в другую и задавать порядок группировки.
- Если в предложении GROUP BY используются вложенные группы, то данные подытоживаются для последней указанной группы. Другими словами, если задана группировка, то 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 СУБД 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, то 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.

Ключевое слово ALL

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

Указание столбцов по их относительному положению

Некоторые СУБД позволяют указывать столбцы в предложении GROUP BY по их положению в списке 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 СУБД. Его применение также оказывается рискованным в том смысле, что существует 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 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.

Предложение HAVING поддерживает все операторы предложения WHERE

На уроках 4 и 5 было показано, как применять предложение WHERE (включая использование метасимволов и операторов сравнения). 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, означающего "два или больше заказов".

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

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. Разница оказывается существенной. 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 подумайте: возникает ли необходимость в использовании как предложения WHERE, так и предло-

жения HAVING в одной инструкции? Конечно, возникает. Предположим, вы хотите усовершенствовать фильтр предыдущей инструкции таким образом, чтобы возвращались имена всех клиентов, которые сделали не менее двух заказов 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 выше 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 записей. При отсутствии предложения WHERE была бы получена лишняя строка (поставщик, предлагающий четыре товара, каждый из которых дешевле, чем 4 доллара), 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
```

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.

Группирование 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, очень важное. 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, то 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.

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, содержащих три или более единиц товара.

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;
```

Вывод

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

Анализ ▼

В этом примере предложение `GROUP BY` используется для группирования 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 с помощью 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 данные извлекаются 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 записей и получения отдельных результатов для каждой группы. Было показано, как с помощью предложения HAVING осуществлять фильтрацию на уровне групп. Кроме того, объяснялось, в чем разница между предложениями ORDER BY и GROUP BY, а также между предложениями WHERE и HAVING.

Упражнения

1. В таблице OrderItems перечислены элементы каждого заказа. Напишите инструкцию SQL, которая возвращает количество позиций (в виде поля order_lines) для каждого номера заказа (order_num), и отсортируйте результаты по полю order_lines.
2. Напишите инструкцию SQL, которая возвращает поле cheapest_item, содержащее самый дешевый товар по каждому поставщику (используйте поле prod_price таблицы Products), и отсортируйте результаты по возрастанию цены.
3. Нам важно знать лучших клиентов, поэтому напишите инструкцию SQL, которая возвращает номер заказа (поле order_num таблицы OrderItems) для всех заказов, содержащих минимум 100 элементов.
4. Для выявления лучших клиентов можно также проверять, сколько денег они потратили. Напишите инструкцию SQL, которая возвращает номер заказа (поле order_num таблицы OrderItems) для всех заказов с суммарной ценой как минимум 1000. *Подсказка:* в данном упражнении необходимо создать вычисляемое поле (значение item_price умножается на quantity) и просуммировать итог по нему. Отсортируйте результаты по номеру заказа.
5. Что неправильного в следующей инструкции SQL? (Постарайтесь понять это, не выполняя саму инструкцию.)

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

УРОК 11

Подзапросы

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

Что такое подзапросы

Инструкции SELECT — это SQL-запросы. Все инструкции, с которыми мы имели дело до сих пор, представляли собой простые запросы на выборку: 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*, т.е. запросы, которые вложены в другие запросы. Почему возникает необходимость в подзапросах? Лучший способ объяснить эту концепцию — рассмотреть несколько примеров.

Фильтрация с помощью подзапросов

Таблицы баз данных, используемые в примерах книги, являются реляционными (все таблицы описаны в приложении A). 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 сделать 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 поле prod_id значится 'RGAN01'. В результате мы получаем номера двух заказов, содержащих данный товар.

Ввод ▼

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

Вывод ▼

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

Следующий шаг заключается в получении идентификаторов клиентов, связанных с заказами 20007 и 20008. Используя предложение IN, о котором рассказывалось 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 в действительности выполняются две операции. Вначале выполняется 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 оказывается точно таким, как и при жестком кодировании предложения WHERE в предыдущем примере.

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 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 в предложении WHERE подзапроса, внешнего по отношению к данному. Этот подзапрос возвращает перечень идентификаторов клиентов, которые используются в предложении WHERE запроса самого высокого уровня, возвращающего искомые данные.

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

Только один столбец

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

Подзапросы и производительность

Представленные здесь запросы возвращают корректные результаты. Однако подзапросы — не всегда самый эффективный способ получения данных такого рода. Подробнее 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`. Заказы хранятся в таблице `Orders` вместе с соответствующими идентификаторами клиентов.

Чтобы выполнить этот запрос, необходимо сделать следующее:

1) извлечь список клиентов из таблицы `Customers`;
2) для каждого выбранного клиента подсчитать количество 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 вычисляемое; 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 значением, которое в данный момент извлекается из таблицы Customers.

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

Подобный синтаксис — имя таблицы и имя столбца, разделенные 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
```

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 наши 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.

Подзапросы не всегда оказываются оптимальным решением

Несмотря на то что показанный здесь запрос корректен, зачастую он оказывается не самым эффективным способом извлечения данных такого рода. Мы еще вернемся к этому примеру на одном из следующих уроков.

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

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.

Упражнения

1. Используйте подзапрос для получения списка клиентов, которые купили товары по цене 10 или выше. Найдите соответствующие номера заказов (order_num) в таблице OrderItems, а затем используйте таблицу Orders, чтобы найти клиентские идентификаторы (cust_id) для этих заказов.
2. Нам нужно узнать даты заказов товара 'BR01'. Напишите инструкцию SQL, которая использует подзапрос для нахождения заказов (в таблице OrderItems), содержащих товар с идентификатором 'BR01' (prod_id), а затем возвращает идентификатор клиента (cust_id) и дату заказа (order_date) из таблицы Orders. Отсортируйте результаты по дате заказа.
3. Давайте усложним задачу. Модифицируйте предыдущую инструкцию, чтобы она возвращала адрес электронной почты клиента (поле cust_email таблицы Customers) для любого клиента, купившего товар с идентификатором 'BR01'. *Подсказка:* вам понадобятся несколько вложенных инструкций SELECT, самая внутренняя из которых возвращает поле order_num из таблицы OrderItems, а средняя возвращает поле cust_id из таблицы Orders.
4. Нам нужен список клиентских идентификаторов с общей суммой заказов по каждому из них. Напишите инструкцию SQL, которая возвращает идентификатор клиента (поле cust_id таблицы Orders) и поле total_ordered, содержащее общую сумму заказов по каждому клиенту (это должен быть подзапрос). Отсортируйте результаты по убыванию суммы заказов. *Подсказка:* ранее вы

уже использовали функцию SUM() для вычисления итогов по заказам.

5. Напишите инструкцию SQL, которая возвращает названия всех товаров (prod_name) из таблицы Products, а также вычисляемый столбец quant_sold, содержащий общее количество проданных единиц по каждому товару (для этого применяется подзапрос к таблице OrderItems, включающий функцию SUM(quantity)).

УРОК 12

Соединение таблиц

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

Что такое соединение

Одна из ключевых особенностей SQL — возможность "на лету" объединять таблицы при выполнении запросов, связанных с извлечением данных. *Соединения* — это самые мощные операции, которые можно выполнить с помощью инструкции `SELECT`, поэтому понимание соединений и их синтаксиса чрезвычайно важно для изучения SQL.

Прежде чем применять соединения, 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 глав будет вполне достаточно для того, чтобы вы смогли получить общее представление.

Что такое реляционные таблицы

Чтобы понять, что собой представляют реляционные таблицы, рассмотрим пример. Предположим, определенная таблица базы данных содержит каталог товаров, в котором каждому элементу соответствует 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), то вам придется обновить все записи о его товарах.
- Если данные многократно повторяются (а такое происходит, когда информация о поставщике указывается для каждого товара), то существует вероятность того, что кое-где данные будут введены с ошибкой. Это затрудняет составление корректных 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. Они проектируются таким образом, чтобы вся информация распределялась 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 информации о поставщиках, за исключением их идентификаторов (первичный ключ таблицы Vendors). Этот ключ связывает таблицу Vendors с таблицей Products. Благодаря идентификатору поставщика можно использовать таблицу Vendors для поиска информации о соответствующем поставщике.

Что это дает? Перечислим 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, то 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 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 соединения

Распределение данных по многим таблицам обеспечивает их более эффективное хранение, упрощает обработку данных и повышает масштабируемость базы данных в

целом. Однако эти преимущества не даются даром — за все приходится платить.

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

Ответ таков: посредством *соединений*. Соединение 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.

Использование интерактивных инструментов СУБД

Важно понимать, что соединение — это не физическая таблица. Другими словами, оно не существует как реальная таблица в базе данных. Соединение создается СУБД по мере необходимости и сохраняется только на время выполнения запроса.

Многие СУБД предлагают графический интерфейс, который можно использовать для интерактивного определения табличных связей. Такие 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, то 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 соединения

Создать соединение очень легко: нужно указать все таблицы, которые должны быть включены в соединение, 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 полученных результатов, одна инструкция SELECT сумела извлечь данные из двух разных таблиц.

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, табличное 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, вы, в сущности, 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 соединения. Без предложения WHERE каждая строка первой таблицы будет образовывать пару с каждой строкой второй таблицы независимо от того, есть ли логика в их соединении или нет.

Декартово произведение

Результаты, возвращаемые при слиянии таблиц без указания условия соединения. 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 результатов запроса, декартово произведение вы, скорее всего, будете использовать очень редко. Данные, полученные таким способом, ставят каждого поставщика в соответствие каждому товару, включая товары с указанием "не того" поставщика (и даже поставщиков, которые вообще не предлагают никаких товаров).

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 соединение

Иногда соединение, возвращающее декартово произведение, называют *перекрестным*.

Внутренние соединения

Соединение, которое мы до сих пор использовали, называется *соединением по равенству* — оно основано на проверке равенства записей двух таблиц. Соединение такого рода называют также *внутренним*. Для подобных соединений можно применять несколько иной синтак-

сис, явно указывающий на тип соединения. Следующая инструкция SELECT возвращает те же самые данные, что и в приведенном ранее 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 соединения 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.

"Правильный" синтаксис

Согласно стандарту 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.

Соединение нескольких таблиц

SQL не ограничивает количество таблиц, которые могут быть объединены посредством инструкции SELECT. Основные правила создания соединения 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 с номером 20007. Все они находятся в таблице OrderItems. Каждый элемент хранится вместе 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 задает оба условия соединения. Дополнительное условие служит для фильтрации только элементов заказа 20007.

К вопросу о производительности

Все СУБД обрабатывают соединения динамически, затрачивая время на обработку каждой указанной таблицы. Этот процесс может оказаться очень ресурсоемким, поэтому не следует использовать табличные соединения без особой надобности. Чем больше таблиц вы объединяете, тем ниже производительность.

Максимальное количество таблиц в соединении

Несмотря на то что SQL не накладывает каких-либо ограничений на количество таблиц в соединении, многие СУБД в реальности имеют такие ограничения. Обратитесь к документации своей СУБД, чтобы узнать, какие ограничения 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 оказываются самым эффективным способом выполнения сложных инструкций SELECT, поэтому тот же самый

запрос можно переписать с использованием синтаксиса соединений.

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, для получения требуемых в запросе данных необходимо обратиться к трем таблицам. Однако вместо подзапросов здесь были применены два табличных соединения. В запросе имеются три условия `WHERE`. Первые два связывают таблицы в соединение, а последнее фильтрует данные по товару `'RGAN01'`.

Имена столбцов в соединениях

Во всех примерах главы столбцы соединяемых таблиц называются одинаково (к примеру, столбцы `cust_id` в таблицах `Customers` и `Orders`). Наличие столбцов с одинаковыми именами не является обязательным требованием, и вы часто будете сталкиваться с базами данных, в которых столбцы называются по-разному. Я сознательно создал демонстрационные таблицы именно такими, чтобы сделать примеры более простыми и понятными.

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, конкретной СУБД, количества данных в таблицах, наличия либо отсутствия индексов и ключей, а также целого ряда других факторов. Следовательно, зачастую бывает целесообразно поэкспериментировать с различными типами запросов для выяснения того, какой из них работает быстрее.

Резюме

Соединения — одно из самых важных и востребованных средств SQL, но их эффективное применение требует знания структуры реляционной базы данных. На этом уроке вы ознакомились с основами построения баз данных и узнали, как создавать соединение по равенству (называемое также внутренним соединением), которое используется чаще всего. На следующем уроке вы научитесь создавать соединения других типов.

Упражнения

1. Напишите инструкцию SQL, которая возвращает имя клиента (`cust_name`) из таблицы `Customers` и соответствующие ему номера заказов (`order_num`) из таблицы `Orders`, сортируя результаты сначала по имени клиента, а затем по номеру заказа. Предложите два варианта запроса: один — с использованием простого синтаксиса соединений по равенству, а второй — с использованием оператора `INNER JOIN`.
2. Давайте доработаем предыдущее упражнение. В дополнение к имени клиента и номеру заказа добавьте третий столбец, `OrderTotal`, содержащий общую стоимость каждого заказа. Это можно сделать двумя способами: путем создания подзапроса к таблице `OrderItems` или путем соединения таблицы `OrderItems` с существующими таблицами с последующим применением итоговой функции. *Подсказка:* следите за тем, где должны использоваться полные имена столбцов.
3. Вернемся к упражнению 2 из урока 11. Напишите инструкцию SQL, возвращающую даты заказов товара `'BR01'`, но на этот раз с использованием простого синтаксиса соединений по равенству. Результат должен получиться таким же, как и на уроке 11.
4. Продолжим в том же духе. Перепишите запрос из упражнения 3 урока 11, на этот раз с использованием синтаксиса ANSI (оператор `INNER JOIN`). Предыдущая инструкция содержала два вложенных запроса. Теперь вам понадобятся два оператора `INNER JOIN`, с которыми вы познакомились на данном уроке. И не забудьте выполнить филь-

трацию по полю prod_id с помощью предложения WHERE.

5. На этот раз ради интереса попробуем применить соединения, итоговые функции и группирование данных. На уроке 10 вы написали запрос для нахождения всех номеров заказов, стоимость которых составляла не менее 1000. Хотелось бы также узнать имена клиентов, сделавших такие заказы. Напишите инструкцию SQL, использующую соединения для получения имени клиента (cust_name) из таблицы Customers и общей стоимости всех его заказов из таблицы OrderItems. *Подсказка:* в соединение придется также включить таблицу Orders, поскольку таблица Customers не связана напрямую с таблицей OrderItems; вместо этого она связана с таблицей Orders, которая, в свою очередь, связана с таблицей OrderItems. Не забудьте применить предложения GROUP BY и HAVING, а также отсортировать результаты по имени клиента. Можно использовать как простой синтаксис соединений по равенству, так и синтаксис ANSI с оператором INNER JOIN. А еще лучше, напишите оба варианта.

УРОК 13

Создание расширенных соединений

На этом уроке вы узнаете о других видах соединений — что они собой представляют и когда они нужны. Вы также узнаете, как применять псевдонимы таблиц и использовать итоговые функции совместно с соединениями.

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 многократного использования одной и той же таблицы в инструкции SELECT.

Рассмотрим следующую инструкцию SELECT. В основном она такая же, как и в примерах предыдущего уро-

ка, только теперь 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 задействуются только в предложении WHERE, но их можно применять и в других местах, например в списке возвращаемых таблиц, в предложении ORDER BY, а также в любой другой части инструкции SELECT.

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 возвращаются клиентскому приложению.

Другие виды соединений

До сих пор мы применяли только простые соединения, которые называются *внутренними* (соединения по равенству). Теперь рассмотрим три других вида соединения: самосоединение, естественное соединение и внешнее соединение.

Самосоединения

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 отправить письма по всем контактным адресам клиентов, которые работают с той же компанией, что и 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 данном решении используется подзапрос (см. урок 11). 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 клиентов, работающих с данной компанией.

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

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
```

