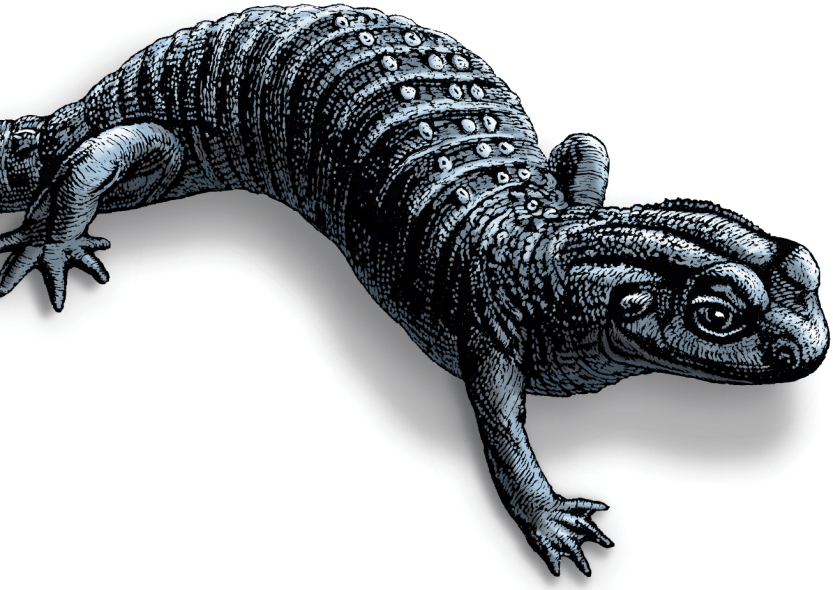
O'REILLY®

Pocket Guide
4-е издание

SQL

MySQL, Oracle, PostgreSQL,
SQL Server, SQLite

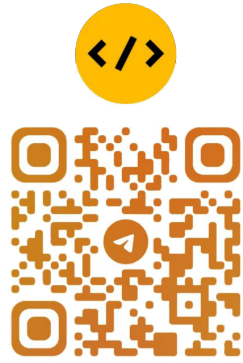


SPRINT book

Элис Жао

FOURTH EDITION

SQL Pocket Guide



Alice Zhao

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

SQL

MySQL, Oracle, PostgreSQL, SQL Server, SQLite

ЧЕТВЕРТОЕ ИЗАНИЕ

Элис Жао

SPRINT book 2024

ББК 32.973.2-018.1
УДК 004.43
Ж32

Жао Элис

Ж32 SQL. Pocket guide. 4-е изд. — Астана: «Спринт Бук», 2024. — 320 с.: ил.

ISBN 978-601-08-3728-7

Если вы аналитик или инженер по обработке данных и используете SQL, популярный карманный справочник станет для вас идеальным помощником. Найдите множество примеров, раскрывающих все сложности языка, а также ключевые аспекты SQL при его использовании в Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL и SQLite.

В обновленном издании Элис Жао описывает, как в этих СУБД используется SQL для формирования запросов и внесения изменений в базу. Получите подробную информацию о типах данных и их преобразованиях, синтаксисе регулярных выражений, оконных функциях, операторах PIVOT и UNPIVOT и многом другом.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1492090403 англ.

ISBN 978-601-08-3728-7

Краткое содержание

Оглавление

Предисловие

Почему именно SQL

С момента выхода последнего издания «SQL. Карманный справочник» в мире данных многое изменилось. Объем генерируемых и собираемых данных резко увеличился, и для обработки притока данных был создан целый ряд инструментов и рабочих мест. Несмотря на эти изменения, SQL оставался неотъемлемой частью ландшафта данных.

Последние 15 лет я работала инженером, консультантом, аналитиком и специалистом по исследованию данных и на каждой из этих должностей использовала SQL. Даже если мои основные обязанности подразумевали работу с другим инструментом, я должна была знать SQL, чтобы получить доступ к данным в компании.

> *Если бы существовала премия языка программирования за лучшую мужскую роль второго плана, то приз достался бы SQL.*

По мере появления новых технологий SQL по-прежнему остается в центре внимания, когда речь идет о работе с данными. Такие облачные хранилища, как Amazon Redshift и Google BigQuery, требуют от пользователей написания SQL-запросов для получения данных. Фреймворки для распределенной обработки данных, такие как Hadoop и Spark, имеют вспомогательные приложения Hive и Spark SQL соответственно, которые предоставляют пользователям SQL-подобные интерфейсы для анализа данных.

Язык SQL существует уже почти пять десятилетий и в ближайшее время не собирается сдавать позиции. Это один из старейших языков программирования, который широко используется и сегодня, и я рада поделиться с вами новейшими и лучшими достижениями.

Цели книги

Есть множество книг по SQL, начиная с тех, которые учат новичков программированию на этом языке, и заканчивая подробными техническими спецификациями для администраторов баз данных. Эта книга не предназначена для глубокого изучения всех концепций SQL и скорее является простым справочником на тот случай, если вы:

- забыли какой-то синтаксис SQL и вам нужно быстро найти его;
- на новом месте работы столкнулись с несколько иным набором инструментов для работы с базами данных и должны разобраться в нюансах различий;
- какое-то время занимались другим языком программирования и хотите быстро освежить знания о том, как работает SQL.

Если SQL играет важную вспомогательную роль в вашей работе, то эта книга — идеальный карманный справочник для вас.

Обновления в четвертом издании

Третье издание «SQL. Карманный справочник» Джонатана Генника вышло в 2010 году и было хорошо принято читателями. В четвертое издание я внесла следующие обновления.

- Синтаксис обновлен для Microsoft SQL Server, MySQL, Oracle Database и PostgreSQL. Информация о Db2 от IBM была удалена вследствие уменьшения ее популярности, а об SQLite — добавлена в связи с ростом ее востребованности.

- В третьем издании разделы были даны в алфавитном порядке. В четвертом я изменила их расположение так, чтобы схожие понятия были сгруппированы.
- В связи с тем что аналитики и специалисты по исследованию данных стали применять SQL в своей работе, я добавила разделы с информацией о том, как использовать этот язык с Python и R (популярные языки программирования с открытым исходным кодом), а также краткий курс по SQL для тех, кому нужно быстро освежить знания.

ЧАСТО ЗАДАВАЕМЫЕ ВОПРОСЫ ПО SQL

Последняя глава этой книги называется «Как мне?..» и содержит часто задаваемые вопросы начинающих пользователей SQL или тех, кто давно не использовал этот язык.

Это хорошая отправная точка, если вы не помните точное ключевое слово или концепцию, которую ищете. Примеры вопросов выглядят так.

- Как найти строки, содержащие дублирующиеся значения?
- Как выбрать строки с максимальным значением для другого столбца?
- Как объединить текст из нескольких полей в одно?

Структура издания

Книгу условно можно разделить на три части.

I. Основные понятия

- В главах 1–3 представлены основные ключевые слова, концепции и инструменты для написания SQL-кода.
- В главе 4 рассматривается каждое предложение SQL-запроса.

II. Объекты базы данных, типы данных и функции

- В главе 5 перечислены распространенные способы создания и модификации объектов в базе данных.
- В главе 6 перечислены типы данных, широко используемые в SQL.
- В главе 7 перечислены операции и функции, широко используемые в SQL.

III. Расширенные концепции

- В главах 8 и 9 раскрываются расширенные концепции построения запросов, в том числе описываются соединения, операторы `case`, оконные функции и т. д.
- В главе 10 рассматриваются решения некоторых часто встречающихся вопросов по SQL.

Условные обозначения

В данной книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и понятия, на которых нужно акцентировать внимание.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширений.

Жирный моноширинный шрифт

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.

Этот элемент указывает на совет или предложение.

Этот элемент указывает на общее примечание.

Этот элемент указывает на предупреждение.

Использование примеров кода

Эта книга призвана помочь вам выполнять свою работу. В общем случае все примеры кода вы можете использовать в своих программах и документации. Вам не нужно обращаться в издательство O'Reilly за разрешением, если вы не собираетесь воспроизводить существенные части кода, или если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, или если вы будете цитировать эту книгу или примеры из нее, отвечая на вопросы заинтересованных лиц. Вам потребуется разрешение издательства O'Reilly, если вы хотите продавать или распространять примеры из книги либо добавить существенные объемы представленного в ней программного кода в документацию вашего продукта.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Спасибо Джонатану Геннику за создание этого карманного справочника с нуля и написание первых трех книг, а также Энди Квану за то, что доверил мне издание четвертой.

Я не смогла бы завершить работу над этой книгой без помощи моих редакторов Амелии Блевинс, Джеффа Блейела и Кейтлин Геган, а также моих технических рецензентов Алисии Невелс, Джоан Ванг, Скотта Хейнса и Томаса Нилда. Я искренне ценю время, которое вы потратили на прочтение каждой страницы этой книги. Ваши отзывы были бесценны.

Моим родителям — спасибо за то, что привили мне любовь к учебе и творчеству. Моим детям Генри и Лили — ваш восторг по поводу этой книги согревает мое сердце. И наконец, моему мужу Али — спасибо за все замечания по этой книге, за твою поддержку и за то, что ты мой самый большой поклонник.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

ГЛАВА 1

Ускоренный курс SQL

Эта небольшая глава предназначена для быстрого ознакомления с базовой терминологией и концепциями SQL.

Что такое база данных

Начнем с основ. *База данных (БД)* — это место для организованного хранения данных. Есть множество способов организации данных и, как следствие, большое количество баз, из которых можно выбирать. Базы данных делятся на две категории: *SQL* и *NoSQL*.

SQL

SQL — это сокращение от *Structured Query Language* (язык структурированных запросов). Представьте, что у вас есть приложение, которое помнит дни рождения ваших друзей и близких. SQL — самый популярный язык, с помощью которого вы можете взаимодействовать с этим приложением.

Русский язык: Привет, приложение. Когда у моего мужа день рождения?

SQL:
```
SELECT * FROM birthdays
WHERE person = 'husband';
```

Базы данных SQL часто называют *реляционными*, поскольку они состоят из отношений, которые чаще всего называют таблицами. Множество таблиц, связанных между собой, составляют БД. На рис. 1.1 изображена таблица в базе данных SQL.

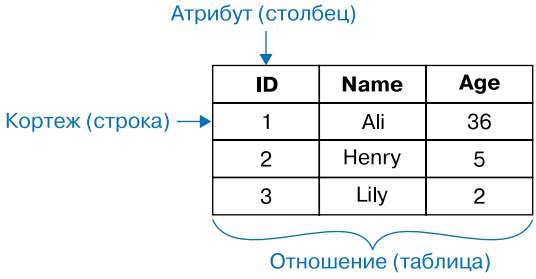


Рис. 1.1. Таблица в базе данных SQL

Главное, что следует отметить в отношении баз данных SQL, — это то, что они требуют наличия предопределенных *схем*. Схему можно рассматривать как способ организации или структурирования данных в базе. Допустим, вы хотите создать таблицу. Прежде чем загружать в нее какие-либо данные, необходимо сначала определиться с ее структурой, в том числе с тем, какие столбцы будут в ней, будут ли они содержать целочисленные или десятичные значения и т. д.

Однако наступает момент, когда данные невозможно организовать таким структурированным образом. Ваши данные могут иметь различные поля, или вам может потребоваться более эффективный способ хранения большого объема данных и доступа к ним. Вот здесь и приходит на помощь NoSQL.

NoSQL

NoSQL означает *не только SQL* (not only SQL). В нашей книге эта категория баз подробно не рассматривается, но я хотела обратить на нее внимание, поскольку с 2010-х этот термин стал весьма популярным и важно понимать, что существуют способы хранения данных, выходящие за рамки таблиц.

Базы данных NoSQL часто называются *нереляционными* и бывают разных форм и размеров. В числе их основных характеристик — наличие динамических схем (то есть схема не должна быть зафиксирована заранее) и возможность горизонтального масштабирования (то есть данные могут быть распределены по нескольким машинам).

Наиболее популярной базой данных NoSQL является *MongoDB*, которая в большей степени представляет собой БД документов. На рис. 1.2 показано, как хранятся данные в MongoDB. Вы можете заметить, что данные больше не находятся в структурированной таблице, а количество полей (аналогично столбцам) варьируется для каждого документа (аналогично строкам).

Рис. 1.2. Коллекция (вариант таблицы) в MongoDB, базе данных NoSQL

Тем не менее основное внимание в этой книге уделяется базам данных SQL. Даже после появления NoSQL очень многие компании по-прежнему хранят бóльшую часть своих данных в таблицах реляционных БД.

Системы управления базами данных (СУБД)

Возможно, вы слышали такие термины, как PostgreSQL или SQLite, и задавались вопросом, чем они отличаются от SQL. Это два типа *систем управления базами данных (СУБД)*, то есть программного обеспечения, используемого для работы с базой.

Программное обеспечения СУБД включает в себя такие функции, как определение способов импорта и организации данных, а также управление доступом пользователей и других программ к данным. *Система управления реляционными базами данных (РСУБД)* — программное обеспечение, специально предназначенное для реляционных БД, или баз, состоящих из таблиц.

Каждая РСУБД имеет свою реализацию SQL, а это означает, что синтаксис немного различается в зависимости от программного обеспечения. Например, вот как можно вывести десять строк данных в пяти различных системах:

MySQL, *PostgreSQL* и *SQLite*

```
SELECT * FROM birthdays LIMIT 10;
```

Microsoft SQL Server

```
SELECT TOP 10 * FROM birthdays;
```

Oracle Database

```
SELECT * FROM birthdays WHERE ROWNUM <= 10;
```

ПОИСК СИНТАКСИСА SQL В GOOGLE

При поиске синтаксиса SQL в Интернете всегда указывайте в поиске РСУБД, с которой работаете. На заре моего изучения SQL я не могла понять, почему скопированный мною из Интернета код не работает, а причина была именно в этом!

Сделайте так.

Поиск: *создать таблицу datetime* ***postgresql***

→ Результат: `timestamp`

Поиск: *создать таблицу datetime* ***microsoft sql server***

→ Результат: `datetime`

А не так.

Поиск: *создать таблицу datetime*

→ Результат: синтаксис может быть для любой РСУБД

В этой книге рассматриваются основы SQL, а также нюансы пяти популярных систем управления базами данных: Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL и SQLite.

Одни из этих систем являются проприетарными, то есть принадлежат компании, и их использование требует денег, а другие имеют открытый исходный код, то есть любой желающий может использовать их бесплатно. В табл. 1.1 подробно описаны различия между РСУБД.

Таблица 1.1. Таблица сравнения РСУБД

РСУБД	Владелец	Основные сведения
Microsoft SQL Server	Microsoft	• Популярная проприетарная РСУБД. • Часто используется вместе с другими продуктами Microsoft, в том числе с Microsoft Azure и фреймворком .NET. • Наиболее часто используется на платформе Windows. • Иное название — MSSQL или SQL Server
MySQL	Открытый исходный код	• Популярная РСУБД с открытым исходным кодом. • Часто используется вместе с такими языками веб-разработки, как HTML/CSS/Javascript. • Приобретена компанией Oracle, хотя по-прежнему имеет открытый исходный код
Oracle Database	Oracle	• Популярная проприетарная РСУБД. • Часто используется в крупных корпорациях благодаря большому количеству доступных функций, инструментов и поддержки. • Иное название — просто Oracle
PostgreSQL	Открытый исходный код	• Популярность быстро растет. • Часто используется вместе с технологиями с открытым исходным кодом, такими как Docker и Kubernetes. • Эффективна и отлично подходит для больших наборов данных

Продолжение ⇩

Таблица 1.1 *(продолжение)*

РСУБД	Владелец	Основные сведения
SQLite	Открытый исходный код	• Самый используемый в мире движок баз данных. • Распространен на платформах iOS и Android. • Легкий и отлично подходит для небольшой БД

Далее в книге:

- Microsoft SQL Server будет называться SQL Server;
- Oracle Database будет называться Oracle.

Инструкции по установке и фрагменты кода для каждой системы приведены в разделе «Программное обеспечение РСУБД» главы 2.

SQL-запрос

В мире SQL часто встречается аббревиатура *CRUD*, которая расшифровывается как Create, Read, Update, Delete (создание, чтение, обновление и удаление). Это четыре основные операции, которые доступны в базе данных.

SQL-операторы

Люди, имеющие *доступ для чтения и записи*, могут выполнять с базой все четыре операции: создавать и удалять таблицы, обновлять в них данные и читать данные из таблиц. Другими словами, эти люди обладают всей полнотой власти.

Они пишут *SQL-операторы* — общий SQL-код, который можно написать для выполнения любой из операций CRUD. Эти люди часто занимают такие должности, как *администратор баз данных* (database administrator, DBA) или *инженер баз данных*.

SQL-запросы

Люди, имеющие *доступ для чтения*, могут выполнять в БД только операцию чтения, то есть могут просматривать данные в таблицах.

Эти люди пишут *SQL-запросы*, которые представляют собой более конкретный тип SQL-операторов. Запросы используются для поиска и отображения данных, иначе называемых *чтением* данных. Иногда это действие называют *запросом к таблицам*. Эти люди часто занимают такие должности, как *аналитик данных* или *специалист по анализу данных*.

Следующие два подраздела представляют собой краткое руководство по написанию SQL-запросов, поскольку это наиболее часто встречающийся тип SQL-кода. Более подробная информация о создании и обновлении таблиц содержится в главе 5.

Оператор SELECT

Самый простой SQL-запрос (который будет работать в любом программном обеспечении SQL) имеет вид:

```
SELECT * FROM my_table;
```

В нем говорится: покажите мне все данные в таблице с именем `my_table` — все столбцы и все строки.

Хотя SQL не чувствителен к регистру (`SELECT` и `select` эквивалентны), можно заметить, что одни слова написаны заглавными буквами, а другие — нет.

- Слова в запросе, записанные заглавными буквами, называются *ключевыми*. Это означает, что SQL зарезервировал их для выполнения той или иной операции над данными.
- Все остальные слова пишутся строчными буквами. Это относится к именам таблиц, столбцов и т. д.

Формат заглавных и строчных букв не является обязательным, но это хорошее условное обозначение, которое следует соблюдать ради удобства чтения. Вернемся к уже знакомому нам запросу:

```
SELECT * FROM my_table;
```

Допустим, вместо того, чтобы возвращать все данные в их текущем состоянии, я хочу их:

- фильтровать;
- сортировать.