В Oracle нет ключевого слова AS

Для пользователей Oracle еще раз напомним о необходимости убирать из инструкций ключевое слово AS.

Анализ ▼

Две таблицы, необходимые для выполнения запроса, в реальности представляют собой одну и ту же таблицу, поэтому таблица Customers появляется в предложении

FROM дважды. И хотя это совершенно допустимо, любые ссылки на таблицу Customers оказались бы неоднозначными, потому что СУБД не знает, на какую именно копию таблицы 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, в предложении SELECT префикс c1 используется для однозначного указания полных имен нужных столбцов. Если этого не сделать, СУБД выдаст сообщение об ошибке, потому что имеются по два столбца с именами cust_id, cust_name и cust_contact. СУБД не знает, какой именно столбец вы имеете в виду (даже если это, в сущности, один и тот же столбец). Первое предложение WHERE объединяет обе копии таблицы, а затем фильтрует данные второй таблицы по столбцу cust_contact, чтобы отобрать только нужные данные.

Самосоединения вместо подзапросов

Самосоединения часто применяются для замены инструкций с подзапросами, которые извлекают данные из той же таблицы, что и внешний запрос. Несмотря на то что конечный результат получается тем же самым, многие СУБД обрабатывают табличные соединения гораздо быстрее, чем подзапросы. Стоит поэкспериментировать с обоими механизмами, чтобы определить, какой вариант запроса работает быстрее.

Естественные соединения

Всякий раз, когда объединяются таблицы, по крайней мере один столбец будет появляться более чем в одной таблице (по нему и выполняется соединение). Обычные соединения (внутренние, которые мы рассмотрели на

предыдущем уроке) возвращают все данные, даже многократные вхождения одного и того же столбца. *Естественное соединение* просто удаляет эти многократные вхождения, и в результате возвращается только один столбец.

Естественным называется такое соединение, в котором извлекаются только уникальные 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';
```

В Oracle нет ключевого слова AS

Пользователи Oracle должны убрать из инструкции 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 соединение, которое мы использовали до сих пор, представляло собой естественное соединение, и, возможно, вам никогда не понадобится внутреннее соединение, не являющееся естественным.

Внешние соединения

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

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

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

Различия в синтаксисе

Важно отметить, что синтаксис внешнего соединения может несколько отличаться в разных СУБД. Различные варианты синтаксиса, описанные далее, охватывают большинство реализаций, но все же, прежде чем начинать работу, обратитесь к документации своей СУБД и уточните, какой синтаксис в ней применяется.

Следующая инструкция SELECT позволяет выполнить простое внутреннее соединение. 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 соединения похож на этот. Для получения имен всех клиентов, включая тех, которые еще не делали заказов, можно воспользоваться следующей инструкцией.

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 соединению, которое мы рассматривали на прошлом уроке, в этой инструкции SELECT используется оператор OUTER JOIN для указания типа соединения (в предложении FROM, а не WHERE). Но, в отличие от внутренних соединений, которые связывают строки двух таблиц, внешние соединения включают в

результат также строки, не имеющие пар. При использовании синтаксиса 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 оператора OUTER JOIN, и LEFT — для таблицы слева). В предыдущем примере используется синтаксис LEFT OUTER JOIN для извлечения всех строк таблицы, указанной в левой части предложения FROM (т.е. таблицы Customers). Чтобы извлечь все строки из таблицы, указанной справа, используйте правое внешнее соединение (RIGHT OUTER JOIN), как показано в следующем примере.

Ввод

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

Внешние соединения в SQLite

SQLite поддерживает левое внешнее соединение, но не правое. К счастью, существует очень простое решение, объясняемое в следующем совете.

Типы внешних соединений

Существуют два основных варианта внешнего соединения: левое и правое. Единственная разница между ними состоит в порядке указания связываемых таблиц. Другими словами, левое внешнее соединение может быть превращено в правое просто за счет перестановки имен таблиц в предложении FROM или WHERE. А раз так, то эти две разновидности внешнего соединения могут заменять друг друга, и решение о том, какое именно из них нужно использовать, определяется личными предпочтениями.

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

Ввод ▼

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

Поддержка полного внешнего соединения

Синтаксис `FULL OUTER JOIN` не поддерживается в MariaDB, MySQL и SQLite.

Использование соединений совместно с итоговыми функциями

Как объяснялось на уроке 9, итоговые функции предназначены для вычисления базовых статистических показателей. Во всех приведенных до сих пор примерах итоговые функции применялись только для одной таблицы, но их можно использовать и по отношению к соединениям.

Рассмотрим пример. Допустим, вы хотите получить список всех клиентов и количество 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 оператор `INNER JOIN` для соединения таблиц `Customers` и `Orders`. 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 соединениями других видов.

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;
```

Вывод ▼

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

Анализ ▼

В этом примере используется левое внешнее соединение 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 соединений

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

- Будьте внимательны при выборе типа соединения. Вероятно, чаще всего вы будете применять внутреннее соединение, хотя в зависимости от ситуации это может быть и внешнее соединение.
- Посмотрите в документации к СУБД, какой именно синтаксис соединений она поддерживает. (Большинство СУБД поддерживает один из вариантов синтаксиса, описанных на этих двух уроках.)
- Проверьте, правильно ли указано условие соединения (независимо от используемого синтаксиса), иначе будут получены неверные данные.