Для этого необходимо изменить оператор `SELECT`, добавив в него несколько *предложений (clauses)*, и результат будет выглядеть примерно так:

```
SELECT *
FROM my_table
WHERE column1 > 100
ORDER BY column2;
```

Более подробную информацию обо всех предложениях можно найти в главе 4, но главное, что следует отметить: предложения всегда должны быть перечислены в одном и том же порядке.

ЗАПОМНИТЕ ЭТОТ ПОРЯДОК

Все SQL-запросы будут содержать некую комбинацию этих предложений. Обязательно запомните этот порядок!

```
SELECT          -- столбцы для отображения
FROM            -- таблица (-ы), из которой (-ых) нужно
                   извлечь данные
WHERE           -- фильтрация строк
GROUP BY        -- разбиение строк на группы
HAVING          -- фильтрация сгруппированных строк
ORDER BY        -- столбцы для сортировки
```

Знак -- представляет собой начало комментария в SQL, то есть текст после него предназначен только для документации и код выполняться не будет.

По большей части, из всех этих предложений обязательные лишь SELECT и FROM. Исключением является случай, когда вы выбираете определенную функцию базы данных, — здесь вам требуется только SELECT.

Классическим мнемоническим выражением для запоминания порядка расположения предложений является:

Sweaty feet will give horrible odors.

Если не хотите думать о потных ногах каждый раз, когда пишете запрос, то вот один из вариантов, который я придумала:

Start Fridays with grandma's homemade oatmeal.

Порядок выполнения

Порядок выполнения SQL-кода обычно не изучается в курсах SQL для начинающих, но я добавила данную тему в книгу, поскольку часто получала этот вопрос, когда преподавала SQL студентам, имеющим опыт программирования на Python.

Разумным было бы предположить, что порядок *написания* предложений соответствует порядку их *выполнения* компьютером, но это не так. После выполнения запроса компьютер обрабатывает данные в следующем порядке:

1) `FROM`;
2) `WHERE`;
3) `GROUP BY`;
4) `HAVING`;
5) `SELECT`;
6) `ORDER BY`.

Если сравнивать с порядком, в котором вы фактически записываете предложения, то можно заметить, что оператор `SELECT` переместился на пятую позицию. Главный вывод, который можно сделать, заключается в том, что SQL работает в следующем порядке:

1) собирает все данные с помощью `FROM`;
2) фильтрует строки данных, используя `WHERE`;
3) группирует строки с помощью `GROUP BY`;
4) фильтрует сгруппированные строки, используя `HAVING`;
5) указывает столбцы для отображения с помощью `SELECT`;
6) переупорядочивает результаты, используя `ORDER BY`.

Модель данных

В заключительном разделе ускоренного курса я хотела бы рассмотреть простую *модель данных* и обратить ваше внимание на некоторые термины, которые вы часто будете слышать, обсуждая SQL в офисе.

Модель данных — это визуализация, которая обобщает информацию о том, как все таблицы в БД связаны друг с другом, а также некоторые сведения о каждой из них. На рис. 1.3 представлена простая модель данных БД об оценках учащихся.

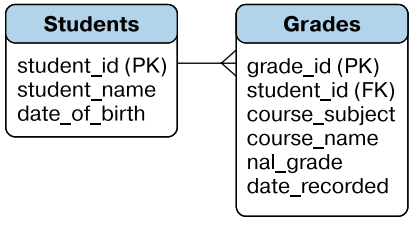


Рис. 1.3. Модель данных об оценках учащихся

В табл. 1.2 перечислены технические термины, описывающие происходящее в модели данных.

Таблица 1.2. Термины, используемые для описания того, что происходит в модели данных

Термин	Определение	Пример
База данных	База данных — место для организованного хранения данных	Эта модель данных отображает все данные в БД об оценках учащихся
Таблица	Таблица состоит из строк и столбцов. В модели данных они представлены прямоугольниками	В базе данных об оценках учащихся имеется две таблицы: `Students` и `Grades`
Колонка	Таблица состоит из множества столбцов, которые иногда называют атрибутами или полями. Каждый столбец содержит определенный тип данных. В модели данных все столбцы таблицы перечислены в каждом прямоугольнике	В таблице `Students` столбцами является `student_id`, `student_name` и `date_of_birth`
Первичный ключ	*Первичный ключ* однозначно идентифицирует каждую строку данных в таблице. Может состоять из одного или нескольких столбцов таблицы. В модели данных обозначается как `pk` (от primary key) или значком ключа	В таблице `Students` в поле первичным ключом является столбец `student_id`. Это говорит о том, что значение `student_id` для каждой строки данных различно
Внешний ключ	*Внешний ключ* в таблице ссылается на первичный ключ в другой таблице. Две таблицы могут быть связаны между собой общим столбцом. Таблица может иметь несколько внешних ключей. В модели данных внешний ключ обозначается как `fk` (от foreign key)	В таблице `Grades` столбец `student_id` является внешним ключом, то есть значения в этом столбце совпадают со значениями в соответствующем столбце первичного ключа в таблице `Students`

Продолжение ⇗

Таблица 1.2 *(продолжение)*

Термин	Определение	Пример
Отношения	*Отношение* описывает, как строки одной таблицы соотносятся со строками другой. В модели данных представлено в виде линии с символами в конечных точках. Распространенными типами являются отношения «один к одному» и «один ко многим»	В этой модели данных две таблицы имеют отношение «один ко многим», представленное разветвлением. Один студент может иметь много оценок, или одна строка таблицы `Students` сопоставляется с несколькими строками таблицы `Grades`

Более подробную информацию об этих терминах можно найти в разделе «Создание таблиц» главы 5.

Возможно, вам интересно, почему мы тратим столько времени на чтение модели данных вместо того, чтобы уже писать код SQL! Причина в том, что часто приходится писать запросы, связывающие несколько таблиц, поэтому рекомендуется сначала ознакомиться с моделью данных, чтобы знать, как эти таблицы связаны между собой.

Модели данных обычно можно найти в репозитории документации компании. Возможно, вы захотите распечатать модели данных, с которыми часто работаете, — это может облегчить удобство их использования.

Вы также можете писать запросы в РСУБД для поиска информации, содержащейся в модели данных, например, таблиц в БД, столбцов или ограничений таблицы.

ВОТ И ВЕСЬ УСКОРЕННЫЙ КУРС!

Оставшаяся часть книги предназначена для использования в качестве справочного материала, и ее необязательно читать по порядку. Вы можете искать в ней понятия, ключевые слова и стандарты.

ГЛАВА 2

Где можно писать SQL-код

В этой главе рассматриваются три места, где можно писать SQL-код.

- *Программное обеспечение РСУБД*. Чтобы написать код SQL, сначала необходимо скачать РСУБД типа MySQL, Oracle, PostgreSQL, SQL Server или SQLite. Нюансы каждой системы представлены в разделе «Программное обеспечение РСУБД» данной главы.
- *Инструменты для работы с базами данных*. После того как вы скачали систему, вы можете прибегнуть к наиболее простому способу написания SQL-кода — использовать *окно терминала*, представляющее собой черно-белый экран, на котором отображается только текст. Большинство людей предпочитают использовать вместо этого *инструмент для работы с базами данных* — более удобное приложение, незаметно подключающееся к РСУБД.

  Этот инструмент будет иметь *графический интерфейс пользователя* (graphical user interface, GUI), который позволяет пользователям визуально исследовать таблицы и упростить редактирование кода SQL. В разделе «Инструменты для работы с базами данных» текущей главы описано, как подключить инструмент к РСУБД.
- *Другие языки программирования*. SQL-код можно написать в рамках многих других языков программирования. В текущей главе основное внимание уделяется двум из них: Python

и R. Это популярные языки программирования с открытым исходным кодом, используемые учеными и аналитиками данных, которым часто приходится писать и SQL-код.

Чтобы не переключаться между Python/R и РСУБД, можно подключить Python/R непосредственно к РСУБД и писать SQL-код в Python/R. В разделе «Другие языки программирования» текущей главы объясняется, как это сделать.

Программное обеспечение РСУБД

Этот раздел содержит инструкции по установке и краткие фрагменты кода для пяти РСУБД, рассматриваемых в данной книге.

Какую РСУБД выбрать

Если вы работаете в компании, которая уже использует некую РСУБД, то с этой системой придется взаимодействовать и вам.

Если вы работаете над персональным проектом, то должны будете решить, какую РСУБД использовать. Вы можете вернуться к табл. 1.1, чтобы получить более подробную информацию о некоторых популярных системах.

БЫСТРЫЙ СТАРТ С SQLITE

Хотите начать писать SQL-код как можно быстрее? SQLite — самая быстрая в настройке РСУБД.

По сравнению с другими системами, о которых пойдет речь в этой книге, SQLite менее безопасна и не может работать с несколькими пользователями, но обеспечивает базовую функциональность языка SQL, реализованную в компактном пакете.

По этой причине я переместила SQLite в начало каждого раздела данной главы, поскольку настраивать эту систему в целом проще, чем другие.

Что такое окно терминала

В этой главе я буду часто ссылаться на *окно терминала*, поскольку после скачивания РСУБД оно является основным способом взаимодействия с ней.

Окно терминала — это приложение на вашем компьютере, которое обычно имеет черный фон и позволяет вводить только текст. Название приложения зависит от операционной системы:

- в Windows используется приложение Command Prompt;
- в macOS и Linux используется приложение Terminal.

Открыв окно терминала, вы увидите *командную строку*, она выглядит как символ >, за которым следует мигающее поле. Это означает, что терминал готов к приему текстовых команд от пользователя.

В следующих подразделах приведены ссылки, по которым можно скачать инсталляторы РСУБД для Windows, macOS и Linux.

В macOS и Linux альтернативой скачиванию инсталлятора является использование менеджера пакетов Homebrew (https://brew.sh). После установки Homebrew можно выполнять простые команды brew install из терминала, чтобы выполнить все установки РСУБД.

SQLite

SQLite — бесплатная система, и ее установочные файлы наиболее легковесные. Это значит, что она не занимает много места на компьютере и ее можно очень быстро установить. Для Windows и Linux инструменты SQLite можно скачать со

страницы SQLite Download Page (https://oreil.ly/gNagl); macOS поставляется с уже установленной SQLite.

Самый простой способ начать использовать SQLite — открыть окно терминала и ввести sqlite3. Однако при таком подходе все действия выполняются в памяти, а значит, изменения не будут сохранены после закрытия SQLite.

```
> sqlite3
```

Если вы хотите, чтобы ваши изменения были сохранены, то при открытии следует подключиться к базе данных, используя следующий синтаксис:

```
> sqlite3 my_new_db.db
```

Командная строка для SQLite выглядит так:

```
sqlite>
```

А это небольшой код для ее быстрого тестирования:

```
sqlite> CREATE TABLE test (id int, num int);
sqlite> INSERT INTO test VALUES (1, 100), (2, 200);
sqlite> SELECT * FROM test LIMIT 1;

1|100
```

Чтобы показать базы данных, таблицы и выйти, введите этот код:

```
sqlite> .databases
sqlite> .tables
sqlite> .quit
```

Если вы хотите отобразить имена столбцов в выходных данных, введите:

```
sqlite> .headers on
```

Чтобы снова скрыть их, введите:

```
sqlite> .headers off
```

MySQL

MySQL является бесплатной, несмотря на то что в настоящее время принадлежит компании Oracle. MySQL Community Server можно скачать со страницы MySQL Community Downloads (https://oreil.ly/Bkv0m). Как вариант, в macOS и Linux установку можно выполнить с помощью Homebrew, введя в терминале команду **`brew install mysql`**.

Командная строка для MySQL выглядит так:

```
mysql>
```

А это небольшой код для ее быстрого тестирования:

```
mysql> CREATE TABLE test (id int, num int);
mysql> INSERT INTO test VALUES (1, 100), (2, 200);
mysql> SELECT * FROM test LIMIT 1;

+------+------+
| id   | num  |
+------+------+
|    1 |  100 |
+------+------+
1 row in set (0.00 sec)
```

Чтобы отобразить базы данных, переключить их, отобразить таблицы и выйти, введите этот код:

```
mysql> show databases;
mysql> connect another_db;
mysql> show tables;
mysql> quit
```

Oracle

Oracle является проприетарной системой и работает на машинах под управлением ОС Windows и Linux. Бесплатную версию Oracle Database Express Edition можно скачать со страницы Oracle Database XE Downloads (https://oreil.ly/FGoXw).

Командная строка для Oracle выглядит так:

```
SQL>
```

А это небольшой код для ее быстрого тестирования:

```
SQL> CREATE TABLE test (id int, num int);
SQL> INSERT INTO test VALUES (1, 100);
SQL> INSERT INTO test VALUES (2, 200);
SQL> SELECT * FROM test WHERE ROWNUM <=1;

        ID         NUM
----------  ----------
         1         100
```

Чтобы отобразить базы данных, показать все таблицы (включая системные), отобразить таблицы, созданные пользователем, и выйти, введите этот код:

```
SQL> SELECT * FROM global_name;
SQL> SELECT table_name FROM all_tables;
SQL> SELECT table_name FROM user_tables;
SQL> quit
```

PostgreSQL

PostgreSQL является бесплатной и часто используется наряду с другими технологиями с открытым исходным кодом. Эту систему можно скачать со страницы PostgreSQL Downloads (https://oreil.ly/8MyzC). В качестве альтернативы в macOS и Linux установку можно выполнить с помощью Homebrew, набрав в терминале команду **`brew install postgresql`**.

Командная строка для PostgreSQL выглядит так:

```
postgres=#
```

А это небольшой код для ее быстрого тестирования:

```
postgres=# CREATE TABLE test (id int, num int);
postgres=# INSERT INTO test VALUES (1, 100),
  (2, 200);
postgres=# SELECT * FROM test LIMIT 1;

 id | num
----+-----
  1 | 100
(1 row)
```

Чтобы отобразить базы данных, переключить их, отобразить таблицы и выйти, введите этот код:

```
postgres=# \l
postgres=# \c another_db
postgres=# \d
postgres=# \q
```

Если вы когда-нибудь увидите postgres-#, это означает, что вы забыли поставить точку с запятой в конце оператора SQL. Введите этот символ — и снова увидите postgres=#.

Если вы когда-нибудь увидите двоеточие, это означает, что вы автоматически переключились на текстовый редактор vi, и выйти из него можно, введя q.

SQL Server

SQL Server является проприетарной системой (принадлежит Microsoft) и работает на компьютерах под управлением Windows и Linux. Она также может быть установлена с помощью Docker. Бесплатную версию SQL Server Express можно скачать со страницы Microsoft SQL Server Downloads (https://oreil.ly/zAxh9).

Командная строка для SQL Server выглядит так:

```
1>
```

А это небольшой код для ее быстрого тестирования:

```
1> CREATE TABLE test (id int, num int);
2> INSERT INTO test VALUES (1, 100), (2, 200);
3> go
1> SELECT TOP 1 * FROM test;
2> go

id          num
----------  ----------
         1         100
(1 row affected)
```

Чтобы отобразить базы данных, переключить их, отобразить таблицы и выйти, введите этот код:

```
1> SELECT name FROM master.sys.databases;
2> go
1> USE another_db;
2> go
1> SELECT * FROM information_schema.tables;
2> go
1> quit
```

В SQL Server код SQL не выполняется до тех пор, пока вы не введете команду go на новой строке.

Инструменты для работы с базами данных

Вместо того чтобы работать с РСУБД напрямую, большинство людей будут использовать для взаимодействия с базой инструментальные средства. Инструмент для работы с БД (далее — инструмент) имеет приятный графический интерфейс, который позволяет наводить курсор, щелкать кнопкой мыши и писать SQL-код в удобной для пользователя обстановке.

«За кулисами» инструмент использует *драйвер базы данных*, который представляет собой программное обеспечение, помогающее инструменту работать с БД. На рис. 2.1 показаны визуальные различия между прямым доступом к базе через окно терминала и косвенным через инструмент.

Есть целый ряд инструментов для работы с базами данных. Одни из них работают только с одной РСУБД, другие — с несколькими. В табл. 2.1 перечислены все системы, а также один из наиболее популярных инструментов для конкретной РСУБД. Все эти инструменты можно бесплатно скачать и использовать; кроме того, существует множество других проприетарных средств.

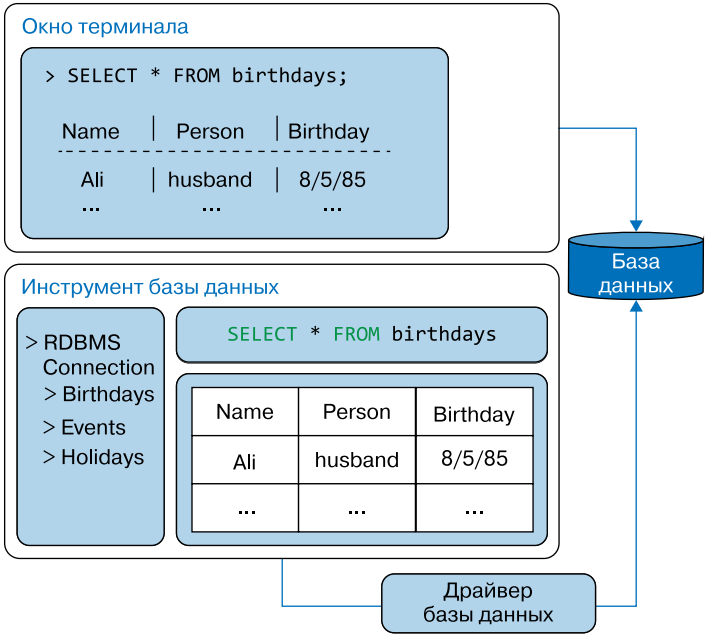


Рис. 2.1. Доступ к РСУБД через окно терминала в сравнении с доступом через инструмент

Таблица 2.1. Таблица сравнения инструментов для работы с базами данных

РСУБД	Инструмент	Подробная информация
SQLite	DB Browser для SQLite	• Сторонний разработчик. • Один из многих вариантов инструментов для SQLite
MySQL	MySQL Workbench	• Тот же разработчик, что и в MySQL
Oracle	Oracle SQL Developer	• Разработан компанией Oracle
PostgreSQL	pgAdmin	• Сторонний разработчик. • Входит в состав инсталляции PostgreSQL

Продолжение ⇩

Таблица 2.1 *(продолжение)*

РСУБД	Инструмент	Подробная информация
SQL Server	SQL Server Management Studio	• Разработан компанией Microsoft
Различные	DBeaver	• Один из многих вариантов инструментальных средств для подключения к различным РСУБД (включая любую из пяти, перечисленных выше)

Подключение инструмента к базе данных

При открытии инструмента первым делом следует подключиться к базе. Это можно сделать несколькими способами.

- *Вариант 1: создать новую базу данных*. Вы можете создать совершенно новую базу данных, написав оператор `CREATE`:

  ```
  CREATE DATABASE my_new_db;
  ```

  После этого можно создавать таблицы для наполнения базы. Более подробную информацию можно найти в разделе «Создание таблиц» главы 5.

- *Вариант 2: открыть файл базы данных*. Возможно, вы скачали или получили файл с расширением `.db`:

  ```
  my_new_db.db
  ```

  Этот файл `.db` уже будет содержать несколько таблиц. Его можно просто открыть в инструменте и начать взаимодействовать с БД.

- *Вариант 3: подключиться к существующей базе*. Возможно, вы захотите работать с базой, которая находится либо на вашем компьютере, либо на *удаленном сервере*, то есть данные находятся на компьютере, расположенном в другом месте. В наши дни это чрезвычайно распространено в сфере *облачных вычислений*, когда люди используют серверы, принадлежащие таким компаниям, как Amazon, Google или Microsoft.

ПОЛЯ ПОДКЛЮЧЕНИЯ К БАЗЕ ДАННЫХ

Для подключения к базе необходимо заполнить следующие поля в инструменте.

- *Host.* Место расположения базы.
  - Если она находится на вашем компьютере, то это должен быть `localhost` или `127.0.0.1`.
  - Если база данных находится на удаленном сервере, то это должен быть IP-адрес этого компьютера: `123.45.678.90`.
- *Port.* Как подключиться к РСУБД.

  В этом поле уже должен быть номер порта по умолчанию, и менять его не следует. Для каждой РСУБД он будет свой.
  - MySQL: `3306`.
  - Oracle: `1521`.
  - PostgreSQL: `5432`.
  - SQL Server: `1433`.
- *База данных.* Имя базы, к которой вы хотите подключиться.
- *Имя пользователя.* Ваше имя пользователя для базы.

  Возможно, в этом поле уже есть имя пользователя по умолчанию. Если вы не помните, как задавали имя, то сохраните значение по умолчанию.
- *Пароль.* Ваш пароль, связанный с именем пользователя.

  Если вы не помните, как задавали пароль для своего имени пользователя, то попробуйте оставить это поле пустым.

Если вы используете SQLite, то вместо того, чтобы заполнять эти пять полей, введите путь к файлу .db базы, к которой пытаетесь подключиться.

После правильного заполнения этих полей вы должны получить доступ к базе. Теперь вы можете использовать инструмент для поиска интересующих вас таблиц и полей и приступить к написанию SQL-кода.

Другие языки программирования

Как я уже говорила выше, SQL-код можно написать в рамках ряда других языков программирования. В данной главе основное внимание уделено двум популярным языкам с открытым исходным кодом: Python и R.

Как специалист по изучению данных или аналитик данных, вы, скорее всего, выполняете анализ на языке Python или R, а также вам необходимо писать SQL-запросы для получения данных из базы.

БАЗОВЫЙ РАБОЧИЙ ПРОЦЕСС АНАЛИЗА ДАННЫХ

1. Напишите SQL-запрос в программе для работы с базами данных.
2. Экспортируйте результаты в файл формата `.csv`.
3. Импортируйте файл `.csv` в Python или R.
4. Продолжайте проводить анализ в Python или R.

Такой подход хорош для быстрого однократного экспорта. Однако если необходимо постоянно редактировать SQL-запрос или работать с несколькими запросами, то этот вариант может быстро надоесть.

УЛУЧШЕННЫЙ РАБОЧИЙ ПРОЦЕСС АНАЛИЗА ДАННЫХ

1. Подключите к базе данных Python или R.
2. Напишите SQL-запросы на Python или R.
3. Продолжайте проводить анализ на Python или R.

Второй подход позволяет выполнять все запросы и анализ в одном инструменте и удобен в случае, если в процессе анали-

за возникает необходимость корректировки запросов. В оставшейся части главы приводится код для каждого этапа этого второго рабочего процесса.

Подключение Python к базе данных

Подключение Python к базе данных осуществляется в три этапа:

1) установить драйвер базы данных для Python;
2) настроить подключение к базе в Python;
3) написать SQL-код на языке Python.

Этап 1. Установка драйвера базы данных для Python

Драйвер базы данных — это программное обеспечение, которое помогает Python взаимодействовать с базой. Есть множество вариантов драйверов на выбор. В табл. 2.2 приведен код установки популярного драйвера для каждой РСУБД.

Эту установку необходимо выполнить один раз. Используйте одну из команд: либо **`pip install`**, либо **`conda install`**. Приведенный в таблице код должен быть запущен в окне терминала.

Таблица 2.2. Установка драйвера для Python с помощью pip или conda

РСУБД	Вариант	Код
SQLite	н/д	Установка не требуется (Python 3 поставляется с sqlite3)
MySQL	pip	`pip install mysql-connector-python`
	conda	`conda install -c conda-forge mysql-connector-python`
Oracle	pip	`pip install cx_Oracle`
	conda	`conda install -c conda-forge cx_oracle`

Продолжение ⇩

Таблица 2.2 *(продолжение)*

РСУБД	Вариант	Код
PostgreSQL	pip	`pip install psycopg2`
	conda	`conda install -c conda-forge psycopg2`
SQL Server	pip	`pip install pyodbc`
	conda	`conda install -c conda-forge pyodbc`

Этап 2. Настройка подключения к базе данных в Python

Чтобы настроить соединение, необходимо знать местоположение и имя базы, к которой вы пытаетесь подключиться, а также имя пользователя и пароль. Более подробную информацию можно найти во врезке «Поля подключения к базе данных» выше в текущей главе.

В табл. 2.3 приведен код Python, который необходимо запускать каждый раз, когда вы планируете писать SQL-код на Python. Вы можете добавить его в начало вашего сценария Python.

Таблица 2.3. Код на языке Python для настройки подключения к базе данных

РСУБД	Код
SQLite	`import sqlite3` `conn = sqlite3.connect('my_new_db.db')`

MySQL	`import mysql.connector`
	`conn = mysql.connector.connect(`
	`host='localhost',`
	`database='my_new_db',`
	`user='alice',`
	`password='password')`

Oracle	`# Connecting to Oracle Express Edition`
	`import cx_Oracle`
	`conn = cx_Oracle.connect(dsn='localhost/XE',`
	`user='alice',`
	`password='password')`

РСУБД	Код

PostgreSQL	`import psycopg2`
	`conn = psycopg2.connect(host='localhost',`
	`database='my_new_db',`
	`user='alice',`
	`password='password')`

SQL Server	`# Connecting to SQL Server Express`
	`import pyodbc`
	`conn = pyodbc.connect(driver='{SQL Server}',`
	`host='localhost\SQLEXPRESS',`
	`database='my_new_db',`
	`user='alice',`
	`password='password')`

Не все аргументы обязательны. Если вы полностью исключите какой-либо аргумент, то будет использовано значение по умолчанию. Например, хостом по умолчанию является localhost, то есть ваш компьютер. Если имя пользователя и пароль не задавались, то эти аргументы можно не указывать.

ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ ПАРОЛЕЙ НА ЯЗЫКЕ PYTHON

Приведенный выше код подходит для проверки подключения к базе данных, но в реальности не стоит сохранять свой пароль в сценарии, чтобы его могли видеть все желающие.

Избежать этого можно с помощью множества способов, в том числе:

- сгенерировав SSH-ключ;
- установив переменные среды;
- создав конфигурационный файл.

Однако все эти варианты требуют дополнительных знаний о компьютерах или форматах файлов.

Рекомендуемый подход: создайте отдельный Python-файл.

Проще всего, на мой взгляд, сохранить имя пользователя и пароль в отдельном файле Python, а затем вызвать этот файл в сценарии подключения к базе данных. Такой способ менее безопасен, чем другие варианты, но при этом самый быстрый.

Чтобы использовать этот подход, начните с создания файла `db_config.py` со следующим кодом:

```
usr = "alice"
pwd = "password"
```

Импортируйте этот файл при настройке подключения к базе данных. В примере ниже код Oracle из табл. 2.3 изменен, чтобы можно было использовать значения файла `db_config.py` вместо жестко запрограммированных значений пользователя и пароля (изменения выделены жирным шрифтом):

```
import cx_Oracle
import db_config

conn = cx_Oracle.connect(dsn='localhost/XE',
       user=db_config.usr,
       password=db_config.pwd)
```

Этап 3. Написание SQL-кода на языке Python

После того как соединение с базой данных будет установлено, можно приступать к написанию SQL-запросов в своем коде на языке Python.

Напишите простой запрос для проверки подключения к базе данных:

```
cursor = conn.cursor()
cursor.execute('SELECT * FROM test;')
result = cursor.fetchall()
print(result)

[(1, 100),
 (2, 200)]
```

При использовании cx_Oracle в Python во избежание ошибки уберите точку с запятой (;) в конце всех запросов.

Сохраните результаты запроса в виде фрейма данных pandas:

```
# pandas must already be installed
import pandas as pd

df = pd.read_sql('''SELECT * FROM test;''', conn)
print(df)
print(type(df))

   id  num
0   1  100
1   2  200
<class 'pandas.core.frame.DataFrame'>
```

После завершения работы с базой данных закройте соединение:

```
cursor.close()
conn.close()
```

В целях экономии ресурсов всегда рекомендуется закрывать соединение с базой.

SQLALCHEMY ДЛЯ ЛЮБИТЕЛЕЙ PYTHON

Другой популярный способ подключиться к базе — использовать пакет SQLAlchemy в Python. Этот пакет представляет собой *объектно-реляционное отображение* (object relational mapper, ORM), который преобразует данные базы в объекты Python, что позволяет писать код на чистом Python, а не использовать синтаксис SQL.

Представьте, что хотите увидеть все имена таблиц в базе данных. (Приведенный ниже код предназначен для PostgreSQL, но SQLAlchemy будет работать с любой РСУБД.)

Без SQLAlchemy:

```
pd.read_sql("""SELECT tablename
            FROM pg_catalog.pg_tables
            WHERE schemaname='public'""", conn)
```

С SQLAlchemy:

```
conn.table_names()
```

При использовании SQLAlchemy объект `conn` поставляется с Python-методом `table_names()`, который, возможно, вам будет легче запомнить, чем синтаксис SQL. SQLAlchemy обеспечивает более чистый Python-код, однако снижает производительность из-за дополнительного времени, которое тратится на преобразование данных в Python-объекты.

Чтобы использовать SQLAlchemy в Python:

1) у вас уже должен быть установлен драйвер базы данных (например, `psycopg2`);
2) в окне терминала введите **`pip install sqlalchemy`** или **`conda install -c conda-forge sqlalchemy`** для установки SQLAlchemy;
3) выполните приведенный ниже код на языке Python для установки соединения с SQLAlchemy. (Код характерен для PostgreSQL.)

В документации по SQLAlchemy (https://oreil.ly/QadLc) приведен код для других РСУБД и драйверов:

```
from sqlalchemy import create_engine
conn = create_engine('postgresql+psycopg2://
       alice:password@localhost:5432/my_new_db')
```

Подключение R к базе данных

Подключение R к базе данных выполняется в три этапа:

1) установить драйвер базы данных для R;
2) настроить подключение к базе в R;
3) написать SQL-код в R.

Этап 1. Установка драйвера базы данных для R

Драйвер базы данных — это программное обеспечение, которое помогает R взаимодействовать с базой. Есть множество вариантов драйверов, из которых можно выбрать подходящий.

В табл. 2.4 приведен код установки популярного драйвера для каждой из РСУБД.

Эта установка выполняется один раз. Приведенный в таблице код должен быть запущен в R.

Таблица 2.4. Установка драйвера для R

РСУБД	Код
SQLite	`install.packages("RSQLite")`
MySQL	`install.packages("RMySQL")`

Oracle	Пакет ROracle можно загрузить со страницы Oracle ROracle Downloads (https://oreil.ly/Hgp6p).
	`setwd("folder_where_you_downloaded_ROracle")`
	
	`# Update the name of the .zip file based on the latest version`
	`install.packages("ROracle_1.3-2.zip", repos=NULL)`

PostgreSQL	`install.packages("RPostgres")`
SQL Server	В операционной системе Windows пакет odbc (Open Database Connectivity) является предустановленным. В macOS и Linux его можно загрузить со страницы Microsoft ODBC (https://oreil.ly/xrSP6). `install.packages("odbc")`

Этап 2. Настройка подключения к базе данных в R

Чтобы настроить соединение, необходимо знать местоположение и имя базы данных, к которой вы пытаетесь подключиться, а также имя пользователя и пароль. Более подробную информацию можно найти во врезке «Поля подключения к базе данных» выше в текущей главе.

В табл. 2.5 приведен код R, который необходимо выполнять каждый раз, когда вы планируете писать SQL-код в R. Его можно включить в начало сценария R.

Таблица 2.5. Код R для настройки подключения к базе данных

РСУБД	Код

SQLite	`library(DBI)`
	`con <- dbConnect(RSQLite::SQLite(),`
	`"my_new_db.db")`

MySQL	`library(RMySQL)`
	`con <- dbConnect(RMySQL::MySQL(),`
	`host="localhost",`
	`dbname="my_new_db",`
	`user="alice",`
	`password="password")`

Oracle	`library(ROracle)`
	`drv <- dbDriver("Oracle")`
	`con <- dbConnect(drv, "alice", "password",`
	`dbname="my_new_db")`

PostgreSQL	`library(RPostgres)`
	`con <- dbConnect(RPostgres::Postgres(),`
	`host="localhost",`
	`dbname="my_new_db",`
	`user="alice",`
	`password="password")`

SQL Server	`library(DBI)`
	`con <- DBI::dbConnect(odbc::odbc(),`
	`Driver="SQL Server",`
	`Server="localhost\\SQLEXPRESS",`
	`Database="my_new_db",`
	`User="alice",`
	`Password="password",`
	`Trusted_Connection="True")`

Не все аргументы обязательны. Если вы полностью исключите какой-либо аргумент, то будет использоваться значение по умолчанию.

- Например, хостом по умолчанию является localhost, то есть ваш компьютер.
- Если имя пользователя и пароль не задавались, то эти аргументы можно не указывать.

ЗАЩИТА ПАРОЛЕЙ В R

Приведенный выше код подходит для проверки подключения к базе данных, но в действительности не стоит сохранять свой пароль в сценарии, где его могут увидеть все.

Есть множество способов избежать этого, в том числе:

- шифрование учетных данных с помощью пакета `keyring`;
- создание конфигурационного файла с использованием пакета `config`;
- настройка переменных окружения с помощью файла `.Renviron`;
- запись пользователя и пароля в качестве глобальной опции в R благодаря команде `options`.

Рекомендуемый подход: запросить у пользователя пароль.

Наиболее простой подход, на мой взгляд, заключается в том, чтобы RStudio запрашивала пароль.

Вместо:

```
con <- dbConnect(...,
    password="password",
    ...)
```

сделайте так:

```
install.packages("rstudioapi")
con <- dbConnect(...,
    password=rstudioapi::askForPassword("Password?"),
    ...)
```

Этап 3. Написание SQL-кода в R

После того как соединение с базой установлено, можно приступать к написанию SQL-запросов в своем коде на языке R.

Чтобы показать все таблицы в базе данных, введите:

```
dbListTables(con)
```

```
[1] "test"
```

Если вы используете SQL Server, то укажите имя схемы, чтобы ограничить количество отображаемых таблиц, — dbListTables(con, schema="dbo"). Сочетание dbo означает «владелец базы данных» и является схемой по умолчанию в SQL Server.

Взгляните на тестовую таблицу в базе данных:

```
dbReadTable(con, "test")
```

```
  id num
1  1 100
2  2 200
```

В Oracle имя таблицы чувствительно к регистру. Поскольку Oracle автоматически преобразует имена таблиц в верхний регистр, то, скорее всего, придется выполнить следующую команду: dbRead Table(con, "TEST").

Напишите простой запрос и выведите фрейм данных:

```
df <- dbGetQuery(con, "SELECT * FROM test
                       WHERE id = 2")
print(df); class(df)
```

```
  id num
1  2 200
[1] "data.frame"
```

Закройте соединение, когда закончите работу с базой данных:

```
dbDisconnect(con)
```

В целях экономии ресурсов всегда рекомендуется закрывать соединение с базой.

ГЛАВА 3

Язык SQL

В этой главе рассматриваются основы языка SQL, в том числе его стандарты, ключевые термины и подъязыки, а также даются ответы на следующие вопросы:

- что такое ANSI SQL и чем он отличается от SQL;
- что такое ключевое слово и предложение;
- имеют ли значение заглавные буквы и пробелы;
- что находится за пределами оператора `SELECT`?

Сравнение с другими языками

Некоторые специалисты в сфере технологий не считают SQL настоящим языком программирования.

Хотя аббревиатура SQL расшифровывается как *«язык структурированных запросов»* (Structured Query Language), его нельзя использовать так же, как некоторые другие популярные языки программирования, например Python, Java или C++. С помощью этих языков вы можете написать код с указанием точных действий, которые должен выполнить компьютер, чтобы решить поставленную задачу. Это называется *императивным программированием*.

Если вы используете Python и хотите просуммировать список значений, то можете точно указать компьютеру, *как* именно

хотите это сделать. Код, показанный в примере ниже, проходит по списку элемент за элементом и добавляет каждое значение к промежуточной сумме, чтобы в итоге вычислить общую:

```
calories = [90, 240, 165]
total = 0
for c in calories:
    total += c
print(total)
```

Если вы используете SQL, то вместо того, чтобы объяснять компьютеру, *как* именно хотите что-то сделать, вы просто описываете, *что* хотите сделать; в данном случае — вычислить сумму. Скрытым образом SQL определяет, как оптимально выполнить код. Это называется *декларативным программированием*.

```
SELECT SUM(calories)
FROM workouts;
```

Исходя из всего вышесказанного, можно сделать главный вывод: SQL не является *языком программирования общего назначения*, как Python, Java или C++, который может использоваться для решения самых разных задач. SQL — *язык программирования специального назначения*, созданный для управления данными в реляционной БД.