- Не забывайте указывать условие соединения, в противном случае вы получите декартово произведение таблиц.
- Можно включать в соединение несколько таблиц и даже применять для каждой из них свой тип соединения. Несмотря на то что это допустимо и часто оказывается полезным, желательно проверить каждое соединение по отдельности, прежде чем применять их вместе. Это намного упростит поиск ошибок.

Резюме

Этот урок стал продолжением предыдущего, посвященного соединениям. Вначале было показано, как и для чего используют псевдонимы, а затем мы продолжили рассмотрение различных видов соединений, изучив варианты синтаксиса для каждого из них. Вы также узнали, как применять итоговые функции совместно с соединениями и какие правила важно соблюдать при использовании соединений.

Упражнения

1. Напишите инструкцию SQL, которая возвращает имя клиента (поле `cust_name` таблицы `Customers`) и номера всех его заказов (поле `order_num` таблицы `Orders`) с использованием оператора `INNER JOIN`.
2. Модифицируйте предыдущую инструкцию SQL для получения списка всех клиентов, даже тех, кто еще не сделал ни одного заказа.
3. Используйте оператор `OUTER JOIN` для соединения таблиц `Products` и `OrderItems`, чтобы получить отсортированный список названий товаров (`prod_name`) и соответствующих им номеров заказов (`order_num`).
4. Модифицируйте предыдущую инструкцию SQL, чтобы получить общее количество заказов по каждому товару (а не номера заказов).
5. Напишите инструкцию SQL, которая возвращает список поставщиков (поле `vend_id` таблицы `Vendors`) и количество предлагаемых ими товаров, включая поставщиков, у которых нет товаров. Для подсчета количества товаров в таблице `Products` понадобится использовать оператор `OUTER JOIN` и итоговую функцию `COUNT()`. Будьте внимательны: столбец `vend_id` имеется в нескольких таблицах, поэтому при ссылке на него придется указывать полное имя.

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 содержится одна инструкция `SELECT`, с помощью 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 *комбинированными*.

Комбинированные запросы 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.

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 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

SQL-запросы комбинируются с помощью оператора UNION, который позволяет указать несколько инструкций SELECT, возвращая один набор результатов.

Оператор UNION

Использовать оператор UNION достаточно легко. 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 списке IN. Вторая инструкция SELECT использует простую проверку на равенство, чтобы найти все вхождения клиента 'Fun4All'.

Чтобы объединить оба запроса, введите следующую инструкцию.

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 оба запроса и вернуть результат в виде одного набора.

Для сравнения приведем тот же самый запрос, но с использованием не оператора UNION, а нескольких 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.

Ограничения оператора UNION

В стандарте SQL не существует ограничений на количество инструкций SELECT, которые могут быть объединены посредством оператора UNION. Но лучше все же обратиться к документации своей СУБД и убедиться в том, что она не налагает каких-либо ограничений на максимально допустимое число инструкций.

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 какой-либо 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 не менее двух инструкций SELECT, отделенных одна от другой

ключевым словом UNION (таким образом, если в запросе четыре инструкции SELECT, то 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).

Имена столбцов в операторе UNION

Если инструкции SELECT, объединяемые с помощью оператора UNION, содержат различные имена столбцов, то какое из них возвращается в запросе? Например, если одна инструкция содержит предложение SELECT prod_name, а другая — предложение SELECT productname, то как будет назван результирующий столбец?

Ответ прост: используется первое встречающееся имя, поэтому в нашем примере объединенный столбец будет назван prod_name, несмотря на наличие другого имени во второй инструкции SELECT. Это также означает, что первому имени столбца можно назначить псевдоним, который и станет именем результирующего столбца.

У данного поведения есть интересный побочный эффект. Поскольку используется первый набор имен столбцов, только эти столбцы могут указываться при сортировке. В нашем примере это означает, что сортировать результирующую таблицу можно только с помощью предложения ORDER BY prod_name, но не ORDER BY productname, поскольку в таблице не будет столбца productname.

При соблюдении этих основных правил и ограничений комбинированные запросы можно применять для решения любых задач, связанных с извлечением 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 объединяются 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 возвращается 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.

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 — это разновидность оператора UNION, делающая то, что не способны выполнить предложения WHERE. Если требуется 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 объединении запросов с помощью 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 Store  Kim Howard          NULL
Village Toys   John Smith          sales@villagetoys.com
```

Анализ ▼

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

Другие виды комбинированных запросов

Некоторые СУБД поддерживают два дополнительных вида 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 с помощью соединений.

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 запросов в виде одной общей таблицы, включающей или исключающей дубликаты. За счет оператора UNION можно значительно упростить сложные предложения WHERE и запросы, связанные с извлечением данных из нескольких таблиц.

Упражнения

1. Напишите инструкцию SQL, которая объединяет две инструкции SELECT, возвращающие идентификатор товара (prod_id) и его количество (quantity) из таблицы OrderItems, причем одна инструкция должна отбирать строки, в которых количество равно 100, а вторая должна отбирать товары, идентификатор которых начинается с префикса 'BNBG'. Отсортируйте результаты по идентификатору товара.
2. Перепишите предыдущую инструкцию SQL, чтобы в ней использовалось только одно предложение SELECT.
3. Это упражнение может показаться немного бессмысленным, но оно хорошо иллюстрирует принципы объединения таблиц с разными столбцами. Напишите инструкцию SQL, которая возвращает названия товаров (prod_name) из таблицы Products, а также имена клиентов (cust_name) из таблицы Customers. Отсортируйте результаты по названию товара.
4. Что неправильного в следующей инструкции SQL? (Постарайтесь понять это, не выполняя саму инструкцию.)

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

УРОК 15

Добавление данных

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

Способы добавления данных

Несомненно, SELECT — наиболее часто используемая инструкция SQL (именно поэтому мы посвятили ее изучению 14 уроков). Но помимо нее регулярно применяются еще три инструкции, которые необходимо знать. Первая из них — INSERT (с двумя другими мы познакомимся на следующем уроке).

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

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

Мы рассмотрим все вышеперечисленные варианты.

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.

Ключевое слово INTO

В некоторых СУБД ключевое слово INTO после инструкции INSERT необязательное. Однако хорошей практикой считается указание этого ключевого слова даже в тех случаях, когда оно 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 такой 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, то рано или поздно столкнетесь с проблемами.

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

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, только на этот раз имена столбцов заданы в явном виде в круглых скобках после имени таблицы. Когда строка вводится в таблицу, СУБД сопоставляет каждый элемент 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.

Нельзя вставить одну и ту же запись дважды

Если вы попробуете последовательно выполнить обе версии примера, то обнаружите, что во втором случае возникает ошибка, поскольку клиент с идентификатором 1000000006 уже существует. Как обсуждалось на уроке 1, первичный ключ должен быть уникальным, а раз поле cust_id — это первичный ключ, СУБД не позволит вставить две записи с одинаковым идентификатором клиента. То же самое касается и следующего примера. Чтобы протестировать альтернативный вариант инструкции INSERT, необходимо сначала удалить только что добавленную запись (как будет показано в следующей главе).

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 заключается в том, что даже если расположение столбцов в таблице изменится, 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 в явном виде, 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',
```

Всегда указывайте список столбцов

Как правило, инструкция INSERT не используется без явного указания списка столбцов. Благодаря этому повышается вероятность того, что вы сможете успешно выполнить запрос, даже если в таблице произойдут изменения.

Аккуратно используйте предложение VALUES

Независимо от синтаксиса инструкции INSERT, в предложении VALUES должно содержаться правильное количество значений. Если имена столбцов отсутствуют, должно быть приведено значение для каждого столбца таблицы. Если имена столбцов указаны, 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.

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 значения).
- В определении столбца задано значение по умолчанию. Это означает, что, если не указано никакое конкретное значение, будет использовано значение по умолчанию.

Пропуск обязательных значений

Если вы пропускаете столбец, для которого не допускаются значения NULL и не заданы значения по умолчанию, то 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. Но существует и другая форма инструкции INSERT, которую можно применять для добавления результатов запроса SELECT. Такая инструкция называется INSERT SELECT. Как подсказывает ее название, она 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.

Пояснения к примеру

В этом 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 убедитесь в том, что не используются значения cust_id, которые уже существуют в таблице Customers (инструкция INSERT завершится неудачей, 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 пустая, то никакие строки добавлены не будут (и никакое сообщение об ошибке не будет выдано, поскольку подобная операция допустима). Если же таблица содержит данные, то все они будут добавлены в таблицу Customers.

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.

Добавление нескольких строк

Инструкция INSERT обычно добавляет только одну строку. Чтобы добавить несколько строк, нужно выполнить несколько инструкций INSERT. Исключение из этого правила — 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-либо таблицы в новую таблицу (которая создается на лету), можно использовать инструкцию CREATE SELECT (или SELECT INTO в SQL Server).

Не поддерживается в Db2

Db2 не поддерживает использование инструкции CREATE SELECT описанным здесь способом.

В отличие от инструкции INSERT SELECT, посредством которой данные добавляются в уже существующую таблицу, инструкция CREATE SELECT копирует данные в новую таблицу (и, в зависимости от СУБД, может перезаписать таблицу, если она уже существует).

В следующем примере демонстрируется применение инструкции CREATE SELECT.

Ввод ▼

```
CREATE TABLE CustCopy AS SELECT * FROM Customers;
```

В SQL Server синтаксис будет таким.

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).

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 соединения.
- Данные можно добавить только в одну таблицу независимо от того, из скольких таблиц они были извлечены.

Создание копий таблиц

Инструкция `CREATE SELECT` представляет собой прекрасное средство создания копий таблиц для экспериментов с новыми инструкциями SQL. Создав копию, вы получите возможность протестировать инструкции SQL на этой копии, а не на таблицах реальной базы данных.

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.

Упражнения

1. Добавьте самого себя в таблицу Customers с помощью инструкции INSERT. Укажите в явном виде все добавляемые столбцы, но только те, для которых предоставляется значение.
2. Создайте резервные копии таблиц Orders и OrderItems.

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 данных какой-либо таблицы предназначена инструкция UPDATE, которую можно использовать двумя способами:

- обновление определенных строк в таблице;
- обновление всех строк в таблице.

Мы рассмотрим оба способа.

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.

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 простая. 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 идет предложение SET, в котором указывается новое значение столбца. В данном случае обновляется значение столбца cust_email:

```
SET cust_email = 'kim@thetoystore.com'
```

Последним идет предложение WHERE, которое сообщает СУБД, какая строка подлежит обновлению. При отсутствии такого предложения СУБД обновит 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.

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).

Предложение FROM

Некоторые СУБД 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 (при условии, что 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 запись 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.

Мы рассмотрим оба способа.

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.