РАСШИРЕНИЯ ДЛЯ SQL

По своей сути SQL — декларативный язык, но существуют расширения, благодаря которым он может выполнять дополнительные действия:

- Oracle имеет *процедурный язык SQL* (PL/SQL);
- в SQL Server имеется *транзакционный SQL* (T-SQL).

С помощью этих расширений вы можете, например, группировать код SQL в процедуры и функции и делать многие другие действия. Синтаксис не соответствует стандартам ANSI, но делает SQL гораздо более эффективным.

Стандарты ANSI

Американский национальный институт стандартов (ANSI) — организация, расположенная в США, которая занимается разработкой стандартов на все, начиная от питьевой воды и заканчивая гайками и болтами.

SQL стал стандартом ANSI в 1986 году. В 1989-м был опубликован очень подробный документ со спецификациями (думаю, объемом в сотни страниц) на тему того, что должен уметь делать язык баз данных и как это должно происходить. Каждые несколько лет стандарты обновляются, поэтому можно встретить такие термины, как ANSI-89 и ANSI-92, которые представляли собой разные наборы стандартов SQL, которые были добавлены в 1989 и 1992 годах соответственно. Последним стандартом является ANSI SQL2016.

SQL В СРАВНЕНИИ С ANSI SQL И MYSQL

SQL — общий термин, обозначающий язык структурированных запросов.

Под *ANSI SQL* понимается SQL-код, который соответствует стандартам ANSI и работает в любой реляционной системе управления базами данных (РСУБД).

MySQL — одна из многих вариантов РСУБД. В ней можно писать как ANSI-код, так и свойственный этой системе SQL-код.

В числе других вариантов РСУБД — *Oracle*, *PostgreSQL*, *SQL Server*, *SQLite* и др.

Даже при наличии стандартов ни одна из двух РСУБД не является абсолютно одинаковой. Хотя некоторые из систем стремятся к полной совместимости с ANSI, все они совместимы с ним лишь частично. В конечном итоге каждый производитель сам выбирает, какие внедрять стандарты и какие

создавать дополнительные функции, которые будут работать только в его программном обеспечении.

ДОЛЖНЫ ЛИ ВЫ СЛЕДОВАТЬ СТАНДАРТАМ

Бóльшая часть базового SQL-кода, который вы пишете, соответствует стандартам ANSI. Если вы обнаружите код, который выполняет некие сложные операции, используя простые, но незнакомые ключевые слова, то велика вероятность того, что он выходит за рамки стандартов.

Если вы работаете исключительно в рамках одной РСУБД, например Oracle или SQL Server, то совершенно нормально не придерживаться стандартов ANSI и использовать все возможности этого программного обеспечения.

Проблема возникает, когда код, работающий в одной РСУБД, требуется использовать в другой. Код, не соответствующий стандарту ANSI, скорее всего, в новой системе не будет работать, и его придется переписывать.

Допустим, у вас есть вот такой запрос, который работает в Oracle. Он не соответствует стандартам ANSI, поскольку функция `DECODE` доступна только в Oracle, но не в другом программном обеспечении. Если скопировать запрос в SQL Server, то код не будет выполняться:

```
-- код, характерный для Oracle
SELECT item, DECODE (flag, 0, 'No', 1, 'Yes')
             AS Yes_or_No
FROM items;
```

Запрос ниже имеет ту же логику, но вместо нее используется оператор `CASE`, который является стандартом ANSI. Благодаря этому он будет работать в Oracle, SQL Server и других программах:

```
-- Код, работающий в любой РСУБД
SELECT item, CASE WHEN flag = 0 THEN 'No'
             ELSE 'Yes' END AS Yes_or_No
FROM items;
```

КАКОЙ СТАНДАРТ ВЫБРАТЬ

Два блока кода, приведенные ниже, выполняют соединение, используя два различных стандарта. Первым широко распространенным стандартом был ANSI-89, за ним последовал ANSI-92, в который были внесены некоторые существенные изменения.

```
-- ANSI-89
SELECT c.id, c.name, o.date
FROM customer c, order o
WHERE c.id = o.id;

-- ANSI-92
SELECT c.id, c.name, o.date
FROM customer c INNER JOIN order o
ON c.id = o.id;
```

Если вы пишете новый SQL-код, то я бы рекомендовала использовать либо последний стандарт (которым в настоящее время является ANSI SQL2016), либо синтаксис, приведенный в документации к РСУБД, в которой вы работаете.

Тем не менее знать более ранние стандарты важно, ведь если ваша компания существует уже несколько десятилетий, то вы, скорее всего, столкнетесь с более старым кодом.

Термины SQL

Ниже представлен блок SQL-кода, который показывает количество продаж, закрытых каждым сотрудником в 2021 году. На примере этого блока я поясню ряд терминов SQL.

```
-- Продажи, выполненные в 2021 году
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Ключевые слова и функции

Ключевые слова и функции — это термины, встроенные в SQL.

Ключевые слова

Ключевое слово — это текст, который имеет определенное значение в SQL. Все ключевые слова в этом блоке кода выделены жирным шрифтом:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

SQL НЕЧУВСТВИТЕЛЕН К РЕГИСТРУ

В целях удобочитаемости ключевые слова обычно пишутся с заглавной буквы. Однако SQL нечувствителен к регистру, то есть прописные `WHERE` и строчные `where` означают одно и то же при выполнении кода.

Функции

Функция — особый тип ключевого слова. Она принимает ноль или больше входных данных, выполняет с ними определенные действия и возвращает результат (вывод). В SQL после имени функции обычно следуют круглые скобки, но не всегда. В блоке кода ниже две функции выделены жирным шрифтом:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Есть четыре категории функций: числовые, строковые, функции даты и времени и др. Например:

- `COUNT()` — числовая функция. Она принимает столбец и возвращает количество ненулевых строк (строк, имеющих значение);
- `YEAR()` — функция даты. Она принимает столбец с типом данных `date` или `datetime`, извлекает годы и возвращает значения в виде нового столбца.

Перечень распространенных функций SQL можно найти в табл. 7.2.

Идентификаторы и псевдонимы

Идентификаторы и псевдонимы — это термины, которые определяет пользователь.

Идентификаторы

Идентификатор — это имя объекта базы данных, например таблицы или столбца. В блоке кода ниже все идентификаторы выделены жирным шрифтом:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Идентификаторы должны начинаться с буквы (`a-z` или `A-Z`), за которой следует любая комбинация букв, цифр и знаков подчеркивания (`_`). В некоторых программах допускается использование дополнительных символов, таких как `@`, `#` и `$`.

В целях удобочитаемости идентификаторы обычно пишутся строчными буквами, а ключевые слова — прописными, хотя код будет выполняться независимо от регистра.

Как правило, не рекомендуется присваивать идентификаторам имя, которое совпадает с существующим ключевым словом. Например, не следует называть столбец COUNT, поскольку это слово уже является ключевым в SQL.

Если вы все же решили так поступить, то во избежание путаницы идентификатор можно взять в двойные кавычки. Например, вместо того, чтобы присваивать столбцу имя COUNT, можно назвать его "COUNT", но лучше использовать совершенно другое имя, скажем num_sales.

В MySQL идентификаторы вместо двойных кавычек ("") берутся в левые одиночные (``).

Псевдонимы

Псевдоним (alias) временно переименовывает столбец или таблицу, только на время выполнения запроса. Другими словами, в результатах запроса будут отображаться новые имена псевдонимов, но исходные имена столбцов в таблицах, к которым выполняется запрос, останутся неизменными. В блоке кода ниже все псевдонимы выделены жирным шрифтом:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Стандартным является использование ключевого слова `AS` при переименовании столбцов (`AS num_sales`) и отсутствие дополнительного текста при переименовании таблиц (`e`). Однако технически любой из этих синтаксисов работает как для столбцов, так и для таблиц.

Помимо работы со столбцами и таблицами, псевдонимы полезны, если необходимо временно присвоить имя подзапросу.

Операторы и предложения

Это способы обращения к подмножествам кода SQL.

Операторы

Оператор начинается с ключевого слова и заканчивается точкой с запятой. Весь этот блок кода называется оператором `SELECT`, поскольку начинается с ключевого слова `SELECT`:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Многие инструменты работы с базами данных, предоставляющие графический интерфейс, не требуют ставить точку с запятой (;) в конце оператора.

Оператор `SELECT` является наиболее популярным типом операторов SQL, и его часто называют запросом, поскольку он находит данные в базе. Другие типы операторов рассматриваются в разделе «Подъязыки» текущей главы.

Предложения

Предложение — это способ ссылки на определенный раздел оператора. Вот наш исходный оператор `SELECT`:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
```

```
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Этот оператор содержит четыре основных предложения:

- SELECT:

  ```
  SELECT e.name, COUNT(s.sale_id) AS num_sales
  ```

- FROM:

  ```
  FROM employee e
    LEFT JOIN sales s ON e.emp_id = s.emp_id
  ```

- WHERE:

  ```
  WHERE YEAR(s.sale_date) = 2021
    AND s.closed IS NOT NULL
  ```

- GROUP BY:

  ```
  GROUP BY e.name;
  ```

В разговоре часто можно услышать, что люди ссылаются на раздел оператора, например «посмотрите на таблицы в предложении FROM». Это полезный способ указать на конкретный участок кода.

На самом деле в этом высказывании больше предложений, чем четыре перечисленных. В грамматике это часть предложения, содержащая подлежащее и сказуемое. Таким образом, можно сослаться на следующее:

```
LEFT JOIN sales s ON e.emp_id = s.emp_id
```

как предложение LEFT JOIN, если вы хотите еще более точно определить участок кода, на который ссылаетесь.

Шесть наиболее популярных предложений начинаются с SELECT, FROM, WHERE, GROUP BY, HAVING и ORDER BY и подробно рассматриваются в главе 4.

Выражения и предикаты

Это комбинации функций, идентификаторов и т. д.

Выражения

Выражение можно рассматривать как формулу, результатом которой является значение. Выражение в блоке кода имело вид:

```
COUNT(s.sale_id)
```

Это выражение включает в себя функцию (`COUNT`) и идентификатор (`s.sale_id`). Вместе они образуют выражение, которое говорит о необходимости подсчета количества продаж.

Другими примерами выражений являются:

- `s.sale_id + 10` — числовое выражение, включающее в себя основные математические операции;
- `CURRENT_DATE` — выражение даты и времени, простая функция, которая возвращает текущую дату.

Предикаты

Предикат — это логическое сравнение, результатом которого является одно из трех значений: `TRUE`, или `FALSE`, или `UNKNOWN`. Иногда их называют условными операторами. В этом блоке кода три предиката выделены жирным шрифтом:

```
SELECT e.name, COUNT(s.sale_id) AS num_sales
FROM employee e
  LEFT JOIN sales s ON e.emp_id = s.emp_id
WHERE YEAR(s.sale_date) = 2021
  AND s.closed IS NOT NULL
GROUP BY e.name;
```

Из этих примеров можно сделать следующие выводы:

- знак равенства (=) — наиболее популярная операция сравнения величин;
- NULL означает отсутствие значения. При проверке отсутствия значения в поле вместо = NULL пишется IS NULL.

Комментарии, кавычки и пробелы

Это знаки препинания, имеющие значение в SQL.

Комментарии

Комментарий — это текст, который игнорируется при выполнении кода, как, например, в следующем примере:

```
-- Продажи закрыты в 2021 году
```

Комментарии полезно добавлять в код для того, чтобы другие рецензенты вашего кода (в том числе и вы сами!) могли быстро понять назначение кода, не читая его целиком.

Чтобы прокомментировать код, используются:

- одна строка текста:
  ```
  -- Это мои комментарии
  ```
- несколько строк текста:
  ```
  /* Это
  мои комментарии */
  ```

Кавычки

В SQL можно использовать два типа кавычек — одинарные и двойные:

```
SELECT "This column"
FROM my_table
WHERE name = 'Bob';
```

- *Одиночные кавычки: строки.* Посмотрите на слово 'Bob'. Одинарные кавычки используются при ссылке на строко-

вое значение. На практике одинарные кавычки встречаются гораздо чаще, чем двойные.

- *Двойные кавычки: идентификаторы.* Взгляните на выражение `"This column"`. Двойные кавычки используются при ссылке на идентификатор. В данном случае, поскольку между словами `This` и `column` стоит пробел, двойные кавычки необходимы для того, чтобы `This column` интерпретировалось как имя столбца. Без двойных кавычек SQL выдал бы ошибку из-за пробела. Тем не менее при именовании столбцов вместо пробелов лучше ставить знак подчеркивания (_), чтобы избежать использования двойных кавычек.

В MySQL идентификаторы вместо двойных кавычек ("") берутся в левые одиночные (``).

Пробел

Для SQL неважно количество пробелов между терминами. Будь то один пробел, табуляция или новая строка, SQL выполнит запрос, начиная с первого ключевого слова и заканчивая точкой с запятой в конце оператора. Два показанных ниже запроса эквивалентны:

```
SELECT * FROM my_table;

SELECT *
  FROM my_table;
```

Простые SQL-запросы могут занимать одну строку кода. В более длинных запросах, состоящих из десятков и даже сотен строк, можно увидеть новые строки для новых предложений, вкладки при перечислении большого количества столбцов или таблиц и т. д.

Конечная цель — читабельный код, поэтому вы должны решить, как расположить код (или следовать рекомендациям вашей компании), чтобы он выглядел чистым и его можно было быстро просмотреть.

Подъязыки

Есть множество типов операторов, которые могут быть написаны на языке SQL. Все они относятся к одному из пяти подъязыков, которые подробно описаны в табл. 3.1.

Таблица 3.1. Подъязыки SQL

Подъязык	Описание	Общие команды	Справочные разделы
Язык запроса данных (Data Query Language, DQL)	Именно с этим языком знакомо большинство людей. Эти операторы используются для получения информации из объекта базы данных, например таблицы, и часто называются SQL-запросами	`SELECT`	Бóльшая часть этой книги посвящена DQL

Язык определения данных (Data Definition Language, DDL)	Это язык, используемый для определения или создания объекта базы данных, например таблицы или индекса	`CREATE`
		`ALTER`
		`DROP`
			Создание, обновление и удаление

Язык манипулирования данными (Data Manipulation Language, DML)	Это язык, используемый для манипулирования данными в базе или их изменения	`INSERT`
		`UPDATE`
		`DELETE`
			Создание, обновление и удаление

Язык управления данными (Data Control Language, DCL)	Это язык, используемый для управления доступом к данным в базе, который иногда называют разрешениями или привилегиями	`GRANT` `REVOKE`	Не обсуждается
Язык управления транзакциями (Transaction Control Language, TCL)	Это язык, используемый для управления транзакциями в базе или для внесения постоянных изменений в нее	`COMMIT` `ROLLBACK`	Управление транзакциями

Большинство аналитиков данных и специалистов по исследованию данных пишут DQL-операторы `SELECT` для запросов к таблицам. Однако важно знать: администраторы БД и инженеры по обработке данных также будут писать на этих других подъязыках код, предназначенный для обслуживания базы.

КРАТКОЕ ОПИСАНИЕ ЯЗЫКА SQL

- ANSI SQL — стандартизированный код SQL, который работает во всех программах баз данных. Многие РСУБД имеют расширения, которые не соответствуют стандартам, но увеличивают функциональность их программного обеспечения.
- Ключевые слова — это термины, которые зарезервированы в SQL и имеют специальное значение.
- Предложения относятся к определенным разделам оператора. Обычно используются следующие предложения: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` и `ORDER BY`.
- Заглавные буквы и пробелы не влияют на выполнение кода в SQL, но существуют рекомендации по улучшению читабельности.
- Помимо операторов `SELECT`, существуют операторы для определения объектов, манипулирования данными и т. д.

ГЛАВА 4

Основы работы с запросами

Запрос — это псевдоним оператора `SELECT`, который состоит из шести основных предложений. В каждом разделе данной главы подробно рассматривается то или иное предложение:

1) `SELECT`;
2) `FROM`;
3) `WHERE`;
4) `GROUP BY`;
5) `HAVING`;
6) `ORDER BY`.

В последнем разделе этой главы рассматривается предложение `LIMIT`, которое поддерживается MySQL, PostgreSQL и SQLite.

Примеры кода, приведенные в этой главе, ссылаются на четыре таблицы:

1) `waterfall` — водопады на Верхнем полуострове штата Мичиган;
2) `owner` — владельцы водопадов;
3) `county` — округа, в которых расположены водопады;
4) `tour` — экскурсионные туры, состоящие из нескольких остановок у водопадов.

Ниже приведен пример запроса, в котором используются шесть основных предложений. За ним следуют результаты запроса, которые также называются *результирующим набором*.