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, инструкция UPDATE очень простая. К счастью, инструкция DELETE еще проще.

Следующая инструкция удаляет одну строку из таблицы Customers.

Ввод ▼

```
DELETE FROM Customers
WHERE cust_id = 1000000006;
```

В предложении DELETE FROM указывается имя таблицы, из которой удаляются данные. Предложение WHERE фильтрует строки, определяя, какие из них должны быть удалены. В нашем примере удаляется строка, относящаяся к клиенту 1000000006. Если бы предложение WHERE было пропущено, инструкция удалила бы все строки из таблицы.

Научитесь работать с внешними ключами

На уроке 12 вы ознакомились с концепцией соединений и узнали, что в объединяемых таблицах должны быть общие поля. Но можно заставить СУБД создавать связи между таблицами с помощью внешних ключей (примеры их определений содержатся в приложении А). Когда имеются внешние ключи, СУБД использует их, чтобы гарантировать ссылочную целостность базы данных. Например, если попытаться добавить новый товар в таблицу Products, указав неизвестный идентификатор поставщика, СУБД не позволит этого сделать, поскольку столбец vend_id представляет собой внешний ключ для таблицы Vendors. Вы спросите, какое отношение все это имеет к инструкции DELETE? Положительным побочным эффектом использования внешних ключей для поддержания ссылочной целостности становится 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, то инструкция DELETE завершится неудачей, и будет выдано сообщение об ошибке. Это веская причина для того, чтобы всегда определять внешние ключи.

Ключевое слово FROM

В некоторых СУБД ключевое слово FROM после инструкции DELETE необязательное. Однако хорошей практикой считается указание этого ключевого слова даже в тех случаях, когда оно не требуется. Поступая таким образом, вы обеспечите переносимость кода между разными СУБД.

В инструкции DELETE нельзя задавать ни имена столбцов, ни метасимволы. Она удаляет строки целиком, а не отдельные столбцы. Для удаления конкретного столбца следует использовать инструкцию UPDATE.

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.

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 предложением WHERE, и на то есть веская причина. Если пропустить предложение WHERE, то инструкция будет применена по отношению ко всем строкам таблицы. Другими словами,

если выполнить инструкцию UPDATE без предложения WHERE, то 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 следует большинство разработчиков баз данных.

- Никогда не выполняйте инструкцию UPDATE или DELETE без предложения WHERE, если только 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 при любой возможности. (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 и убедитесь в том, что оно правильно фильтрует записи. Всегда есть риск по ошибке 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 рисками, которыми чревато их применение. Вы также узнали, почему настолько важно указывать предложение WHERE в инструкциях UPDATE и DELETE, и изучили основные правила, которым необходимо следовать, чтобы по неосторожности не повредить данные.

Упражнения

1. Аббревиатуры штатов США должны записываться прописными буквами. Напишите инструкцию SQL, которая обновляет все адреса, относящиеся к США, чтобы как штат поставщика (поле vend_state таблицы Vendors), так и штат клиента (поле cust_state таблицы Customers) были записаны в верхнем регистре.
2. В упражнении 1 на предыдущем уроке вы добавили самого себя в таблицу Customers. Теперь удалите свою запись. Не забудьте использовать предложение WHERE (предварительно протестировав его с помощью инструкции SELECT, прежде чем выполнять инструкцию DELETE), иначе вы рискуете удалить всех клиентов!

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.

Есть два способа создания таблиц.

- Большинство СУБД содержит 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 интерактивные инструменты, в действительности вся работа выполняется инструкциями SQL. Но вам не приходится писать их, поскольку СУБД создает и выполняет их незаметно для вас (то же самое справедливо и для процедуры изменения существующих таблиц).

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.

Различия в синтаксисе

Точный синтаксис инструкции CREATE TABLE может немного отличаться в различных СУБД. Обязательно обратитесь к документации своей СУБД за дополнительной информацией и выясните, какой синтаксис и какие возможности она поддерживает.

Примеры для конкретных СУБД

Примеры инструкций CREATE TABLE для конкретных СУБД содержатся в сценариях создания демонстрационных таблиц, рассмотренных в приложении А.

Создание простой таблицы

Чтобы создать таблицу с помощью инструкции CREATE TABLE, необходимо указать следующие данные:

- имя новой таблицы;
- имена и определения столбцов таблицы, разделенные запятыми;
- в некоторых СУБД также требуется, чтобы было указано расположение таблицы (т.е. в какой конкретно базе данных она создается).

Следующая инструкция создает таблицу Products, используемую в примерах книги.

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 (включающее все ее столбцы) берется в круглые скобки. Определения столбцов разделяются запятыми. Создаваемая в данном примере таблица состоит из пяти столбцов. Определение каждого столбца начинается с его имени (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 B приведен перечень основных типов данных в SQL.) Инструкция в целом заканчивается точкой с запятой, которая ставится после закрывающей круглой скобки.

Ранее уже говорилось о том, что синтаксис инструкции CREATE TABLE зависит от СУБД, и данный пример наглядно демонстрирует это. В большинстве СУБД инструкция будет работать в приведенной форме, но 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 A).

Замена существующих таблиц

Когда вы создаете новую таблицу, указываемое вами имя не должно существовать в базе данных, иначе будет выдано сообщение об ошибке. Чтобы избежать случайной перезаписи, СУБД требует, чтобы вы вначале вручную удалили таблицу (подробности описаны далее), а затем вновь создали ее, а не просто перезаписали.

Форматирование инструкции

Помните о том, что пробелы игнорируются инструкциями SQL. Инструкцию можно ввести в одной длинной строке или разбить ее на несколько строк — результат будет одним и тем же. 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 необязательно, но все же настоятельно рекомендуется.

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 поддерживает 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 ▼

С помощью этой инструкции создается таблица Orders, используемая в примерах книги. Таблица состоит из трех столбцов: номер и дата заказа, а также идентификатор клиента. Все три столбца необходимы, 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

С помощью этой инструкции создается таблица Vendors, также используемая в примерах книги. Столбцы с идентификатором и именем поставщика необходимы, поэтому оба они 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.

Указание значения NULL

Во многих СУБД отсутствие спецификации NOT NULL трактуется как NULL. Но это не всегда так. В некоторых СУБД наличие ключевого слова NULL обязательно, и если оно не указано, то генерируется сообщение об ошибке. Обратитесь к документации своей СУБД, чтобы получить информацию о синтаксисе данной инструкции.

Первичные ключи и значения NULL

На уроке 1 говорилось о том, что первичные ключи — это столбцы, значения которых уникальным образом идентифицируют каждую строку таблицы. Столбцы, допускающие отсутствие значений, не могут использоваться в качестве уникальных идентификаторов.

Что такое NULL?

Не путайте значения NULL с пустыми строками. NULL означает отсутствие значения; это не пустая строка. Если указать в коде '' (две одинарные кавычки, между которыми ничего нет), то это значение можно будет ввести в столбец типа NOT NULL. Пустая строка представляет собой допустимое значение; она не служит указанием на 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 ▼

С помощью 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()`, пользователям Oracle нужно вводить дату как `DEFAULT SYSDATE`, а пользователям SQL Server — как `DEFAULT GETDATE()`. К сожалению, способ получения системной даты в каждой СУБД свой. В табл. 17.1 приведен синтаксис для основных СУБД (если ваша СУБД не представлена в этом списке, обратитесь к ее документации).

Таблица 17.1. Получение системной даты

СУБД	Функция/переменная
Db2	`CURRENT_DATE`
MySQL	`CURRENT_DATE()` или `Now()`
Oracle	`SYSDATE`
PostgreSQL	`CURRENT_DATE`
SQL Server	`GETDATE()`
SQLite	`date('now')`

Использование значений DEFAULT вместо NULL

Многие разработчики баз данных применяют значения `DEFAULT` вместо столбцов `NULL`, особенно в столбцах, которые будут использованы в итоговых вычислениях или при группировании 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, необходимо 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 ▼

С помощью 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.

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 ее удаление).
5. Присвоение новой таблице имени, которое ранее принадлежало старой таблице.
6. Восстановление триггеров, хранимых процедур, индексов и внешних ключей, если это необходимо.

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.

Используйте инструкцию ALTER TABLE с осторожностью

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 всегда будет возможность 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 простая операция. 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.

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, то СУБД заблокирует операцию до тех пор, пока не будет удалена данная связь. Такое поведение приветствуется, поскольку благодаря этому можно воспрепятствовать ошибочному удалению нужных таблиц.

Переименование таблиц

В разных СУБД переименование таблиц осуществляется по-разному. 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, инструкция `ALTER TABLE` служит 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.

Упражнения

1. Добавьте столбец для адресов сайтов (`vend_web`) в таблицу `Vendors`. Текстовое поле должно быть достаточно большим, чтобы вместить URL-адрес.
2. Используйте инструкцию `UPDATE` для обновления таблицы `Vendors`, добавив в нее адрес какого-нибудь сайта (можете использовать любой адрес).

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.

Представления в SQLite

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 соединения 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 соединения таблиц.

Согласованность СУБД

Синтаксис создания представлений примерно одинаков во всех основных СУБД.

Зачем нужны представления

Выше был рассмотрен один из случаев использования представления. Чаще всего 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.)

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 соединениями и фильтрами или если были задействованы вложенные представления, то производительность СУБД резко снизится. Рекомендуется провести тестирование, прежде чем создавать приложения, в которых интенсивно используются представления.

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 обновляются автоматически.

Правила и ограничения представлений

Прежде чем создавать представления, следует узнать о связанных с ними ограничениях. К сожалению, ограничения 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.

- У представлений, как и у таблиц, должны быть уникальные имена (они не могут быть названы так же, как другие таблицы или представления).
- Не существует ограничения на количество создаваемых представлений.
- 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 у каждого возвращаемого столбца было имя. 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 частью данного представления. Например, если представление возвращает только информацию о клиентах, имеющих адреса электронной почты, то 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.

Обратитесь к документации своей СУБД

Список этих правил довольно длинный, и документация к вашей СУБД почти наверняка содержит еще какие-то правила. Придется потратить время на изучение подобных ограничений, прежде чем приступать к созданию представлений.

Создание представлений

Итак, вы узнали, 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 нового представления, еще не существующего в базе данных.

Удаление представлений

Для удаления представления предназначена инструкция DROP VIEW. Ее синтаксис очень простой:

```
DROP VIEW имя_представления;
```

Чтобы перезаписать (или обновить) представление, вначале нужно применить по отношению к нему инструкцию DROP VIEW, а затем заново создать представление.