```
-- Туры с двумя и более общественными водопадами
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
FROM     tour t LEFT JOIN waterfall w
         ON t.stop = w.id
WHERE    w.open_to_public = 'y'
GROUP BY t.name
HAVING   COUNT(*) >= 2
ORDER BY tour_name;

tour_name   num_waterfalls
----------  ---------------
M-28                      6
Munising                  6
US-2                      4
```

Запрос к базе данных означает получение данных из базы, обычно из таблицы или нескольких таблиц.

Можно запросить представление вместо таблицы. Представления выглядят как таблицы и являются производными от них, но сами не содержат никаких данных. Дополнительную информацию о представлениях можно найти в разделе «Представления» главы 5.

Предложение SELECT

В предложении `SELECT` указываются столбцы, которые должен возвращать оператор.

В предложении `SELECT` за ключевым словом `SELECT` следует список имен столбцов и/или выражений, разделенных запятыми. Каждое имя столбца и/или выражение становится столбцом в результатах.

Выбор столбцов

В простейшем предложении SELECT перечисляются имена одного или нескольких столбцов из таблиц, указанных в предложении FROM:

```
SELECT id, name
FROM owner;

id     name
-----  ----------------
    1  Pictured Rocks
    2  Michigan Nature
    3  AF LLC
    4  MI DNR
    5  Horseshoe Falls
```

Выбор всех столбцов

Чтобы вернуть все столбцы из таблицы, можно не записывать имя каждого столбца, а использовать одну звездочку (*):

```
SELECT *
FROM owner;

id     name              phone          type
-----  ----------------  -------------  --------
    1  Pictured Rocks    906.387.2607   public
    2  Michigan Nature   517.655.5655   private
    3  AF LLC private
    4  MI DNR            906.228.6561   public
    5  Horseshoe Falls   906.387.2635   private
```

Звездочка — полезное сокращение при тестировании запросов, так как позволяет вам сэкономить много времени на вводе текста. Однако использовать ее в производственном коде рискованно: столбцы таблицы могут меняться со временем, и это приведет к сбою кода, если их окажется меньше или больше, чем ожидалось.

Выбор выражений

Помимо простого перечисления столбцов, в предложении SELECT можно перечислить и более сложные выражения, которые будут возвращаться в качестве столбцов в результатах.

Следующий оператор содержит выражение для расчета сокращения численности населения на 10 %, округленное до нуля знаков после запятой:

```
SELECT name, ROUND(population * 0.9, 0)
FROM county;

name         ROUND(population * 0.9, 0)
----------   --------------------------
Alger                              8876
Baraga                             7871
Ontonagon                          7036
...
```

Выбор функций

Выражения в списке SELECT обычно ссылаются на столбцы таблиц, из которых производится выборка, но бывают и исключения. Например, широко используемой функцией, которая не ссылается ни на какие таблицы, является функция, возвращающая текущую дату:

```
SELECT CURRENT_DATE;

CURRENT_DATE
-------------
2021-12-01
```

Приведенный выше код работает в MySQL, PostgreSQL и SQLite.

Эквивалентный код, работающий в других РСУБД, можно найти в разделе «Функции даты и времени» главы 7.

Большинство запросов содержат как предложение SELECT, так и предложение FROM, но при использовании определенных функций базы данных, таких как CURRENT_DATE, требуется только SELECT.

Кроме того, в предложение `SELECT` можно добавлять выражения, которые являются *подзапросами* (запрос, вложенный в другой запрос). Более подробная информация приведена в подразделе «Выбор подзапросов» далее в этой главе.

Псевдонимы столбцов

Псевдоним столбца предназначен для того, чтобы дать временное имя любому столбцу или выражению, перечисленному в предложении `SELECT`. Это временное имя, или псевдоним столбца, затем отображается как имя столбца в результатах.

Обратите внимание, что такое изменение имени не является постоянным, поскольку имена столбцов в исходных таблицах остаются прежними. Псевдоним существует только внутри запроса.

Данный код выводит на экран три столбца:

```
SELECT id, name,
ROUND(population * 0.9, 0)
FROM county;

id     name         ROUND(population * 0.9, 0)
-----  -----------  --------------------------
    2  Alger                              8876
    6  Baraga                             7871
    7  Ontonagon                          7036
...
```

Допустим, мы хотим переименовать названия столбцов в результатах. Имя `id` — слишком неоднозначное, и мы хотели бы дать более описательное. А `ROUND(population * 0.9, 0)` — слишком длинное, и мы хотели бы дать столбцу более простое имя.

Чтобы создать псевдоним столбца, после имени столбца или выражения нужно добавить либо имя псевдонима, либо ключевое слово `AS` и имя псевдонима:

```
-- alias_name
SELECT id county_id, name,
       ROUND(population * 0.9, 0) estimated_pop
FROM county;
```

или:

```
-- AS alias_name
SELECT id AS county_id, name,
       ROUND(population * 0.90, 0) AS estimated_pop
FROM county;

county_id   name        estimated_pop
----------  ----------  --------------
         2  Alger                 8876
         6  Baraga                7871
         7  Ontonagon             7036
...
```

На практике при создании псевдонимов используются оба варианта. В случае предложения `SELECT` второй вариант более популярен, поскольку ключевое слово `AS` облегчает визуальный поиск имен столбцов и псевдонимов среди длинного списка имен столбцов.

Более старые версии PostgreSQL требуют использования AS при создании псевдонима столбца.

Псевдонимы столбцов не являются обязательными, однако настоятельно рекомендуется использовать их при работе с выражениями, чтобы дать адекватные имена столбцам в результатах.

Псевдонимы с учетом регистра и пунктуации

Как видно на примере псевдонимов столбцов `county_id` и `estimated_pop`, при именовании псевдонимов столбцов принято использовать строчные буквы с подчеркиванием вместо пробелов.

Вдобавок можно создавать псевдонимы, содержащие заглавные буквы, пробелы и знаки препинания, используя синтаксис двойных кавычек, как показано в данном примере:

```
SELECT id AS "Waterfall #",
  name AS "Waterfall Name"
FROM waterfall;

Waterfall #   Waterfall Name
-----------   --------------
          1   Munising Falls
          2   Tannery Falls
          3   Alger Falls
...
```

Уточнение столбцов

Допустим, вы пишете запрос, который извлекает данные из двух таблиц, и обе они содержат столбец `name`. Если бы вы просто включили `name` в предложение `SELECT`, то код не знал бы, к какой таблице вы обращаетесь.

Для решения этой проблемы можно *уточнить* имя столбца, указав имя его таблицы. Другими словами, можно присвоить столбцу префикс, указывающий, к какой таблице он принадлежит, используя *точечную нотацию*, как, например, `table_name.column_name`.

В примере ниже выполняется запрос к одной таблице, и хотя здесь нет необходимости определять столбцы, это показано для демонстрации. Вот как следует уточнить столбец по имени таблицы:

```
SELECT owner.id, owner.name
FROM owner;
```

Если в SQL возникает ошибка, связанная с неоднозначным именем столбца, это означает, что в нескольких таблицах в запросе есть столбец с одним и тем же именем и вы не указали, к какой комбинации таблиц и столбцов обращаетесь. Устранить ошибку можно, уточнив имя столбца.

Уточнение таблиц

Если вы уточняете имя столбца по имени таблицы, то можете также определить ее имя по имени ее базы данных или схемы. Этот запрос извлекает данные именно из таблицы `owner` в схеме `sqlbook`:

```
SELECT sqlbook.owner.id, sqlbook.owner.name
FROM sqlbook.owner;
```

Код получился длинным, так как `sqlbook.owner` повторяется несколько раз. Чтобы сэкономить на вводе текста, можно указать *псевдоним таблицы*. В следующем примере таблице `owner` присвоен псевдоним `o`:

```
SELECT o.id, o.name
FROM sqlbook.owner o;
```

или:

```
SELECT o.id, o.name
FROM owner o;
```

СРАВНЕНИЕ ПСЕВДОНИМОВ СТОЛБЦОВ С ПСЕВДОНИМАМИ ТАБЛИЦ

Псевдонимы столбцов определяются в предложении `SELECT` в целях переименования столбца в результатах. Обычно содержат ключевое слово `AS`, хотя это и не обязательно.

```
-- Псевдоним столбца
SELECT num AS new_col
FROM my_table;
```

Псевдонимы таблиц определяются в предложении `FROM` в целях создания временного псевдонима для таблицы. Обычно принято исключать ключевое слово `AS`, хотя работает и его добавление.

```
-- Псевдоним таблицы
SELECT *
FROM my_table mt;
```

Выбор подзапросов

Подзапрос — это запрос, вложенный в другой запрос. Подзапросы могут располагаться в различных предложениях, в том числе и в `SELECT`.

В следующем примере, помимо `id`, `name` и `population`, допустим, мы хотим увидеть еще и среднюю численность населения всех округов. Добавив подзапрос, мы создадим в результатах новый столбец для средней численности населения:

```
SELECT id, name, population,
       (SELECT AVG(population) FROM county)
       AS average_pop
FROM county;

id     name         population   average_pop
-----  ----------  -----------  ------------
    2  Alger              9862         18298
    6  Baraga             8746         18298
    7  Ontonagon          7818         18298
...
```

Здесь следует отметить несколько моментов.

- Подзапрос должен быть заключен в круглые скобки.
- При написании подзапроса в предложении `SELECT` настоятельно рекомендуется указывать псевдоним столбца, которым в данном случае является `average_pop`. Таким образом, столбец в результатах будет иметь простое имя.
- В столбце `average_pop` есть только одно значение, которое повторяется во всех строках. При добавлении подзапроса в предложение `SELECT` результат подзапроса должен возвращать один столбец и либо ноль, либо одну строку, как показано в этом подзапросе для расчета средней численности населения:

  ```
  SELECT AVG(population) FROM county;

  AVG(population)
  ---------------
            18298
  ```

- Если подзапрос вернул нулевые строки, то новый столбец будет заполнен значениями NULL.

НЕКОРРЕЛИРОВАННЫЕ И КОРРЕЛИРОВАННЫЕ ПОДЗАПРОСЫ

Предыдущий пример представляет собой *некоррелированный подзапрос*, то есть подзапрос не ссылается на внешний запрос. Подзапрос может выполняться самостоятельно, независимо от внешнего запроса.

Другой тип подзапроса называется *коррелированным*, и он действительно ссылается на значения во внешнем запросе. Как следствие, время обработки часто значительно замедляется, поэтому лучше переписать запрос, используя вместо него JOIN. Ниже приведен пример коррелированного подзапроса, а также более эффективный код.

Проблемы производительности при использовании коррелированных подзапросов

Запрос, показанный ниже, возвращает количество водопадов для каждого владельца. Обратите внимание: шаг `o.id = w.owner_id` в подзапросе ссылается на таблицу `owner` во внешнем запросе, что делает его коррелированным подзапросом.

```
SELECT o.id, o.name,
       (SELECT COUNT(*) FROM waterfall w
       WHERE o.id = w.owner_id) AS num_waterfalls
FROM owner o;

id     name              num_waterfalls
-----  ----------------  --------------
    1  Pictured Rocks                 3
    2  Michigan Nature                3
    3  AF LLC                         1
    4  MI DNR                         1
    5  Horseshoe Falls                0
```

Более правильным подходом было бы переписать запрос с помощью JOIN. В этом случае сначала соединяются таблицы,

а затем выполняется остальная часть запроса, что гораздо быстрее, чем повторное выполнение подзапроса для каждой строки данных. Дополнительную информацию о соединениях можно найти в разделе «Соединение таблиц» главы 9.

```
SELECT   o.id, o.name,
         COUNT(w.id) AS num_waterfalls
FROM     owner o LEFT JOIN waterfalls w
         ON o.id = w.owner_id
GROUP BY o.id, o.name

id     name              num_waterfalls
-----  ----------------  ---------------
    1  Pictured Rocks                  3
    2  Michigan Nature                 3
    3  AF LLC                          1
    4  MI DNR                          1
    5  Horseshoe Falls                 0
```

Ключевое слово DISTINCT

Если в предложении `SELECT` указан столбец, то по умолчанию возвращаются все строки. Для большей ясности можно добавить ключевое слово `ALL`, но это не обязательно. Эти запросы возвращают каждую комбинацию `type/open_to_public`:

```
SELECT o.type, w.open_to_public
FROM owner o
JOIN waterfall w ON o.id = w.owner_id;
```

или:

```
SELECT ALL o.type, w.open_to_public
FROM owner o
JOIN waterfall w ON o.id = w.owner_id;

type      open_to_public
--------  ---------------
public    y
public    y
public    y
private   y
private   y
```

```
private   y
private   y
public    y
```

Если необходимо удалить из результатов повторяющиеся строки, то можно воспользоваться ключевым словом `DISTINCT`. Этот запрос возвращает список уникальных комбинаций `type/open_to_public`:

```
SELECT DISTINCT o.type, w.open_to_public
FROM owner o
JOIN waterfall w ON o.id = w.owner_id;

type      open_to_public
--------  ---------------
public    y
private   y
```

COUNT и DISTINCT

Чтобы подсчитать количество уникальных значений в *одном столбце*, можно объединить ключевые слова `COUNT` и `DISTINCT` в предложении `SELECT`. Этот запрос возвращает количество уникальных значений столбца `type`:

```
SELECT COUNT(DISTINCT type) AS unique
FROM owner;

unique
-------
      2
```

Чтобы подсчитать количество уникальных комбинаций нескольких столбцов, можно обернуть запрос `DISTINCT` в подзапрос, а затем выполнить `COUNT` для подзапроса. Этот запрос возвращает количество уникальных комбинаций `type/open_to_public`:

```
SELECT COUNT(*) AS num_unique
FROM (SELECT DISTINCT o.type, w.open_to_public
     FROM owner o JOIN waterfall w
     ON o.id = w.owner_id) my_subquery;
```

```
num_unique
-----------
          2
```

MySQL и PostgreSQL поддерживают использование синтаксиса `COUNT(DISTINCT)` для нескольких столбцов. Два этих запроса эквивалентны предыдущему запросу и не требуют подзапроса:

```
-- эквивалент MySQL
SELECT COUNT(DISTINCT o.type, w.open_to_public)
       AS num_unique
       FROM owner o JOIN waterfall w
            ON o.id = w.owner_id;

-- эквивалент PostgreSQL
SELECT COUNT(DISTINCT (o.type, w.open_to_public))
       AS num_unique
       FROM owner o JOIN waterfall w
            ON o.id = w.owner_id;

num_unique
-----------
          2
```

Предложение FROM

С помощью предложения `FROM` вы можете указать источник данных, которые хотите получить. В простейшем случае в предложении `FROM` запроса указывается одна таблица или представление.

```
SELECT name
FROM waterfall;
```

Вы можете уточнить таблицу или представление, указав имя базы данных или схемы с помощью точечной нотации. Этот запрос извлекает данные конкретно из таблицы `waterfall` в схеме `sqlbook`:

```
SELECT name
FROM sqlbook.waterfall;
```

Получение данных из нескольких таблиц

Вместо того чтобы получать данные из одной таблицы, часто требуется объединить данные из нескольких. Самый распространенный способ сделать это — использовать предложение `JOIN` внутри предложения `FROM`. Этот запрос извлекает данные из таблиц `Waterfall` и `Tour` и выводит одну таблицу результатов:

```
SELECT *
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

id     name             ... name       stop  ...
-----  ---------------      ---------  -----
    1  Munising Falls       M-28           1
    1  Munising Falls       Munising       1
    2  Tannery Falls        Munising       2
    3  Alger Falls          M-28           3
    3  Alger Falls          Munising       3
...
```

Разберем каждую часть блока кода.

Псевдонимы таблиц

```
waterfall w JOIN tour t
```

Таблицам `Waterfall` и `Tour` присвоены псевдонимы `w` и `t`, которые являются временными именами таблиц в запросе. Псевдонимы таблиц необязательны в предложении `JOIN`, но очень помогают сокращать имена таблиц, на которые необходимо ссылаться в предложениях `ON` и `SELECT`.

JOIN ... ON ...

```
waterfall w JOIN tour t
ON w.id = t.stop
```

Эти две таблицы соединяются с помощью ключевого слова `JOIN`. За предложением `JOIN` всегда следует предложение `ON`,

которое определяет, как таблицы должны быть связаны друг с другом. В данном случае идентификатор водопада в таблице `Waterfall` должен совпадать со значением остановки с водопадом в таблице `Tour`.

Вы можете увидеть, что предложения FROM, JOIN и ON расположены на разных строках или с отступом. Это не обязательно, но улучшает удобочитаемость, особенно при соединении большого количества таблиц.

Таблица результатов

Результатом запроса всегда является одна таблица. В таблице `waterfall` 12 столбцов, а в таблице `tour` — три. После соединения таблица результатов содержит 15 столбцов.

```
id     name            ... name       stop ...
-----  --------------      ---------  -----
    1 Munising Falls       M-28           1
    1 Munising Falls       Munising       1
    2 Tannery Falls        Munising       2
    3 Alger Falls          M-28           3
    3 Alger Falls          Munising       3
...
```

Вы заметите, что в таблице результатов есть два столбца с именем `name`. Первый — из таблицы `Waterfall`, а второй — из таблицы `Tour`. Чтобы сослаться на них в предложении `SELECT`, необходимо уточнить имена столбцов.

```
SELECT w.name, t.name
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

name             name
---------------  ---------
Munising Falls   M-28
Munising Falls   Munising
Tannery Falls    Munising
...
```

Чтобы различать эти два столбца, необходимо присвоить псевдонимы именам столбцов.

```
SELECT w.name AS waterfall_name,
       t.name AS tour_name
FROM waterfall w JOIN tour t
     ON w.id = t.stop;

waterfall_name   tour_name
---------------  ----------
Munising Falls   M-28
Munising Falls   Munising
Tannery Falls    Munising
Alger Falls      M-28
Alger Falls      Munising
...
```

Варианты JOIN

В предыдущем примере, если `waterfall` не указана ни в одном туре, то не появится в таблице результатов. Если вы хотите видеть в результатах данные обо всех водопадах, то необходимо использовать другой тип соединения.

> **ПО УМОЛЧАНИЮ JOIN ОЗНАЧАЕТ INNER JOIN**
>
> В этом примере используется простое ключевое слово `JOIN` для объединения данных из двух таблиц, хотя лучше всего явно указывать тип используемого соединения. Само по себе `JOIN` по умолчанию является `INNER JOIN`; это означает, что в результатах будут возвращены только те записи, которые содержатся в обеих таблицах.

В SQL используются различные типы соединений; о них мы более подробно поговорим в разделе «Соединение таблиц» главы 9.

Получение данных из подзапросов

Подзапрос — это запрос, вложенный в другой запрос. Подзапросы в предложении `FROM` должны быть автономными операторами `SELECT`, то есть они не ссылаются на внешний запрос и могут выполняться самостоятельно.

Подзапрос в предложении FROM также известен как производная таблица, поскольку в конечном итоге подзапрос фактически действует как таблица в течение всего времени выполнения запроса.

В этом запросе перечислены все водопады, находящиеся в публичной собственности, причем часть, содержащая подзапрос, выделена жирным шрифтом:

```
SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM (SELECT * FROM owner WHERE type = 'public') o
     JOIN waterfall w
     ON o.id = w.owner_id;

waterfall_name   owner_name
--------------   --------------
Little Miners    Pictured Rocks
Miners Falls     Pictured Rocks
Munising Falls   Pictured Rocks
Wagner Falls     MI DNR
```

Важно понимать, в каком порядке выполняется запрос.

Этап 1. Выполнение подзапроса

Сначала выполняется содержимое подзапроса. Видно, что в результате получается таблица, содержащая только информацию о типе владельцев `public`:

```
SELECT * FROM owner WHERE type = 'public';

id     name             phone          type
-----  ---------------  -------------  -------
    1  Pictured Rocks   906.387.2607   public
    4  MI DNR           906.228.6561   public
```

Если вернуться к исходному запросу, то можно заметить, что за подзапросом сразу следует буква `o`. Это временное имя (или псевдоним), которое присваивается результатам подзапроса.

Псевдонимы необходимы для подзапросов в предложении FROM в MySQL, PostgreSQL и SQL Server, но не в Oracle и SQLite.

Этап 2. Выполнение всего запроса

Далее можно представить, что на месте подзапроса стоит буква o. Теперь запрос выполняется как обычно:

```
SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM o JOIN waterfall w
     ON o.id = w.owner_id;

waterfall_name   owner_name
--------------   --------------
Little Miners    Pictured Rocks
Miners Falls     Pictured Rocks
Munising Falls   Pictured Rocks
Wagner Falls     MI DNR
```

ПОДЗАПРОСЫ В СРАВНЕНИИ С ПРЕДЛОЖЕНИЕМ WITH

В качестве альтернативы написанию подзапроса можно использовать написание обобщенного табличного выражения (common table expression, CTE) с помощью предложения WITH. Преимущество предложения WITH заключается в том, что подзапрос именуется заранее, в результате чего код получается более чистым, а также появляется возможность ссылаться на подзапрос несколько раз.