Использование представлений для упрощения сложных соединений

Чаще всего представления используются для упрощения сложных запросов, и нередко это относится к соединениям. 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 ▼

С помощью этой инструкции создается представление ProductCustomers, которое объединяет три таблицы для получения списка клиентов, заказавших какой-нибудь товар. Если затем выполнить инструкцию SELECT * FROM ProductCustomers, то она вернет список всех клиентов, сделавших заказы.

Для получения списка клиентов, заказавших товар 'RGAN01', необходимо выполнить следующий запрос.

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.

Создание повторно используемых представлений

Хорошей идеей будет 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, другой распространенный случай использования представлений — 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)
```

Ниже показана аналогичная инструкция, но в ней применяется оператор || (см. урок 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. 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 ▼

С помощью этой инструкции создается представление, использующее тот же самый запрос, что и в предыдущей инструкции SELECT. Чтобы извлечь данные, требуемые 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)
```

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 содержит инструкцию SELECT, синтаксис которой должен четко соответствовать правилам и ограничениям, принятым в конкретной 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 значение 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
```

Предложения WHERE

Если предложение WHERE используется для извлечения данных из представления, то 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, с помощью 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.

Упражнения

1. Создайте представление CustomersWithOrders, включающее все столбцы из таблицы Customers, но содержащее записи только тех клиентов, которые разместили заказы. *Подсказка:* выполните соединение с таблицей Orders для фильтрации клиентов, после чего используйте инструкцию SELECT для отбора нужных данных.
2. Что неправильного в следующей инструкции SQL? (Постарайтесь понять это, не выполняя саму инструкцию.)

```
CREATE VIEW OrderItemsExpanded AS
SELECT order_num,
       prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM OrderItems
ORDER BY order_num;
```

УРОК 19

Хранимые процедуры

На этом уроке вы узнаете, что такое хранимые процедуры и как они применяются. Вы также ознакомитесь с базовым синтаксисом создания и запуска хранимых процедур.

Что такое хранимые процедуры

Большинство SQL-запросов, которые мы до сих пор создавали, были простыми в том смысле, что в них применялась единственная инструкция по отношению к одной или нескольким таблицам. Но далеко не всегда запросы настолько просты — зачастую приходится использовать несколько инструкций для выполнения сложной операции. 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 понятна. Решение подобной задачи потребует применения множества инструкций SQL по отноше-

нию ко многим таблицам. Более того, сами инструкции и их порядок не постоянны: они могут (и будут) изменяться в зависимости от того, какие товары имеются на складе, а какие нужно дозаказывать.

Как написать 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 этом нуждается).

Альтернативный вариант — создать *хранимую процедуру*, которая представляет собой набор из нескольких инструкций, сохраненный для последующего выполнения.

Отсутствие поддержки в SQLite

Хранимые процедуры не поддерживаются в SQLite.

Введение в тему

Хранимые процедуры — тема достаточно сложная, и полностью рассмотреть ее можно только в отдельной книге. На данном уроке вы не научитесь всему, что необходимо знать о хранимых процедурах. Скорее, это введение в тему, призванное ознакомить вас с тем, что собой представляют хранимые процедуры и что с их помощью можно делать. По сути, приведенные 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.

- Они применяются для упрощения сложных запросов (как уже говорилось выше) за счет инкапсуляции инструкций в один блок, удобный для выполнения.
- Они нужны 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 одну и ту же хранимую процедуру, значит, будет выполняться один и тот же код.
- Следствием этого становится предотвращение ошибок. Чем больше действий необходимо выполнить, тем выше вероятность появления ошибок. Отсутствие ошибок обеспечивает целостность данных.
- Они помогают упростить управление изменениями. Если таблицы, имена столбцов или деловые правила изменяются, обновлять приходится только код хранимой процедуры и ничего больше.
- Следствием этого является повышение безопасности. Предоставление доступа к базе данных 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. Но 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 не придется менять.
- Хранимые процедуры сложнее в написании, чем основные инструкции SQL, и их создание 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 множеством хранимых процедур, которые предназначены для управления базами данных и таблицами. Обратитесь к документации своей СУБД, чтобы получить больше информации об этом.

Не можете писать хранимые процедуры? Тогда просто используйте готовые

Во многих СУБД различаются меры безопасности, 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 — требует указания имени хранимой процедуры и передаваемых ей аргументов. 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. У процедуры четыре параметра: идентификатор поставщика (первичный ключ таблицы Vendors), название товара, цена и описание. Эти четыре параметра соответствуют четырем переменным, определенным в хранимой процедуре. 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 (первичный ключ 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 в

данном примере используется хранимая процедура. Она выполняет следующие действия:

- проверяет правильность передаваемых данных, гарантируя 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 следующее:

- необязательные аргументы со значениями по умолчанию, присваиваемыми в случае, если аргумент не задан пользователем;
- именованные параметры, указываемые в виде пар *`параметр=значение`*;
- выходные параметры, позволяющие хранимой процедуре обновлять переменную, используемую вызывающим приложением;
- 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 количество 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 имеет один аргумент: `ListCount`. Вместо того чтобы передавать значение в хранимую процедуру, данный аргумент возвращает значение из нее. Подобное поведение аргумента определяется ключевым словом `OUT`. 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 не имеет 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.

Рассмотрим еще один пример, но на этот раз будем добавлять новый заказ в таблицу Orders. Данный пример рассчитан только на SQL Server, однако он наглядно демонстрирует, 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 имеет один аргумент: идентификатор кли-

ента, 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 (с помощью 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.

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). Преимущества очевидны: 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 закомментировать код — поставить в начале строки символы `--` (два дефиса). Некоторые СУБД поддерживают и альтернативный синтаксис комментариев, но синтаксис `--` универсален, поэтому лучше использовать его.

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;
```

Анализ ▼

Данная хранимая процедура также создает новый заказ в таблице `Orders`. Но на этот раз СУБД сама генерирует номер заказа. Большинство СУБД поддерживает 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 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 дополнительные 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 поддержание 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 считать реляционными.

Хороший пример — таблицы заказов, с которыми мы работали на всех предыдущих 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.

Процесс добавления нового заказа заключается в выполнении следующих действий.

1. Проверка, содержится ли информация о клиенте в базе данных. Если нет, такая информация добавляется.
2. Получение идентификатора клиента.
3. Добавление строки в таблицу `Orders` и связывание ее с идентификатором клиента.
4. Получение идентификатора нового заказа, присвоенного ему в таблице `Orders`.
5. 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, или 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 строка была добавлена в таблицу `Orders`, но до того, как будут добавлены строки в таблицу `OrderItems`? Теперь в базе данных появится 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 узнаете об этом.

Как решить указанную 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, то 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 в реальности 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 таблицы `OrderItems` и `Orders` 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 результатов 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).

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` и `DROP`. Их можно задействовать в транзакциях, но если понадобится выполнить откат, то действие этих инструкций аннулировано не будет.

Управление транзакциями

Теперь, когда вы знаете, что такое обработка транзакций, перейдем к управлению транзакциями.

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 (замените ... кодом запроса).

Ввод ▼

```
BEGIN TRANSACTION
...
COMMIT TRANSACTION
```

Анализ ▼

В этом примере все инструкции, заключенные между фразами `BEGIN TRANSACTION` и `COMMIT TRANSACTION`, должны быть или выполнены целиком, или не выполнены вообще.

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 СУБД 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 в составе транзакций 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 ра-

ботаете. 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, то 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, то для сохранения данных выполняется инструкция `COMMIT`.

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 — это 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, — это не инструкция SELECT, а результирующий набор, выборка, полученная в результате выполнения инструкции SELECT. После того как курсор сохранен, приложения могут "прокручивать" (просматривать) его строки в прямом или обратном порядке, когда возникает такая необходимость.

Поддержка в SQLite

В SQLite поддерживается разновидность курсоров, называемая *шагами*. Основные концепции, рассматриваемые на этом уроке, применимы к шагам, но синтаксис будет совершенно иным.

В различных СУБД поддерживаются 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.).
- 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.

Работа с курсорами

Работу с курсором можно разбить на несколько этапов.

- Прежде чем курсор сможет быть использован, его следует объявить (определить). В ходе этого процесса не происходит извлечения 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 занимаемые 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 предложениями. Чтобы показать, как это работает, мы создадим курсор, который будет извлекать список всех клиентов, не имеющих адреса электронной почты. Такой курсор будет частью приложения, позволяющего менеджеру вводить недостающие адреса.

Ниже приведена версия 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 поле содержит 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 указанный 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 (например, в переменной). 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 расширенные средства 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*.

На предыдущих уроках мы неоднократно говорили о реляционных таблицах и ссылочной целостности. 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`. Аналогичным образом каждый клиент, упомянутый в таблице `Orders`, должен быть представлен и в таблице `Customers`.

Несмотря на то что можно проводить соответствующие проверки перед вводом новых строк (выполняя инструкцию SELECT для другой таблицы, чтобы удостовериться в правильности нужных значений), лучше избегать этого по следующим причинам.

- Если правила, обеспечивающие целостность базы данных, навязываются на клиентском уровне, то 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).

Ограничения зависят от СУБД

Существуют различные виды 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 рассказывалось 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. Наличие уникального идентификатора 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 первичным ключом, 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 допускаются 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`.

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 первичного ключа 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 отдельную 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`. Таким образом, у каждого клиента есть свой уникальный идентификатор. Номер заказа представляет собой первичный ключ 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, то будут 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 клиентских идентификаторов 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 хранить 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)
);
```

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 сделать это заключается 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, разрешается удалять все связанные со строкой данные при ее удалении из таблицы. Например, если разрешено каскадное удаление и имя клиента удаляется из таблицы `Customers`, то все связанные с его заказами строки удаляются автоматически.

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)
```

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 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 открыто доступной). Поэтому каждому сотруднику присваивается уникальный идентификатор (первичный ключ) в дополнение к его номеру социального страхования.

Поскольку идентификатор сотрудника служит первичным ключом, можно быть уверенным в том, что он уникален. А для того чтобы СУБД проверила уникальность каждого номера социального страхования (исключив вероятность опечатки при вводе, когда для сотрудника указывается чужой номер), необходимо задать ограничение `UNIQUE` для столбца, в котором содержатся номера социального страхования.

Синтаксис ограничений уникальности напоминает синтаксис других ограничений: в определении таблицы указывается ключевое слово `UNIQUE` или отдельно задается спецификация `CONSTRAINT`.

Ограничения на значения столбца

Ограничения на значения столбца нужны для того, чтобы данные в столбце (или наборе столбцов) соответствовали требуемым критериям. Вот наиболее распространенные ограничения:

- **ограничение максимального и минимального значений** — например, для предотвращения появления заказов на 0 (нуль) товаров (несмотря на то что 0 — допустимое число);
- **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 будет самостоятельно отвергать некорректные данные.

В следующем примере налагается ограничение на значения столбца `quantity` таблицы `OrderItems`, 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.

Если необходимо проконтролировать тот факт, что в столбце с обозначением пола содержится только буква 'M' или 'F', добавьте следующую строку в инструкцию ALTER TABLE.

Ввод ▼

```
ADD CONSTRAINT CHECK (gender LIKE '[MF]')
```

Пользовательские типы данных

В некоторых СУБД пользователи 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 неизвестен). Такой тип данных можно указывать в определениях таблиц. Преимущество пользовательских типов данных заключается 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. Узнайте в документации к 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 идея. Чем больше объем текста, в котором нужно провести поиск, тем больше времени требуется на выявление мест вхождения нужных данных.

Именно поэтому книги снабжают *предметным указателем*. Это список ключевых слов, терминов и терминологических оборотов, 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? В общем-то, 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, то 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 сде-

лали 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 их 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 столбцы одинаковым образом, эти столбцы 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 названиями товаров в таблице `Products`.

Ввод ▼

```
CREATE INDEX prod_name_ind
ON Products (prod_name);
```

Пересмотр индексов

Эффективность индексов снижается, если в таблицу добавляются данные или происходит их обновление. Многие администраторы баз данных обнаруживают, что изначально идеальный набор индексов перестает быть таковым после нескольких месяцев работы с базой данных. Целесообразно регулярно пересматривать индексы и, в случае необходимости, перестраивать их.

Анализ ▼

Каждый индекс должен обладать уникальным именем. В данном случае оно определено как `prod_name_ind`. Предложение `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.

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 конкретным 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, связанный с инструкциями 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 журнал с целью регистрации каждого случая обновления или удаления строки);

- дополнительная проверка и, в случае необходимости, отмена ввода данных (например, если необходимо удостовериться в том, что доступный клиенту лимит кредита не превышен, 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, переводящий 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;
```

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 несанкционированного просмотра. В то же время данные должны быть всегда доступны для определенных пользователей, поэтому большинство СУБД предоставляет в распоряжение администраторов баз данных инструменты, с помощью 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 это именно он и что ему разрешено проводить операции, которые он собирается выполнить. Одни СУБД задействуют для этого средства безопасности операционной системы, другие хранят свои собственные списки пользователей и паролей, третьи интегрируются с внешними серверами служб 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 с помощью 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 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. 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 друг с другом и как распределены данные по столбцам, 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 в примерах книги, являются частью системы приема заказов воображаемого дистрибьютора игрушек. Эти таблицы служат для решения нескольких задач:

- управление списками поставщиков;
- управление каталогами товаров;
- управления списками клиентов;
- регистрация заказов от клиентов.

Всего требуется пять таблиц (они тесно связаны между собой в рамках схемы реляционной базы данных). В следующих разделах описана каждая из них.

Упрощенные примеры

Таблицы, используемые в книге, нельзя назвать полными. В реальной системе регистрации заказов хранилось бы множество других данных, не включенных в демонстрационные таблицы (например, платежные реквизиты, номера инвойсов, трекинг-номера 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. Для каждой таблицы приводится список столбцов вместе с 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 этой таблице в качестве первичного ключа следует использовать столбец `vend_id`.

Таблица `Products`

В таблице `Products` (табл. А.2) содержится перечень товаров. Каждый товар имеет уникальный идентификатор (столбец `prod_id`) и связан с соответствующим поставщиком через столбец `vend_id` (уникальный идентификатор поставщика в таблице `Vendors`).

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 этой 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, каждому из которых назначен 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
`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 этой 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 име-

ет уникальный номер (столбец order_num) и связан с соответствующим клиентом через столбец cust_id (уникальный идентификатор 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 этой 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, причем 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 идентифицируется по номеру заказа и порядковому номеру в заказе (первый элемент заказа, второй и т.д.). Элемент заказа связан с соответствующим ему заказом через столбец order_num (уникальный идентификатор 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 (применяется последовательная нумерация)
`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 этой 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 баз данных часто используют диаграммы отношений, демонстрирующие, как связаны между собой все таблицы. Помните, что отношения определяются внешними ключами, описанными в табл. А.1–А.5. На рис. А.1 приведена диаграмма отношений для пяти демонстрационных таблиц.

Получение демонстрационных таблиц

Чтобы попрактиковаться в выполнении примеров книги, вам понадобится набор заполненных таблиц. Все необходимое можно найти на сайте книги (см. введение),

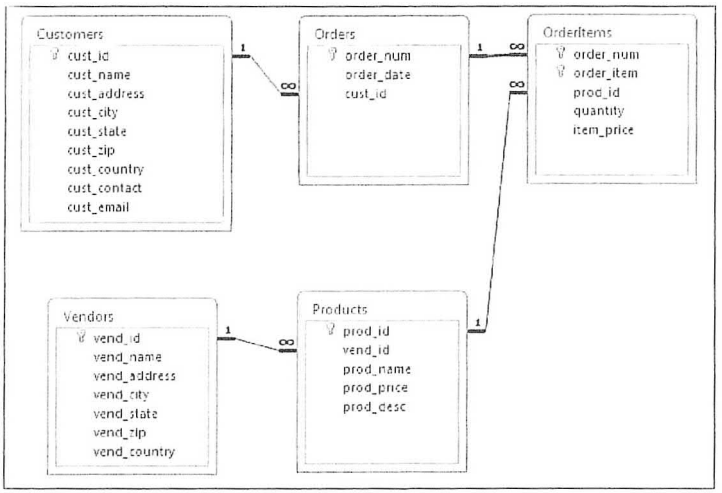

Рис. А.1. *Диаграмма отношений между демонстрационными таблицами*

где доступны готовые базы данных или сценарии для их создания. Для каждой СУБД имеются 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`, которые необходимы для заполнения демонстрационных таблиц.

Инструкции SQL, содержащиеся в этих файлах, зависят от СУБД, поэтому выполняйте только те сценарии, которые соответствуют вашей СУБД. Сценарии предназначены лишь для удобства читателей, и автор книги не несет никакой ответственности за возможные проблемы при их использовании.

На момент публикации книги были доступны сценарии для следующих СУБД:

- IBM Db2 (включая Db2 on Cloud);
- Microsoft SQL Server (включая Microsoft SQL Server Express);
- MariaDB;
- MySQL;
- Oracle (включая Oracle Express);
- PostgreSQL;
- SQLite.

Файлы примеров для SQLite

В SQLite данные хранятся в одном файле. При желании можете использовать сценарии *create.txt* и *populate.txt* для создания собственного файла данных SQLite. Но помимо этого для SQLite предоставлен готовый файл, который можно скачать из архива примеров.

По мере необходимости этот список может быть дополнен другими СУБД.

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, прежде чем заполнять таблицы.

Инструкции по конфигурированию для конкретных СУБД

Действия по конфигурированию 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

Синтаксис инструкций SQL

Для того чтобы помочь вам быстро узнать нужный синтаксис, в этом приложении перечислены основные инструкции 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, необязательны.
- 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` (см. урок 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 [параметры] [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 (см. урок 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. (см. уроки 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 либо из представлений (см. уроки 2–14).

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 В

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, в столбцах с данными числового типа будут допускаться 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 проектировании таблиц обращайте особое внимание на используемые в них типы данных. Некорректные типы данных способны существенно замедлить работу базы данных. Изменение типов данных для уже имеющихся и заполненных столбцов — задача нетривиальная (к тому же чреватая потерей данных).

В одном приложении невозможно дать исчерпывающую информацию по типам данных и способам их ис-

пользования. Здесь рассмотрены лишь основные типы данных, рассказано, для чего они нужны, и указаны возможные проблемы совместимости.

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.

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 видов: фиксированной и переменной длины (табл. В.1).

Строки фиксированной длины могут состоять из фиксированного количества символов, и это количество определяется при создании таблицы. Например, можно разрешить ввод 30 символов в столбец, предназначенный для хранения имен, или 11 символов в столбец с номером 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 имени, рассчитанном на ввод 30 символов, будет сохранено ровно 30 символов (в случае необходимости текст дополняется пробелами).

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 типы не имеют никаких ограничений. В любом случае сохраняются только заданные символы (и никаких дополнительных).

Таблица B.1. Строковые типы данных

Тип данных	Описание
`CHAR`	Строка фиксированной длины, состоящая от 1 до 255 символов. Ее размер должен быть определен на этапе создания таблицы
`NCHAR`	Разновидность типа данных `CHAR`, предназначенная для поддержки многобайтовых символов или символов Unicode (точная спецификация зависит от СУБД)
`NVARCHAR`	Разновидность типа данных `TEXT`, предназначенная для поддержки многобайтовых символов или символов Unicode (точная спецификация зависит от СУБД)
`TEXT` (также может называться `LONG`, `MEMO` или `VARCHAR`)	Текст переменной длины

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 переменной длины (или переменную часть столбца). (Индексы рассматривались на уроке 22.)

Использование кавычек

Независимо от типа строковых данных строка всегда должна быть заключена в одинарные кавычки.

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, то будет сохранено число 1234, и вы потеряете одну цифру.

Основное правило таково: если число используется в вычислениях (итоговых сумм, средних значений и т.п.), то его следует хранить в столбце, предназначенном для числовых данных. Если же оно используется в качестве строкового литерала (пусть он и состоит только из цифр), то 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 вещественных чисел (и дробей), а некоторые — только целые числа. В табл. B.2 перечислены наиболее часто используемые числовые типы данных. Не все СУБД следуют соглашениям о наименовании и описаниям, приведенным в таблице.

Таблица В.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 числа в диапазоне от –2 147 483 648 до 2 147 483 647
REAL	4-байтовые числа с плавающей запятой
SMALLINT	2-байтовые целые числа в диапазоне от –32 768 до 32 767
TINYINT	1-байтовые целые числа в диапазоне от 0 до 255

Кавычки не используются

В отличие от строковых типов данных, числа никогда не заключаются в кавычки.

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. B.3). Аналогично числовым типам, в большинстве СУБД имеется несколько типов данных даты и времени, каждый со своим диапазоном и степенью точности.

Таблица B.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

Формат даты

Не существует стандартного способа указания даты, который подходил бы для любой СУБД. В большинстве СУБД поддерживается формат вида `2020-12-30` или `Dec 30th, 2020`, но даже эти значения могут оказаться проблемой для некоторых СУБД. Обязательно обратитесь к документации своей СУБД и узнайте список поддерживаемых ею форматов даты.

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 '2020-12-30'}` для значений дат, `{t '21:46:29'}` для значений времени и `{ts '2020-12-30 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 остальных типов данных, рассмотренных нами до сих пор и предназначенных для конкретного применения, бинарные типы могут содержать любые данные, даже информацию в двоичном виде, в частности графические изображения, мультимедийные объекты и текстовые документы (табл. В.4).

Сравнение типов данных

Чтобы увидеть реальный пример и сравнить типы данных в различных СУБД, воспользуйтесь сценариями создания демонстрационных таблиц (см. приложение А). Просмотрев сценарии, предназначенные для разных СУБД, вы воочию убедитесь в том, насколько сложна задача согласования типов данных.

Таблица B.4. Бинарные типы данных

Тип данных	Описание
`BINARY`	Двоичные данные фиксированной длины (максимальная длина может быть от 255 до 8000 байт, в зависимости от СУБД)
`LONG RAW`	Двоичные данные переменной длины (до 2 Гбайт)
`RAW` (в некоторых СУБД называется `BINARY`)	Двоичные данные фиксированной длины (до 255 байт)
`VARBINARY`	Двоичные данные переменной длины (максимальная длина может быть от 255 до 8000 байт, в зависимости от СУБД)

ПРИЛОЖЕНИЕ Г

Зарезервированные слова SQL

В инструкциях SQL применяется множество ключевых слов, 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, поддерживаемых в основных СУБД. Учитывайте следующее.

- Ключевые слова сильно зависят от конкретной СУБД, поэтому не все приведенные ниже слова используются во всех СУБД.
- Во многих СУБД имеется расширенный перечень ключевых слов SQL, включающий термины, которые специфичны для конкретной реализации языка. Большинство таких слов не представлено ниже.
- Чтобы обеспечить совместимость и переносимость базы данных, следует избегать применения в ее схеме 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
TEMP
TEMPORARY
TEXT
TEXTSIZE
THEN
TIME
TIMESTAMP
TO
TOP
TRAILING
TRAN
TRANSACTION
TRANSLATE
TRIGGER
TRIM
TRUE
TRUNCATE
TYPE
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

ПРИЛОЖЕНИЕ Д

Ответы на упражнения

Урок 2

1.