```
WITH o AS (SELECT * FROM owner
           WHERE type = 'public')

SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM o JOIN waterfall w
     ON o.id = w.owner_id;
```

Предложение WITH поддерживается в MySQL 8.0+ (2018 и более поздние версии), PostgreSQL, Oracle, SQL Server и SQLite. В разделе «Обобщенные табличные выражения» главы 9 приведены дополнительные примеры использования этой техники.

Зачем использовать подзапрос в предложении FROM

Основное преимущество использования подзапросов заключается в том, что можно превратить большую задачу в более мелкие. Приведу два примера.

- *Пример 1. Несколько этапов для достижения результатов.* Допустим, вы хотите найти среднее количество остановок в туре. Сначала нужно будет найти количество остановок в каждом туре, а затем усреднить результаты.

  Количество остановок в каждом туре можно найти с помощью этого запроса:

```
SELECT name, MAX(stop) as num_stops
FROM tour
GROUP BY name;

name       num_stops
---------  ----------
M-28               11
Munising            6
US-2               14
```

  Затем можно превратить этот запрос в подзапрос и написать вокруг него еще один запрос для нахождения среднего значения:

```
SELECT AVG(num_stops) FROM
(SELECT name, MAX(stop) as num_stops
FROM tour
GROUP BY name) tour_stops;

AVG(num_stops)
-----------------
10.3333333333333
```

- *Пример 2. Таблица в предложении FROM слишком велика.* Первоначальная цель состояла в том, чтобы перечислить все водопады, находящиеся в государственной собствен-

ности. На самом деле это можно сделать без подзапроса и вместо этого использовать `JOIN`:

```
SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM   owner o
JOIN   waterfall w ON o.id = w.owner_id
WHERE  o.type = 'public';

waterfall_name   owner_name
--------------   --------------
Little Miners    Pictured Rocks
Miners Falls     Pictured Rocks
Munising Falls   Pictured Rocks
Wagner Falls     MI DNR
```

Предположим, что выполнение запроса занимает очень много времени. Это может произойти при соединении больших таблиц (например, десятки миллионов строк). Есть несколько способов переписать запрос, чтобы ускорить его выполнение, и одним из них является использование подзапроса.

Поскольку нас интересуют только владельцы типа `public`, мы можем сначала написать подзапрос, который отфильтрует всех частных владельцев. Затем уменьшенная таблица владельцев будет объединена с таблицей `waterfall`, что займет меньше времени и даст те же результаты:

```
SELECT w.name AS waterfall_name,
       o.name AS owner_name
FROM   (SELECT * FROM owner
       WHERE type = 'public') o
       JOIN waterfall w ON o.id = w.owner_id;

waterfall_name   owner_name
--------------   --------------
Little Miners    Pictured Rocks
Miners Falls     Pictured Rocks
Munising Falls   Pictured Rocks
Wagner Falls     MI DNR
```

Это лишь два из множества примеров того, как с помощью подзапросов разбивать большой запрос на более мелкие шаги.

Предложение WHERE

Предложение `WHERE` позволяет ограничить результаты запроса только интересующими нас строками, или, проще говоря, служит для фильтрации данных. В редких случаях требуется отобразить все строки таблицы, чаще речь о строках, соответствующих определенным критериям.

При исследовании таблицы, содержащей миллионы строк, никогда не следует выполнять команду SELECT * FROM my_table;, поскольку ее выполнение займет неоправданно много времени.

Вместо этого целесообразно отфильтровать данные. Сделать это можно с помощью двух распространенных способов.

- Фильтрация по столбцу в предложении WHERE.

  Еще лучше отфильтровать по уже проиндексированному столбцу, чтобы еще больше ускорить поиск.

  ```
  SELECT *
  FROM my_table
  WHERE year_id = 2021;
  ```

- Покажите несколько первых строк данных с помощью предложения LIMIT (или WHERE ROWNUM <= 10 в Oracle или SELECT TOP 10 * в SQL Server).

  ```
  SELECT *
  FROM my_table
  LIMIT 10;
  ```

Показанный ниже запрос находит информацию обо всех водопадах, в названии которых нет слова `Falls`. Более подробно о ключевом слове `LIKE` можно узнать в главе 7.

```
SELECT id, name
FROM waterfall
WHERE name NOT LIKE '%Falls%';

id     name
-----  ----------------
    7  Little Miners
   14  Rapid River Fls
```

Выделенный жирным шрифтом раздел часто называют условным оператором или предикатом. Предикат выполняет логическое сравнение для каждой строки данных, результатом которого является TRUE/FALSE/UNKNOWN.

Таблица waterfall содержит 16 строк. Для каждой строки проверяется, содержит ли она название водопада *Falls*. Если нет, то предикат name NOT LIKE '%Falls%' имеет значение TRUE и строка возвращается в результаты, как это было для двух предыдущих строк.

Множественные предикаты

Можно также объединять несколько предикатов с помощью операции типа AND или OR. В примере ниже показана информация о водопадах, в названии которых отсутствует слово *Falls* и которые к тому же не имеют владельца:

```
SELECT id, name
FROM waterfall
WHERE name NOT LIKE '%Falls%'
      AND owner_id IS NULL;

id     name
-----  ----------------
   14  Rapid River Fls
```

Более подробную информацию об операциях можно найти в разделе «Операции» главы 7.

Фильтрация по подзапросам

Подзапрос — это запрос, вложенный в другой запрос, и обычно он находится в предложении WHERE. В примере ниже извлекается информация об общедоступных водопадах, расположенных в округе Alger:

```
SELECT w.name
FROM   waterfall w
```

```
WHERE  w.open_to_public = 'y'
       AND w.county_id IN (
           SELECT c.id FROM county c
           WHERE c.name = 'Alger');

name
---------------
Munising Falls
Tannery Falls
Alger Falls
...
```

В отличие от подзапросов в предложениях SELECT или FROM, подзапросы в предложении WHERE не требуют псевдонима. Более того, при добавлении псевдонима будет выдана ошибка.

Зачем использовать подзапрос в предложении WHERE

Первоначальной целью запроса был поиск информации об общедоступных водопадах, расположенных в округе Alger. Если бы вы писали этот запрос с нуля, то, скорее всего, начали бы со следующего:

```
SELECT w.name
FROM   waterfall w
WHERE  w.open_to_public = 'y';
```

На текущий момент у вас есть данные обо всех общедоступных водопадах. Осталось найти информацию о тех, которые находятся конкретно в округе Alger. Вы знаете, что в таблице `waterfall` нет столбца с названием округа, но в таблице округов он есть.

У вас есть два варианта добавления названия округа в результаты. Можно либо написать подзапрос в предложении `WHERE`, который специально извлекает информацию об округе Alger, либо соединить таблицы `waterfall` и `county`:

```
-- Подзапрос в предложении WHERE
SELECT w.name
FROM   waterfall w
```

```
WHERE  w.open_to_public = 'y'
       AND w.county_id IN (
           SELECT c.id FROM county c
           WHERE c.name = 'Alger');
```

или:

```
-- в предложении JOIN
SELECT w.name
FROM   waterfall w INNER JOIN county c
       ON w.county_id = c.id
WHERE  w.open_to_public = 'y'
       AND c.name = 'Alger';

name
---------------
Munising Falls
Tannery Falls
Alger Falls
...
```

Эти два запроса дают одинаковые результаты. Преимущество первого подхода заключается в том, что подзапросы зачастую проще понять, чем соединение. Преимущество второго подхода состоит в том, что соединение обычно выполняется быстрее, чем подзапросы.

РАБОТА ▸ ОПТИМИЗАЦИЯ

При написании SQL-кода выполнить одно и то же действие часто можно несколькими способами.

Вашим главным приоритетом должно быть написание *работающего* кода. Пусть он долго выполняется или выглядит некрасиво. Это неважно, поскольку он работает!

На следующем этапе, если у вас есть время, вы можете *оптимизировать* код, улучшив производительность. Возможно, вы перепишете его с помощью JOIN, сделаете более читабельным с помощью отступов и заглавных букв и т. д.

Не зацикливайтесь на создании наиболее оптимизированного кода заранее, а лучше напишите код, который работает. Написание элегантного кода приходит с опытом.

Другие способы фильтрации данных

Предложение `WHERE` — не единственное место в операторе `SELECT`, где можно фильтровать строки данных.

- Предложение `FROM`: при соединении таблиц в предложении `ON` указывается, как они должны быть связаны между собой. Здесь можно внести условия по ограничению строк данных, возвращаемых запросом. Более подробную информацию см. в разделе «Соединение таблиц» главы 9.
- Предложение `HAVING`: если в операторе `SELECT` есть агрегаты, то в предложении `HAVING` указывается, как их фильтровать. Более подробную информацию см. в разделе «Предложение HAVING» далее в текущей главе.
- Предложение `LIMIT`: чтобы вывести на экран определенное количество строк, можно использовать предложение `LIMIT`. В Oracle это делается с помощью `WHERE ROWNUM`, а в SQL Server — с помощью `SELECT TOP`. Более подробную информацию см. в разделе «Предложение LIMIT» далее в текущей главе.

Предложение GROUP BY

Цель предложения `GROUP BY` — собрать строки в группы и обобщить строки внутри групп каким-либо образом, в конечном итоге возвращая только одну строку на группу. Иногда это называют разрезанием строк на группы и свертыванием строк в каждой группе.

Этот запрос подсчитывает количество водопадов на каждом из маршрутов:

```
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
FROM     waterfall w INNER JOIN tour t
         ON w.id = t.stop
GROUP BY t.name;
```

```
tour_name num_waterfalls
----------  --------------
M-28                     6
Munising                 6
US-2                     4
```

Здесь следует сосредоточиться на двух составляющих:

- *сбор строк*, который задается в предложении `GROUP BY`;
- *суммирование строк* внутри групп, которое задается в предложении `SELECT`.

Этап 1. Сбор строк

В предложении `GROUP BY`:

```
GROUP BY t.name
```

мы указываем, что хотели бы просмотреть все строки данных и объединить в одну группу водопады из тура `M-28`, в другую — все водопады из тура `Munising` и т. д. Скрытым образом данные группируются так:

```
tour_name   waterfall_name
----------  ---------------
M-28        Munising Falls
M-28        Alger Falls
M-28        Scott Falls
M-28        Canyon Falls
M-28        Agate Falls
M-28        Bond Falls

Munising    Munising Falls
Munising    Tannery Falls
Munising    Alger Falls
Munising    Wagner Falls
Munising    Horseshoe Falls
Munising    Miners Falls

US-2        Bond Falls
US-2        Fumee Falls
US-2        Kakabika Falls
US-2        Rapid River Fls
```

Этап 2. Суммирование строк

В предложении SELECT:

```
SELECT t.name AS tour_name,
       COUNT(*) AS num_waterfalls
```

мы указываем, что для каждой группы или каждого тура хотим подсчитать количество строк данных в группе. Поскольку каждая строка представляет собой водопад, наше действие приведет к подсчету общего количества водопадов в каждом туре.

Приведенная здесь функция COUNT() более формально известна как *агрегатная*, или функция, которая суммирует множество строк данных в одно значение. Дополнительные агрегатные функции можно найти в разделе «Агрегатные функции» главы 7.

В данном примере COUNT(*) возвращает количество водопадов в каждом туре. Однако это происходит только потому, что каждая строка данных в таблицах waterfall и tour представляет собой один водопад.

Если один водопад был указан в нескольких строках, то COUNT(*) даст большее значение, чем ожидалось. В этом случае для поиска уникальных водопадов можно использовать COUNT(DISTINCT waterfall_name). Более подробную информацию можно найти в пункте «COUNT и DISTINCT» выше в текущей главе.

Основной вывод заключается в том, что важно вручную дважды перепроверять результаты работы агрегатной функции, чтобы убедиться, что она суммирует данные именно так, как вы задумали.

Теперь, когда группы созданы с помощью предложения GROUP BY, агрегатная функция будет применена к каждой группе один раз:

```
tour_name   COUNT(*)
----------  ---------
```

```
M-28                6
M-28
M-28
M-28
M-28
M-28

Munising            6
Munising
Munising
Munising
Munising
Munising

US-2                4
US-2
US-2
US-2
```

Все столбцы, к которым не была применена агрегатная функция, а в данном случае это столбец `tour_name`, теперь сворачиваются в одно значение:

```
tour_name   COUNT(*)
----------  ---------
M-28                6
Munising            6
US-2                4
```

Такое сворачивание множества детальных строк в одну агрегированную строку означает, что при использовании предложения GROUP BY в предложении SELECT должны содержаться только:

- все столбцы, перечисленные в предложении GROUP BY: t.name
- агрегации: COUNT(*)

```
SELECT t.name AS tour_name,
       COUNT(*) AS num_waterfalls
...
GROUP BY t.name;
```

Невыполнение этого требования может привести к появлению сообщения об ошибке или возврату неверных значений.

GROUP BY НА ПРАКТИКЕ

При использовании GROUP BY вы должны определить:

1) какие столбцы хотите использовать для разделения или группировки данных (например, название тура);
2) как вы хотите обобщить данные в каждой группе (например, подсчитать количество водопадов в каждом туре).

После этого:

1) в предложении SELECT перечислите столбцы, по которым нужно сгруппировать данные (например, название тура), и агрегат (-ы), который (-е) нужно вычислить в каждой группе (например, количество водопадов);
2) в предложении GROUP BY перечислите все столбцы, которые не являются агрегатами (например, название тура).

Более сложные ситуации группировки, включая ROLLUP, CUBE и GROUPING SETS, рассматриваются в разделе «Группировка и агрегирование» главы 8.

Предложение HAVING

Предложение HAVING накладывает ограничения на строки, возвращаемые в результате выполнения запроса GROUP BY. Другими словами, оно позволяет фильтровать результаты после применения GROUP BY.

Предложение HAVING всегда следует непосредственно за предложением GROUP BY. Без GROUP BY не может быть HAVING.

Это запрос, который перечисляет количество водопадов в каждом туре, используя предложение GROUP BY:

```
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
```

```
FROM     waterfall w INNER JOIN tour t
         ON w.id = t.stop
GROUP BY t.name;

tour_name   num_waterfalls
----------  --------------
M-28                     6
Munising                 6
US-2                     4
```

Допустим, мы хотим перечислить только те туры, в которых ровно шесть остановок. Для этого необходимо добавить предложение HAVING после предложения GROUP BY:

```
SELECT   t.name AS tour_name,
         COUNT(*) AS num_waterfalls
FROM     waterfall w INNER JOIN tour t
         ON w.id = t.stop
GROUP BY t.name
HAVING COUNT(*) = 6;


tour_name   num_waterfalls
----------  --------------
M-28                     6
Munising                 6
```

СРАВНЕНИЕ WHERE С HAVING

Целью обоих предложений является фильтрация данных. Если вы пытаетесь:

- фильтровать по определенным столбцам, то пропишите условия в предложении WHERE;
- фильтровать по агрегатам, то записывайте условия в предложение HAVING.

Содержимое предложений WHERE и HAVING нельзя поменять местами:

- никогда не помещайте условие с агрегированием в предложение WHERE. Это приведет к ошибке;
- никогда не помещайте в предложение HAVING условия, не связанные с агрегированием. Такие условия гораздо эффективнее вычисляются в предложении WHERE.

Вы заметите, что предложение HAVING относится к агрегированию COUNT(*):

```
SELECT COUNT(*) AS num_waterfalls
...
HAVING COUNT(*) = 6;
```

а не к псевдониму:

```
# код не будет выполняться
SELECT COUNT(*) AS num_waterfalls
...
HAVING num_waterfalls = 6;
```

Причина этого заключается в порядке выполнения предложений. Предложение SELECT записывается перед предложением HAVING. Однако на самом деле предложение SELECT выполняется *после* предложения HAVING.

Это означает, что псевдоним num_waterfalls в предложении SELECT не существует на момент выполнения предложения HAVING. Вместо этого в предложении HAVING должна быть ссылка на необработанную агрегацию COUNT(*).

Исключение составляют MySQL и SQLite, которые допускают использование псевдонимов (num_waterfalls) в предложении HAVING.

Предложение ORDER BY

Предложение ORDER BY используется для указания порядка сортировки результатов запроса.

Этот запрос возвращает список владельцев и водопадов без какой-либо сортировки:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,
       w.name AS waterfall_name
FROM   waterfall w
       LEFT JOIN owner o ON w.owner_id = o.id;
```

```
owner             waterfall_name
----------------  ---------------
Pictured Rocks    Munising Falls
Michigan Nature   Tannery Falls
AF LLC            Alger Falls
MI DNR            Wagner Falls
Unknown           Horseshoe Falls
...
```

ФУНКЦИЯ COALESCE

Функция COALESCE заменяет все NULL-значения в столбце на другое значение. В данном случае она превратила NULL-значения в столбце `o.name` в текст `'Unknown'`.