```
SELECT cust_id
FROM Customers;
```

2.

```
SELECT DISTINCT prod_id
FROM OrderItems;
```

3.

```
SELECT *
# SELECT cust_id
FROM Customers;
```

Урок 3

1.

```
SELECT cust_name
FROM Customers
ORDER BY cust_name DESC;
```

2.

```
SELECT cust_id, order_num
FROM Orders
ORDER BY cust_id, order_date DESC;
```

3.

```
SELECT quantity, item_price
FROM OrderItems
ORDER BY quantity DESC, item_price DESC;
```

4.

```
SELECT vend_name,
FROM Vendors
ORDER vend_name DESC;
```

После имени vend_name не должно быть запятой (она используется только для разделения столбцов). Кроме того, в предложении ORDER пропущено ключевое слово BY.

Урок 4

1.

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_price = 9.49;
```

2.

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_price >= 9;
```

3.

```
SELECT DISTINCT order_num
FROM OrderItems
WHERE quantity >= 100;
```

4.

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price BETWEEN 3 AND 6
ORDER BY prod_price;
```

Урок 5

1.

```
SELECT vend_name
FROM Vendors
WHERE vend_country = 'USA' AND vend_state = 'CA';
```

2.

```
-- Решение 1
SELECT order_num, prod_id, quantity
FROM OrderItems
WHERE (prod_id = 'BR01' OR prod_id = 'BR02' OR
       prod_id = 'BR03')
  AND quantity >= 100;

-- Решение 2
SELECT order_num, prod_id, quantity
FROM OrderItems
WHERE prod_id IN ('BR01', 'BR02', 'BR03')
  AND quantity >= 100;
```

3.

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price >= 3 AND prod_price <= 6
ORDER BY prod_price;
```

4.

```
SELECT vend_name
FROM Vendors
ORDER BY vend_name
WHERE vend_country = 'USA' AND vend_state = 'CA';
```

Предложение `ORDER BY` должно стоять после предложения `WHERE`.

Урок 6

1.

```
SELECT prod_name, prod_desc
FROM Products
WHERE prod_desc LIKE '%toy%';
```

2.

```
SELECT prod_name, prod_desc
FROM Products
WHERE NOT prod_desc LIKE '%toy%'
ORDER BY prod_name;
```

3.

```
SELECT prod_name, prod_desc
FROM Products
WHERE prod_desc LIKE '%toy%'
  AND prod_desc LIKE '%carrots%';
```

4.

```
SELECT prod_name, prod_desc
FROM Products
WHERE prod_desc LIKE '%toy%carrots%';
```

Урок 7

1.

```
SELECT vend_id,
       vend_name as vname,
       vend_address AS vaddress,
       vend_city AS vcity
FROM Vendors
ORDER BY vname;
```

2.

```
SELECT prod_id, prod_price,
       prod_price*0.9 AS sale_price
FROM Products;
```

Урок 8

1.

```
-- DB2, PostgreSQL
SELECT cust_id, cust_name,
       UPPER(LEFT(cust_contact, 2)) ||
```

```
       UPPER(LEFT(cust_city, 3))
       AS user_login
FROM Customers;

-- Oracle, SQLite
SELECT cust_id, cust_name,
       UPPER(SUBSTR(cust_contact, 1, 2)) ||
       UPPER(SUBSTR(cust_city, 1, 3))
       AS user_login
FROM Customers;

-- MySQL
SELECT cust_id, cust_name,
       CONCAT(UPPER(LEFT(cust_contact, 2)),
       UPPER(LEFT(cust_city, 3)))
       AS user_login
FROM Customers;

-- SQL Server
SELECT cust_id, cust_name,
       UPPER(LEFT(cust_contact, 2)) +
       UPPER(LEFT(cust_city, 3))
       AS user_login
FROM Customers;
```

2.

```
-- DB2, MariaDB, MySQL
SELECT order_num, order_date
FROM Orders
WHERE YEAR(order_date) = 2020 AND
      MONTH(order_date) = 1
ORDER BY order_date;

-- Oracle, PostgreSQL
SELECT order_num, order_date
FROM Orders
WHERE EXTRACT(year FROM order_date) = 2020 AND
      EXTRACT(month FROM order_date) = 1
ORDER BY order_date;
```

```
-- PostgreSQL
SELECT order_num, order_date
FROM Orders
WHERE DATE_PART('year', order_date) = 2020
  AND DATE_PART('month', order_date) = 1
ORDER BY order_num;

-- SQL Server
SELECT order_num, order_date
FROM Orders
WHERE DATEPART(yy, order_date) = 2020 AND
      DATEPART(mm, order_date) = 1
ORDER BY order_date;

-- SQLite
SELECT order_num
FROM Orders
WHERE strftime('%Y', order_date) = '2020'
  AND strftime('%m', order_date) = '01';
```

Урок 9

1.

```
SELECT SUM(quantity) AS items_ordered
FROM OrderItems;
```

2.

```
SELECT SUM(quantity) AS items_ordered
FROM OrderItems
WHERE prod_id = 'BR01';
```

3.

```
SELECT MAX(prod_price) AS max_price
FROM Products
WHERE prod_price <= 10;
```

Урок 10

1.

```
SELECT order_num, COUNT(*) as order_lines
FROM OrderItems
GROUP BY order_num
ORDER BY order_lines;
```

2.

```
SELECT vend_id, MIN(prod_price) AS cheapest_item
FROM Products
GROUP BY vend_id
ORDER BY cheapest_item;
```

3.

```
SELECT order_num
FROM OrderItems
GROUP BY order_num
HAVING SUM(quantity) >= 100
ORDER BY order_num;
```

4.

```
SELECT order_num, SUM(item_price*quantity)
                  AS total_price
FROM OrderItems
GROUP BY order_num
HAVING SUM(item_price*quantity) >= 1000
ORDER BY order_num;
```

5.

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

Предложение `GROUP BY items` некорректно. В нем должно быть указано имя реального, а не вычисляемого столбца. Предложение `GROUP BY order_num` было бы допустимым.

Урок 11

1.

```
SELECT cust_id
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE item_price >= 10);
```

2.

```
SELECT cust_id, order_date
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE prod_id = 'BR01')
ORDER BY order_date;
```

3.

```
SELECT cust_email
FROM Customers
WHERE cust_id IN
      (SELECT cust_id
       FROM Orders
       WHERE order_num IN (SELECT order_num
                           FROM OrderItems
                           WHERE prod_id = 'BR01'));
```

4.

```
SELECT cust_id,
       (SELECT SUM(item_price*quantity)
        FROM OrderItems
        WHERE Orders.order_num = OrderItems.order_num)
       AS total_ordered
FROM Orders
ORDER BY total_ordered DESC;
```

5.

```
SELECT prod_name,
       (SELECT SUM(quantity)
        FROM OrderItems
        WHERE Products.prod_id = OrderItems.prod_id)
       AS quant_sold
FROM Products;
```

Урок 12

1.

```
-- Соединение по равенству
SELECT cust_name, order_num
FROM Customers, Orders
WHERE Customers.cust_id = Orders.cust_id
ORDER BY cust_name, order_num;

-- Синтаксис ANSI
SELECT cust_name, order_num
FROM Customers INNER JOIN Orders
ON Customers.cust_id = Orders.cust_id
ORDER BY cust_name, order_num;
```

2.

```
-- Решение с использованием подзапросов
SELECT cust_name,
       order_num,
       (SELECT Sum(item_price*quantity)
       FROM OrderItems
       WHERE Orders.order_num = OrderItems.order_num)
       AS OrderTotal
FROM Customers, Orders
WHERE Customers.cust_id = Orders.cust_id
ORDER BY cust_name, order_num;
```

```
-- Решение с использованием соединений
SELECT cust_name,
       Orders.order_num,
       Sum(item_price*quantity) AS OrderTotal
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
  AND Orders.order_num = OrderItems.order_num
GROUP BY cust_name, Orders.order_num
ORDER BY cust_name, order_num;
```

3.

```
SELECT cust_id, order_date
FROM Orders, OrderItems
WHERE Orders.order_num = OrderItems.order_num
  AND prod_id = 'BR01'
ORDER BY order_date;
```

4.

```
SELECT cust_email
FROM Customers
INNER JOIN Orders
        ON Customers.cust_id = Orders.cust_id
INNER JOIN OrderItems
        ON Orders.order_num = OrderItems.order_num
WHERE prod_id = 'BR01';
```

5.

```
-- Соединение по равенству
SELECT cust_name, SUM(item_price*quantity)
                  AS total_price
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
  AND Orders.order_num = OrderItems.order_num
GROUP BY cust_name
HAVING SUM(item_price*quantity) >= 1000
ORDER BY cust_name;

-- Синтаксис ANSI
SELECT cust_name, SUM(item_price*quantity)
                  AS total_price
```

```
FROM Customers
INNER JOIN Orders
        ON Customers.cust_id = Orders.cust_id
INNER JOIN OrderItems
        ON Orders.order_num = OrderItems.order_num
GROUP BY cust_name
HAVING SUM(item_price*quantity) >= 1000
ORDER BY cust_name;
```

Урок 13

1.

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

2.

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

3.

```
SELECT prod_name, order_num
FROM Products LEFT OUTER JOIN OrderItems
  ON Products.prod_id = OrderItems.prod_id
ORDER BY prod_name;
```

4.

```
SELECT prod_name, COUNT(order_num) AS orders
FROM Products LEFT OUTER JOIN OrderItems
  ON Products.prod_id = OrderItems.prod_id
GROUP BY prod_name
ORDER BY prod_name;
```

5.

```
SELECT Vendors.vend_id, COUNT(prod_id)
FROM Vendors
LEFT OUTER JOIN Products
            ON Vendors.vend_id = Products.vend_id
GROUP BY Vendors.vend_id;
```

Урок 14

1.

```
SELECT prod_id, quantity
FROM OrderItems
WHERE quantity = 100
UNION
SELECT prod_id, quantity
FROM OrderItems
WHERE prod_id LIKE 'BNBG%'
ORDER BY prod_id;
```

2.

```
SELECT prod_id, quantity
FROM OrderItems
WHERE quantity = 100 OR prod_id LIKE 'BNBG%'
ORDER BY prod_id;
```

3.

```
SELECT prod_name
FROM Products
UNION
SELECT cust_name
FROM Customers
ORDER BY prod_name;
```

4.

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state  = 'MI'
ORDER BY cust_name;
```

```
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state = 'IL'
ORDER BY cust_name;
```

После первой инструкции SELECT не должно быть точки с запятой, поскольку она считается концом запроса. Кроме того, в случае сортировки результатов запросов SELECT, объединяемых с помощью оператора UNION, допускается только одно предложение ORDER BY, которое должно стоять после завершающей инструкции SELECT.

Урок 15

1.

```
-- Подставьте свои данные
INSERT INTO Customers(cust_id,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country,
                      cust_email)
VALUES(1000000042,
       'Ben''s Toys',
       '123 Main Street',
       'Oak Park',
       'MI',
       '48237',
       'USA',
       'ben@forta.com');
```

2.

```
-- MySQL, MariaDB, Oracle, PostgreSQL, SQLite
CREATE TABLE OrdersBackup AS SELECT * FROM Orders;
CREATE TABLE OrderItemsBackup AS SELECT * FROM
OrderItems;

-- SQL Server
SELECT * INTO OrdersBackup FROM Orders;
SELECT * INTO OrderItemsBackup FROM OrderItems;
```

Урок 16

1.

```
UPDATE Vendors
SET vend_state = UPPER(vend_state)
WHERE vend_country = 'USA';
UPDATE Customers
SET cust_state = UPPER(cust_state)
WHERE cust_country = 'USA';
```

2.

```
-- Сначала протестируйте предложение WHERE,
-- чтобы оно отбирало только удаляемую запись
SELECT * FROM Customers
WHERE cust_id = 1000000042;
-- И лишь после этого выполняйте следующую инструкцию!
DELETE Customers
WHERE cust_id = 1000000042;
```

Урок 17

1.

```
ALTER TABLE Vendors
ADD vend_web CHAR(100);
```

2.

```
UPDATE Vendors
SET vend_web = 'https://google.com/'
WHERE vend_id = 'DLL01';
```

Урок 18

1.

```
CREATE VIEW CustomersWithOrders AS
SELECT Customers.cust_id,
       Customers.cust_name,
       Customers.cust_address,
       Customers.cust_city,
       Customers.cust_state,
       Customers.cust_zip,
       Customers.cust_country,
       Customers.cust_contact,
       Customers.cust_email
FROM Customers
JOIN Orders ON Customers.cust_id = Orders.cust_id;

SELECT * FROM CustomersWithOrders;
```

2.

```
CREATE VIEW OrderItemsExpanded AS
SELECT order_num,
       prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM OrderItems
ORDER BY order_num;
```

Предложение ORDER BY не допускается в представлениях, ведь это по сути обычные таблицы. Если требуется отсортировать данные, используйте предложение ORDER BY в инструкции SELECT, которая извлекает данные из самого представления.

Предметный указатель

N

NOT NULL 224
NULL 65; 224

O

ODBC 321

S

SOUNDEX 109
SQL 28

А

Арифметическая
операция 101

Б

База данных 22
Безопасность 296

В

Внешнее соединение 181
Внешний ключ 217; 285
Внутреннее соединение 167
Вычисляемое поле 93; 94; 246

Г

Группирование 133

Д

Дата 227; 320
Декартово произведение 165
Добавление
нескольких строк 210
полных строк 202
результатов запроса 207
части строки 206

Е

Естественное соединение 179

З

Запись 26
Запрос 147
комбинированный 189
Значение по умолчанию 226

И

Индекс 290
Инструкция
ALTER TABLE 228; 284; 309
BEGIN
TRANSACTION 267
CLOSE 280
COMMIT 268; 310
CREATE INDEX 293; 310
CREATE PROCEDURE 256;
310
CREATE SELECT 210
CREATE TABLE 221; 283;
311
CREATE TRIGGER 295
CREATE VIEW 240; 311
DECLARE 258; 275
DELETE 216; 312
DROP 312
DROP TABLE 231
DROP VIEW 240
EXECUTE 255
FETCH 277
GRANT 297
INSERT 201; 312
INSERT SELECT 207; 312
OPEN CURSOR 276

RENAME 232
REVOKE 297
ROLLBACK 268; 313
SAVEPOINT 270
SAVE TRANSACTION 270
SELECT 33; 313
порядок предложений 143
SELECT INTO 210
SET TRANSACTION 267
START TRANSACTION 267
TRUNCATE TABLE 218
UPDATE 213; 314
Итоговая функция 120; 184

К

Кавычки 63
Каскадное удаление 286
Ключ
внешний 217; 285
первичный 27; 160; 283
Ключевое слово 33; 323
ALL 128; 136
AND 69
AS 99; 176
ASC 57
BETWEEN 64
CONSTRAINT 284
DEFAULT 226
DESC 54
DISTINCT 40; 128
IN 75
INTO 203
LIKE 81
NOT 77
OR 71
REFERENCES 287
TOP 41
UNIQUE 288
Комбинированный запрос 189
Комментарий 45; 260
Конкатенация 95
Копирование 210
Критерий отбора 59
Курсор 273
закрытие 279
открытие 276
создание 275

М

Метасимвол 82
знак подчеркивания 86
знак процента 83
квадратные скобки 87

Н

Неявная фиксация 268

О

Обновление 213; 228
Ограничение 281
на значения столбца 288
уникальности 287
Оператор
EXCEPT 198
INTERSECT 198
UNION 190
UNION ALL 195
арифметический 103
сравнения 61
Откат транзакции 266

П

Первичный ключ 27; 160; 283
Переименование 232
Перекрестное соединение 167

Переносимый код 106
Подзапрос 147
  в качестве вычисляемого поля 152
Поле 94
Полное внешнее соединение 184
Пользовательский тип данных 290
Предикат 82
Предложение 50
  GROUP BY 134
  HAVING 137
  LIMIT 43
  ON 168
  ORDER BY 50; 141
  SET 214
  VALUES 202
  WHERE 59; 69
Представление 235
  создание 240
  удаление 240
Пробел 37
Псевдоним 99; 175
  имя 130

Р

Регистр символов 36; 56; 83
Реляционная таблица 159

С

Самосоединение 177
Символ подстановки 39
Скобки 74
Соединение 159
  внешнее 181
  внутреннее 167
  естественное 179
  нескольких таблиц 168
  перекрестное 167
  полное внешнее 184
  по равенству 167
Сортировка 49; 141; 196
  в указанном направлении 54
  по невыбранным столбцам 51; 54
  по нескольким столбцам 51
  по положению столбца 53
Ссылочная целостность 162; 217
Столбец 24; 94
Строка 26
СУБД 22
Схема 24

Т

Таблица 23
  копирование 211
  обновление 228
  переименование 232
  реляционная 159
  создание 221
  удаление 231
Тип данных 25; 315
  бинарный 321
  даты и времени 320
  денежный 319
  пользовательский 290
  строковый 316
  числовой 318
Точка сохранения 266; 270
Транзакция 263
  откат 268
Триггер 294

У

Удаление 216; 231
Условие фильтрации 59

Ф

Фиксация транзакции 266
  неявная 268
Фильтрация 59
  по группам 137
Функция 105
  AVG() 120
  COUNT() 122
  DATE_PART() 113
  DATEPART() 112
  LTRIM() 99
  MAX() 124
  MIN() 125
  RTRIM() 97
  SOUNDEX() 109
  SUM() 126
  to_date() 114
  TRIM() 99
  UPPER() 108
  YEAR() 114
  итоговая 120; 184
  строковая 108
  числовая 115

Х

Хранимая процедура 251
  создание 256

Ш

Шаблон поиска 82

5-е издание

SQL за 10 минут

Хорошее знание SQL требуется всем, кто работает с базами данных, включая разработчиков приложений, веб-дизайнеров, администраторов СУБД и даже пользователей Microsoft Office. В книге предлагаются готовые решения для тех, кто хочет быстро получить результат.

Эксперт по базам данных Бен Форта расскажет обо всем, что касается основ SQL: от простых запросов на выборку данных до более сложных тем, таких как соединения, подзапросы, хранимые процедуры, курсоры, триггеры и табличные ограничения.

Все темы последовательно излагаются в виде простых и коротких уроков, на каждый из которых уйдет не более 10 минут. Большинство уроков дополняется упражнениями, предназначенными для закрепления материала.

Что можно узнать за 10 минут:

- Основные инструкции SQL
- Создание сложных запросов с множеством предложений и операторов
- Извлечение, сортировка и форматирование данных
- Фильтрация результатов запроса
- Применение итоговых функций для получения сводных данных
- Соединение таблиц
- Добавление, обновление и удаление данных
- Создание и изменение таблиц
- Работа с представлениями, хранимыми процедурами и триггерами

Бен Форта — директор департамента обучающих решений в компании Adobe. Автор множества бестселлеров, включая книги по базам данных и регулярным выражениям.

Категория: компьютерные технологии/ базы данных

ISBN 978-5-907365-67-4

9 785907 365674 21087

SAMS
informit.com/sams

Pearson

Диалектика
www.williamspublishing.com