Если бы здесь не использовалась функция COALESCE, то все водопады, не имеющие владельцев, были бы исключены из результатов. Вместо этого они теперь они помечены как имеющие владельца `'Unknown'` и могут быть отсортированы и добавлены в результаты.

Более подробную информацию можно найти в главе 7.

Этот запрос возвращает тот же список, но сначала отсортированный в алфавитном порядке по владельцу, а затем по водопаду:

```
SELECT   COALESCE(o.name, 'Unknown') AS owner,
         w.name AS waterfall_name
FROM     waterfall w
         LEFT JOIN owner o ON w.owner_id = o.id
ORDER BY owner, waterfall_name;
```

```
owner             waterfall_name
----------------  ---------------
AF LLC            Alger Falls
MI DNR            Wagner Falls
Michigan Nature   Tannery Falls
Michigan Nature   Twin Falls #1
Michigan Nature   Twin Falls #2
...
```

По умолчанию сортировка производится по возрастанию, то есть текст будет располагаться от A до Z, а числа — от меньшего к большему. Управлять сортировкой каждого столбца можно с помощью ключевых слов ASCENDING и DESCENDING (сокращенно ASC и DESC).

Ниже приведена предыдущая сортировка, но с изменениями, и на этот раз она сортирует имена владельцев в обратном порядке:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,
       w.name AS waterfall_name
...
ORDER BY owner DESC, waterfall_name ASC;

owner             waterfall_name
----------------  ---------------
Unknown           Agate Falls
Unknown           Bond Falls
Unknown           Canyon Falls
...
```

Вы можете сортировать по столбцам и выражениям, которые не входят в список SELECT:

```
SELECT   COALESCE(o.name, 'Unknown') AS owner,
         w.name AS waterfall_name
FROM     waterfall w
         LEFT JOIN owner o ON w.owner_id = o.id
ORDER BY o.id DESC, w.id;

owner             waterfall_name
----------------  ---------------
MI DNR            Wagner Falls
AF LLC            Alger Falls
Michigan Nature   Tannery Falls
...
```

Можно выполнять сортировку и по номеру позиции столбца:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,
       w.name AS waterfall_name
...
ORDER BY 1 DESC, 2 ASC;
```

```
owner             waterfall_name
----------------  ---------------
Unknown           Agate Falls
Unknown           Bond Falls
Unknown           Canyon Falls
...
```

Поскольку строки таблицы SQL не упорядочены, если не включить в запрос предложение ORDER BY, то при каждом выполнении запроса результаты могут отображаться в разном порядке.

> **ORDER BY НЕЛЬЗЯ ИСПОЛЬЗОВАТЬ В ПОДЗАПРОСЕ**
>
> Из шести основных предложений только ORDER BY не может быть использовано в подзапросе. К сожалению, нельзя принудительно упорядочить строки подзапроса.
>
> Во избежание этой проблемы необходимо переписать запрос, используя другую логику, чтобы не использовать предложение ORDER BY в подзапросе, а добавить ORDER BY только во внешний запрос.

Предложение LIMIT

При быстром просмотре таблицы лучше всего возвращать не всю таблицу, а ограниченное количество строк.

MySQL, PostgreSQL и SQLite поддерживают предложение LIMIT. Oracle и SQL Server используют другой синтаксис при той же функциональности:

```
-- MySQL, PostgreSQL и SQLite
SELECT *
FROM owner
LIMIT 3;

-- Oracle
SELECT *
FROM owner
WHERE ROWNUM <= 3;
```

```
-- SQL Server
SELECT TOP 3 *
FROM owner;

id   name              phone          type
---  ----------------  -------------  --------
  1  Pictured Rocks    906.387.2607   public
  2  Michigan Nature   517.655.5655   private
  3  AF LLC                           private
```

Другой способ ограничения количества возвращаемых строк — фильтрация по столбцу в предложении `WHERE`. Фильтрация будет выполняться еще быстрее, если столбец проиндексирован.

ГЛАВА 5

Создание, обновление и удаление

Бóльшая часть этой книги посвящена чтению данных из базы с помощью SQL-запросов. Чтение — одна из четырех основных операций с базой, таких как создание, чтение, обновление и удаление (create, read, update and delete — CRUD).

В этой главе основное внимание уделяется оставшимся трем операциям для БД, таблиц, индексов и представлений. Кроме того, в разделе «Управление транзакциями» рассматривается выполнение нескольких команд как единого целого.

Базы данных

База данных — это место для организованного хранения информации.

В базе можно создавать *объекты базы данных*, которые хранят или ссылаются на данные. К общим объектам относятся таблицы, ограничения, индексы и представления.

Модель данных или *схема* описывает, как эти объекты организованы внутри самой базы.

На рис. 5.1 показана база данных, содержащая множество таблиц. Особенности определения таблиц (например, таблица

Sales содержит пять столбцов) и того, как они связаны друг с другом (например, столбец customer_id в таблице Sales соответствует столбцу customer_id в таблице Customer), — все это является частью *схемы* базы данных.

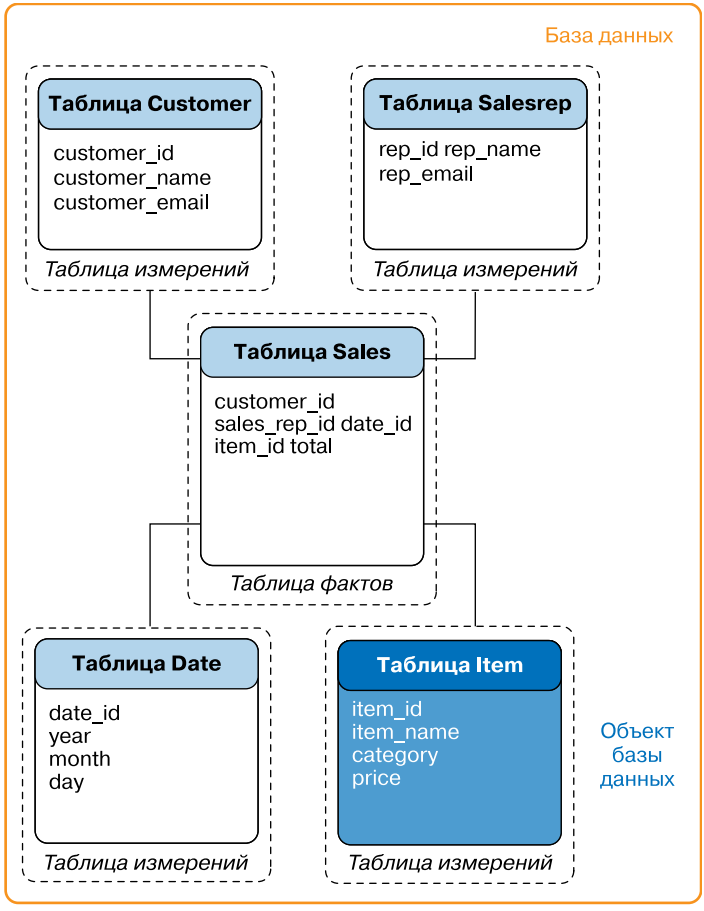


Рис. 5.1. База данных, содержащая схему «Звезда»

Таблицы на рис. 5.1 расположены по *схеме «Звезда»*, которая является одним из основных способов организации таблиц

в базе. Схема содержит *таблицу фактов* в центре и окружена *таблицами измерений* (также известными как *таблицы поиска*). В таблицу фактов записываются совершённые операции (в данном случае продажи), а также идентификаторы дополнительной информации, которая полностью подробно описана в таблицах измерений.

Модель данных в сравнении со схемой

При проектировании базы сначала разрабатывается модель данных, то есть то, как база должна быть организована на высоком уровне. Она может выглядеть, как показано на рис. 5.1, и содержать имена таблиц, способы их связи друг с другом и т. д.

Когда вы будете готовы действовать, вам следует создать *схему*, которая представляет собой реализацию модели данных в базе. В программном обеспечении, в котором вы работаете, указываются таблицы, ограничения, первичные и внешние ключи и т. д.

Определение схемы различается для некоторых РСУБД.

В MySQL схема — то же самое, что база данных, и два этих термина могут использоваться как взаимозаменяемые.

В Oracle схема состоит из объектов базы данных, принадлежащих определенному пользователю, поэтому термины «схема» и «пользователь» используются как взаимозаменяемые.

Отображение имен существующих баз данных

Все объекты баз данных находятся в базах, поэтому сначала нужно посмотреть, какие БД существуют в настоящее время. В табл. 5.1 приведен код для отображения имен всех существующих баз в каждой РСУБД.

Таблица 5.1. Код для отображения имен существующих баз данных

РСУБД	Код
MySQL	`SHOW databases;`
Oracle	`SELECT * FROM global_name;`
PostgreSQL	`\l`
SQL Server	`SELECT name FROM master.sys.databases;`
SQLite	`.database` (или найдем в файловом браузере файлы `.db`)

В большинстве систем базы расположены внутри РСУБД. Однако в SQLite базы хранятся вне системы в виде .db-файлов. Чтобы использовать базу, при запуске SQLite необходимо указать имя .db-файла:

```
> sqlite3 existing_db.db
```

Отображение имени текущей базы данных

Возможно, прежде, чем писать какие-либо запросы, потребуется уточнить, в какой базе данных вы сейчас находитесь. В табл. 5.2 приведен код для отображения имени базы, в которой вы находитесь в данный момент, для каждой РСУБД.

Таблица 5.2. Код для отображения имени текущей базы данных

РСУБД	Код
MySQL	`SELECT database();`
Oracle	`SELECT * FROM global_name;`
PostgreSQL	`SELECT current_database();`
SQL Server	`SELECT db_name();`
SQLite	`.database`

Вероятно, вы заметили, что текущий код базы данных совпадает с существующим кодом баз для Oracle и SQLite.

Экземпляр Oracle может одновременно подключаться только к одной базе, и обычно вы не переключаетесь между базами. В SQLite можно одновременно открывать и работать только с одним файлом базы данных.

Переключение на другую базу данных

Вы можете захотеть использовать данные в другой базе или переключиться на вновь созданную. В табл. 5.3 показан код для перехода к другой базе данных в каждой РСУБД.

Таблица 5.3. Код для переключения на другую базу данных

РСУБД	Код
MySQL, SQL Server	`USE another_db;`
Oracle	Обычно вы не переключаете базы данных (см. предыдущее примечание), но, чтобы переключить пользователей, нужно ввести команду: `connect another_user`
PostgreSQL	`\c another_db`
SQLite	`.open another_db`

Создание базы данных

Если у вас есть привилегия `CREATE`, то вы можете создать новую базу. В противном случае вы сможете работать только с уже существующей. В табл. 5.4 приведен код для создания базы в каждой РСУБД.

Таблица 5.4. Код для создания базы данных

РСУБД	Код
MySQL, Oracle, PostgreSQL, SQL Server	`CREATE DATABASE my_new_db;`
SQLite	`> sqlite3 my_new_db.db`

Oracle: оператор CREATE DATABASE в Oracle требует выполнения некоторых дополнительных действий (касающихся экземпляров, переменных среды и т. д.), которые можно найти в документации Oracle (https://oreil.ly/lXKOF).

SQLite: символ > не является символом, который вы действительно вводите. Он просто означает, что это код командной строки, а не код SQL.

Удаление базы данных

Если у вас есть привилегии `DELETE`, то вы можете удалить базу. В табл. 5.5 приведен код для удаления базы в каждой из РСУБД.

При удалении базы вы потеряете все содержащиеся в ней данные. Отменить эту операцию невозможно, если только не была создана резервная копия. Я не рекомендую выполнять эту команду, если вы не уверены на 100 %, что база вам не нужна.

Таблица 5.5. Код для удаления базы данных

РСУБД	Код
MySQL, Oracle, PostgreSQL, SQL Server	`DROP DATABASE my_new_db;`
SQLite	Удалить `.db`-файл в браузере файлов

Оператор DROP DATABASE в Oracle требует выполнения некоторых дополнительных действий (касающихся монтирования и т. д.), которые можно найти в документации Oracle (https://oreil.ly/v0Tjd).

В некоторых РСУБД невозможно удалить базу, в которой вы находитесь в данный момент. Прежде чем удалять базу, необходимо сначала переключиться на другую, например на базу данных по умолчанию.

- В PostgreSQL БД по умолчанию является `postgres`:

```
\c postgres
DROP DATABASE my_new_db;
```

- В SQL Server БД по умолчанию является `master`:

```
USE master;
go
DROP DATABASE my_new_db;
go
```

Создание таблиц

Таблицы состоят из строк и столбцов и хранят все данные в базе. В SQL к таблицам предъявляется несколько дополнительных требований:

- каждая строка должна быть уникальной;
- все данные в столбце должны быть одного типа (целое число, текст и т. д.).

В SQLite данные в столбце не обязательно должны иметь один и тот же тип данных. Эта система более гибкая в том отношении, что каждое значение имеет тип данных, связанный с ним, а не со всем столбцом.

Для совместимости с другими РСУБД SQLite поддерживает столбцы, имеющие привязки к типу данных. Эти назначения типов являются рекомендуемыми типами данных для столбцов и необязательны.

Создание простой таблицы

Создание таблицы в SQL выполняется в два этапа. Сначала необходимо определить структуру таблицы, а затем загрузить в нее данные.

1. Создание таблицы.

   Код, показанный ниже, создает пустую таблицу `my_simple_table` с тремя столбцами: `id`, `country` и `name`. Все значения в первом столбце (`id`) должны быть целыми числами, а два других столбца (`country`, `name`) могут содержать не менее 2 и не более 15 символов:

   ```
   CREATE TABLE my_simple_table (
      id INTEGER,
      country VARCHAR(2),
      name VARCHAR(15)
   );
   ```

   Дополнительные типы данных, помимо `INTEGER` и `VARCHAR`, перечислены в главе 6.

2. Вставка строки:

   1) вставка одной строки данных. Этот код вставляет одну строку данных в столбцы `id`, `country` и `name`:

      ```
      INSERT INTO my_simple_table (id, country, name)
      VALUES (1, 'US', 'Sam');
      ```

   2) вставка нескольких строк данных.

В табл. 5.6 показано, как в каждой РСУБД вставить в таблицу несколько строк данных, а не одну строку за один раз.

Таблица 5.6. Код для вставки нескольких строк данных

РСУБД	Код

MySQL, PostgreSQL, SQL Server, SQLite	`INSERT INTO my_simple_table`
	`(id, country, name)`
	`VALUES (2, 'US', 'Selena'),`
	`(3, 'CA', 'Shawn'),`
	`(4, 'US', 'Sutton');`

РСУБД	Код

Oracle	`INSERT ALL`
	`  INTO my_simple_table (id, country, name)`
	`  VALUES (2, 'US', 'Selena')`
	`  INTO my_simple_table (id, country, name)`
	`  VALUES (3, 'CA', 'Shawn')`
	`  INTO my_simple_table (id, country, name)`
	`  VALUES (4, 'US', 'Sutton')`
	`SELECT * FROM dual;`

После вставки данных таблица будет выглядеть так:

```
SELECT * FROM my_simple_table;

id   country   name
---  --------  -------
  1  US        Sam
  2  US        Selena
  3  CA        Shawn
  4  US        Sutton
```

При вставке строк данных порядок значений должен точно соответствовать порядку имен столбцов.

Значения в любых столбцах, исключенных из списка столбцов, по умолчанию будут `NULL`, если не указано другое значение по умолчанию.

Для создания таблицы вам необходимы привилегии CREATE. Если при выполнении предыдущего кода вы получаете ошибку, значит, у вас нет на это прав и вам нужно обратиться к администратору базы данных.

Отображение имен существующих таблиц

Прежде чем создавать таблицу, можно посмотреть, нет ли в базе таблицы с таким именем. В табл. 5.7 приведен код для отображения имен таблиц, существующих в базе данных (для каждой РСУБД).

Таблица 5.7. Код для отображения имен существующих таблиц

РСУБД	Код
MySQL	`SHOW tables;`

Oracle	`-- Все таблицы, включая системные`
	`SELECT table_name FROM all_tables;`
	
	`-- Все таблицы, созданные пользователями`
	`SELECT table_name FROM user_tables;`

PostgreSQL	`\dt`
SQL Server	`SELECT table_name` `FROM information_schema.tables;`
SQLite	`.tables`

Создание новой таблицы

В MySQL, PostgreSQL и SQLite при создании таблицы можно проверить наличие существующих таблиц с помощью ключевых слов `IF NOT EXISTS`:

```
CREATE TABLE IF NOT EXISTS my_simple_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15)
);
```

Если имени таблицы нет, то будет создана новая таблица. Если оно есть и `IF NOT EXISTS` не используется, то будет выдано сообщение об ошибке. При использовании `IF NOT EXISTS` новая таблица не создается, и вы сможете избежать сообщения об ошибке.

Заменить существующую таблицу можно двумя способами:

- использовать `DROP TABLE`, чтобы полностью удалить имеющуюся таблицу, а затем создать новую;
- усечь (truncate) существующую таблицу, то есть сохранить схему (структуру) таблицы, но очистить данные внутри нее. Это можно сделать, используя `DELETE FROM` для удаления данных из таблицы.

Создание таблицы с ограничениями

Ограничение (constraint) — это правило, определяющее, какие данные могут быть вставлены в таблицу. Этот код создает две таблицы и несколько ограничений (выделены жирным шрифтом):

```
CREATE TABLE another_table (
   country VARCHAR(2) NOT NULL,
   name VARCHAR(15) NOT NULL,
   description VARCHAR(50),
   CONSTRAINT pk_another_table
      PRIMARY KEY (country, name)
);

CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2) DEFAULT 'CA'
      CONSTRAINT chk_country
      CHECK (country IN ('CA','US')),
   name VARCHAR(15),
   cap_name VARCHAR(15),
   CONSTRAINT pk
      PRIMARY KEY (id),
   CONSTRAINT fk1
      FOREIGN KEY (country, name)
      REFERENCES another_table (country, name),
   CONSTRAINT unq_country_name
      UNIQUE (country, name),
   CONSTRAINT chk_upper_name
      CHECK (cap_name = UPPER(name))
);
```

С помощью ключевого слова `CONSTRAINT` можно дать имя ограничению, чтобы его можно было использовать в последующем. Это ключевое слово является необязательным. Не следует использовать одно и то же имя для столбца и ограничения.

Для быстрого доступа к разделам ограничений используйте следующие ключевые слова: `NOT NULL`, `DEFAULT`, `CHECK`, `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`.

Ограничение: недопущение значений NULL в столбце с NOT NULL

В таблице SQL ячейки, не имеющие значения, заменяются значением NULL. Для каждого столбца можно указать, разрешены ли значения NULL:

```
CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2) NULL,
   name VARCHAR(15)
);
```

Ограничение NOT NULL для столбца id означает, что в столбце не допускаются значения NULL. Другими словами, в столбец нельзя вставить отсутствующие значения, иначе будет выдано сообщение об ошибке.

Ограничение NULL для столбца country означает, что в столбце допускаются значения NULL. Если при вставке данных в таблицу исключить столбец country, то значение вставляться не будет, а ячейка будет заменена значением NULL.

Если не указывать NULL или NOT NULL, то для столбца name по умолчанию будет установлено значение NULL, то есть он будет допускать значения NULL.

Ограничение: установка значений по умолчанию в столбце с помощью DEFAULT

При вставке данных в таблицу отсутствующие значения заменяются значением NULL. Чтобы заменить отсутствующие значения другим, можно использовать ограничение DEFAULT. Этот код превращает любое отсутствующее значение страны в 'CA':

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2) DEFAULT 'CA',
   name VARCHAR(15)
);
```

Ограничение: ограничение значений в столбце с помощью CHECK

Ограничить допустимые значения в столбце можно с помощью ограничения `CHECK`. В приведенном ниже коде в столбце `country` разрешены только значения `'CA'` и `' US'`.

Ключевое слово `CHECK` можно поместить сразу после имени столбца и типа данных:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2) CHECK
       (country IN ('CA', 'US')),
   name VARCHAR(15)
);
```

Или можно поместить ключевое слово `CHECK` после всех имен столбцов и типов данных:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15),
   CHECK (country IN ('CA','US'))
);
```

Можно также включить логику, проверяющую несколько столбцов:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15),
   CONSTRAINT chk_id_country
   CHECK (id > 100 AND country IN ('CA','US'))
);
```

Ограничение: требование уникальности значений в столбце с помощью UNIQUE

Вы можете потребовать, чтобы значения столбца были уникальными, используя ограничение `UNIQUE`.

Ключевое слово UNIQUE можно поместить сразу после имени столбца и типа данных:

```
CREATE TABLE my_table (
   id INTEGER UNIQUE,
   country VARCHAR(2),
   name VARCHAR(15)
);
```

Или можно поместить ключевое слово UNIQUE после всех имен столбцов и типов данных:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15),
   UNIQUE (id)
);
```

Кроме того, можно включить логику, которая заставляет комбинацию нескольких столбцов быть уникальной. Показанный ниже код требует уникальных комбинаций country/name, то есть в одной строке может быть 'CA/Emma', а в другой — 'US/Emma':

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15),
   CONSTRAINT unq_country_name
   UNIQUE (country, name)
);
```

Создание таблицы с первичными и внешними ключами

Первичные и внешние ключи — особые типы ограничений, которые однозначно идентифицируют строки данных.

Установка первичного ключа

Первичный ключ однозначно идентифицирует каждую строку данных в таблице. Он может состоять из одного или нескольких столбцов таблицы. Каждая таблица должна иметь первичный ключ.

Ключевые слова `PRIMARY KEY` можно поместить сразу после имени столбца и типа данных:

```
CREATE TABLE my_table (
   id INTEGER PRIMARY KEY,
   country VARCHAR(2),
   name VARCHAR(15)
);
```

Или можно поместить ключевые слова `PRIMARY KEY` после всех имен столбцов и типов данных:

```
CREATE TABLE my_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15),
   PRIMARY KEY (id)
);
```

Чтобы задать первичный ключ, состоящий из нескольких столбцов (также известный как *составной ключ*), введите такой код:

```
CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2),
   name VARCHAR(15) NOT NULL,
   CONSTRAINT pk_id_name
   PRIMARY KEY (id, name)
);
```

Создавая `PRIMARY KEY`, вы накладываете на столбец (столбцы) следующие ограничения: они не могут содержать значения `NULL` (`NOT NULL`) и значения должны быть уникальными (`UNIQUE`).

РЕКОМЕНДАЦИИ ПО РАБОТЕ С ПЕРВИЧНЫМИ КЛЮЧАМИ

Каждая таблица должна иметь первичный ключ. Это гарантирует, что каждая строка может быть однозначно идентифицирована.

Рекомендуется, чтобы первичные ключи состояли из столбцов идентификаторов, например `(country_id, name_id)`, а не `(country, name)`. Технически несколько строк могут иметь одну и ту же комбинацию страны и названия. В случае добавления столбцов, содержащих уникальные идентификаторы (101, 102 и т. д.), комбинация `country_id` и `name_id` гарантированно будет уникальной.

Первичные ключи должны быть неизменяемыми. Это позволяет всегда идентифицировать конкретную строку таблицы по одному и тому же первичному ключу.

Установка внешнего ключа

Внешний ключ в таблице ссылается на первичный ключ в другой таблице. Две таблицы могут быть связаны между собой общим столбцом. Таблица может иметь ноль или более внешних ключей.

На рис. 5.2 показана модель данных, состоящая из двух таблиц: таблицы `customers`, имеющей первичный ключ `id`, и таблицы `orders`, имеющей первичный ключ `o_id`. С точки зрения клиентов, столбец `order_id` совпадает со значениями столбца `o_id`, что делает `order_id` внешним ключом, поскольку он ссылается на первичный ключ в другой таблице.

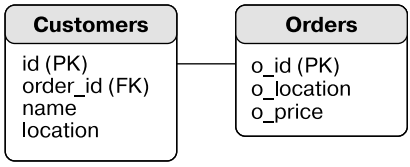


Рис. 5.2. Две таблицы с первичными ключами и внешним ключом

Чтобы задать внешний ключ, выполните следующие действия.

1. Найдите таблицу, на которую планируете ссылаться, и определите ее первичный ключ.

   В данном случае мы будем ссылаться на таблицу `orders`, а именно на колонку `o_id`:

```
CREATE TABLE orders (
   o_id INTEGER PRIMARY KEY,
   o_location VARCHAR(20),
   o_price DECIMAL(6,2)
);
```

2. Создайте таблицу с внешним ключом, который ссылается на первичный ключ в другой таблице.

   В данном случае мы создаем таблицу `customers`, в которой столбец `order_id` ссылается на первичный ключ `o_id` в таблице `orders`:

```
CREATE TABLE customers (
   id INTEGER PRIMARY KEY,
   order_id INTEGER,
   name VARCHAR(15),
   location VARCHAR(20),
   FOREIGN KEY (order_id)
   REFERENCES orders (o_id)
);
```

   Чтобы можно было задать внешний ключ, состоящий из нескольких столбцов, первичный ключ также должен состоять из нескольких столбцов:

```
CREATE TABLE orders (
   o_id INTEGER,
   o_location VARCHAR(20),
   o_price DECIMAL(6,2),
   PRIMARY KEY (o_id, o_location)
);

CREATE TABLE customers (
   id INTEGER PRIMARY KEY,
   order_id INTEGER,
   name VARCHAR(15),
   location VARCHAR(20),
```

```
    CONSTRAINT fk_id_name
    FOREIGN KEY (order_id, location)
    REFERENCES orders (o_id, o_location)
);
```

Внешний ключ (order_id) и первичный ключ, на который он ссылается (o_id), должны иметь один и тот же тип данных.

Создание таблицы с автоматически генерируемым полем

Если вы планируете загружать набор данных без столбца с уникальным идентификатором, то вам потребуется создать столбец, который будет автоматически генерировать уникальный идентификатор. Код, приведенный в табл. 5.8, автоматически генерирует последовательные номера (1, 2, 3 и т. д.) в столбце `u_id`, в каждой РСУБД.

Таблица 5.8. Код для автоматической генерации уникального идентификатора

РСУБД	Код

MySQL	`CREATE TABLE my_table (`
	`  u_id INTEGER PRIMARY KEY AUTO_INCREMENT,`
	`  country VARCHAR(2),`
	`  name VARCHAR(15)`
	`);`

Oracle	`CREATE TABLE my_table (`
	`  u_id INTEGER GENERATED BY DEFAULT`
	`  ON NULL AS IDENTITY,`
	`  country VARCHAR2(2),`
	`  name VARCHAR2(15)`
	`);`

PostgreSQL	`CREATE TABLE my_table (`
	`  u_id SERIAL,`
	`  country VARCHAR(2),`
	`  name VARCHAR(15)`
	`);`

РСУБД	Код

SQL Server	`-- u_id начинается с 1 и увеличивается на 1`
	`CREATE TABLE my_table (`
	`  u_id INTEGER IDENTITY(1,1),`
	`  country VARCHAR(2),`
	`  name VARCHAR(15)`
	`);`

SQLite	`CREATE TABLE my_table (`
	`  u_id INTEGER PRIMARY KEY AUTOINCREMENT,`
	`  country VARCHAR(2),`
	`  name VARCHAR(15)`
	`);`

В Oracle вместо VARCHAR обычно используется VARCHAR2. С точки зрения функциональности они идентичны, но VARCHAR может быть однажды изменен, поэтому безопаснее использовать VARCHAR2.

SQLite не рекомендует использовать AUTOINCREMENT без крайней необходимости, так как он задействует дополнительные вычислительные ресурсы. Код по-прежнему будет выполняться без ошибок.

Вставка результатов запроса в таблицу

Вместо того чтобы вручную вводить значения для вставки в новую таблицу, можно загрузить в новую таблицу данные из существующей таблицы (одной или нескольких).

Рассмотрим пример таблицы:

```
SELECT * FROM my_simple_table;

id   country   name
---  --------  -------
  1  US        Sam
  2  US        Selena
  3  CA        Shawn
  4  US        Sutton
```

Создайте новую таблицу с двумя столбцами:

```
CREATE TABLE new_table_two_columns (
             id INTEGER,
             name VARCHAR(15)
);
```

Вставьте результаты запроса в новую таблицу:

```
INSERT INTO new_table_two_columns
            (id, name)
SELECT id, name
FROM   my_simple_table
WHERE  id < 3;
```

Новая таблица будет выглядеть следующим образом:

```
SELECT * FROM new_table_two_columns;

id   name
---  -------
  1  Sam
  2  Selena
```

Кроме того, можно вставлять значения из существующей таблицы и попутно добавлять или изменять другие значения.

Создайте новую таблицу с четырьмя столбцами:

```
CREATE TABLE new_table_four_columns (
             id INTEGER,
             name VARCHAR(15),
             new_num_column INTEGER,
             new_text_column VARCHAR(30)
);
```

Вставьте результаты запроса в новую таблицу и заполните значения новых столбцов:

```
INSERT INTO new_table_four_columns
(id, name, new_num_column, new_text_column)
SELECT id, name, 2017, 'stargazing'
FROM   my_simple_table
WHERE  id = 2;
```

Вставьте результаты запроса в новую таблицу и измените значение в строке (в данном случае `id`):

```
INSERT INTO new_table_four_columns
       (id, name, new_num_column, new_text_column)
SELECT 3, name, 2017, 'wolves'
FROM   my_simple_table
WHERE  id = 2;
```

Новая таблица будет выглядеть так:

```
SELECT * FROM new_table_four_columns;

id   name     new_num_column   new_text_column
---  -------  ---------------  ----------------
  2  Selena   2017             stargazing
  3  Selena   2017             wolves
```

Вставка данных из текстового файла в таблицу

Возможно, вам понадобится загрузить в таблицу данные из *текстового файла* (данные, хранящиеся в виде обычного текста без специального форматирования). Широко используемым типом текстового файла является `.csv` (значения, разделенные запятыми). Текстовые файлы могут быть открыты в приложениях, не относящихся к РСУБД, в том числе в Excel, Notepad, TextEdit и т. д.

В приведенном ниже коде показано, как загрузить файл `my_data.csv` в таблицу.

Содержимое файла `my_data.csv` таково:

```
unique_id,canada_us,artist_name
5,"CA","Celine"
6,"CA","Michael"
7,"US","Stefani"
8,,"Olivia"
...
```

Создайте таблицу:

```
CREATE TABLE new_table (
   id INTEGER,
   country VARCHAR(2),
   name VARCHAR(15)
);
```

Код, приведенный в табл. 5.9, загружает файл `my_data.csv` в таблицу `new_table` для каждой РСУБД. При загрузке данных можно указать дополнительные сведения о них, например:

- данные разделяются запятыми (`,`);
- текстовые значения берутся в двойные кавычки (`""`);
- каждая новая строка располагается на новой строке (`\n`);
- первая строка текстового файла (содержащая заголовок) должна быть проигнорирована.

Таблица 5.9. Код для вставки данных из файла .csv

РСУБД	Код

MySQL	`LOAD DATA LOCAL`
	`INFILE '<file_path>/my_data.csv'`
	`INTO TABLE new_table`
	`FIELDS TERMINATED BY ','`
	`ENCLOSED BY '"'`
	`LINES TERMINATED BY '\n'`
	`IGNORE 1 ROWS;`

Oracle	Хотя это можно сделать в командной строке с помощью `sqlldr`, лучше загружать данные через графический интерфейс пользователя, например SQL*Loader или SQL Developer

PostgreSQL	`\copy new_table`
	`FROM '<file_path>/my_data.csv'`
	`DELIMITER ',' CSV HEADER`

SQL Server	`BULK INSERT new_table`
	`FROM '<file_path>/my_data.csv'`
	`WITH`
	`(`
	`FORMAT = 'CSV',`
	`FIELDTERMINATOR = ',',`
	`FIELDQUOTE = '"',`
	`ROWTERMINATOR = '\n',`
	`FIRSTROW = 2,`
	`TABLOCK`
	`);`

SQLite	`.mode csv`
	`.import <file_path>/my_data.csv`
	`new_table --skip 1`

После вставки данных таблица будет выглядеть так:

```
SELECT * FROM new_table;

id   country   name
---  --------  --------
  5  CA        Celine
  6  CA        Michael
  7  US        Stefani
  8  NULL      Olivia
...
```

ПРИМЕРЫ ПУТИ К ФАЙЛУ НА РАБОЧЕМ СТОЛЕ

Если файл `my_data.csv` находится на вашем рабочем столе, то вот как будет выглядеть путь к файлу для каждой операционной системы:

- Linux: `/home/my_username/Desktop/my_data.csv`;
- MacOS: `/Users/my_username/Desktop/my_data.csv`;
- Windows: `C:\Users\my_username\Desktop\my_data.csv`.

Если MySQL выдает ошибку, сообщающую о том, что загрузка локальных данных отключена, то включить ее можно, обновив глобальную переменную local_infile, выйдя из системы и перезапустив MySQL:

```
SET GLOBAL local_infile=1;
quit
```

Отсутствующие данные и NULL-значения

Каждая РСУБД по-своему интерпретирует отсутствующие данные из файла `.csv`. Если в нем есть такая строка:

```
8,,"Olivia"
```

вставляемая в SQL-таблицу, то недостающее значение между `8` и `Olivia` будет заменено на:

- значение `NULL` в PostgreSQL и SQL Server;
- пустую строку (`''`) в MySQL и SQLite.

В MySQL и SQLite можно с помощью `\N` в `.csv`-файле представлять значение `NULL` в таблице SQL. Если в файле есть такая строка:

```
8,\N, "Olivia"
```

вставляемая в таблицу MySQL, то при этом `\N` будет заменено на значение `NULL` в таблице.

При вставке в таблицу SQLite `\N` будет жестко закодирован в таблице. Затем можно выполнить код:

```
UPDATE new_table
SET country = NULL
WHERE country = '\N';
```

чтобы заменить в таблице заполнители `\N` на значения `NULL`.

Изменение таблиц

В этом разделе рассматриваются способы изменения имени таблицы, столбцов, ограничений и данных в таблице.

Для изменения таблицы необходимы привилегии ALTER. Если при выполнении кода, приведенного в этом разделе, вы получаете ошибку, значит, у вас нет на это прав и вам необходимо обратиться к администратору базы данных.

Переименование таблицы или столбца

После создания таблицы ее можно переименовать, как и ее столбцы.

При изменении таблицы она будет изменена навсегда. Отменить изменения невозможно, если только не была создана резервная копия. Дважды проверьте свои операторы, прежде чем выполнять их.

Переименование таблицы

В коде, приведенном в табл. 5.10, показано, как переименовать таблицу в каждой РСУБД.

Таблица 5.10. Код для переименования таблицы

РСУБД	Код
MySQL, Oracle, PostgreSQL, SQLite	`ALTER TABLE old_table_name RENAME TO new_table_name;`
SQL Server	`EXEC sp_rename 'old_table_name', 'new_table_name';`

Переименование столбца

В коде, приведенном в табл. 5.11, показано, как переименовать столбец в каждой РСУБД.

Таблица 5.11. Код для переименования столбца

РСУБД	Код
MySQL, Oracle, PostgreSQL, SQLite	`ALTER TABLE my_table RENAME COLUMN old_column_name TO new_column_name;`
SQL Server	`EXEC sp_rename 'my_table.old_column_name', 'new_column_name', 'COLUMN';`

Отображение, добавление и удаление столбцов

После создания таблицы можно просматривать, добавлять и удалять столбцы из таблицы.

Отображение столбцов таблицы

В коде, приведенном в табл. 5.12, показано, как отобразить столбцы таблицы в каждой из РСУБД.

Таблица 5.12. Код для отображения столбцов таблицы

РСУБД	Код
MySQL, Oracle	`DESCRIBE my_table;`
PostgreSQL	`\d my_table`

SQL Server	`SELECT column_name`
	`FROM information_schema.columns`
	`WHERE table_name = 'my_table';`

SQLite	`pragma table_info(my_table);`

Добавление столбца в таблицу

В коде, приведенном в табл. 5.13, показано, как добавить столбец в таблицу в каждой из РСУБД.

Таблица 5.13. Код для добавления столбца в таблицу

РСУБД	Код

MySQL, PostgreSQL	`ALTER TABLE my_table`
	`  ADD new_num_column INTEGER,`
	`  ADD new_text_column VARCHAR(30);`

Oracle	`ALTER TABLE my_table ADD (`
	`  new_num_column INTEGER,`
	`  new_text_column VARCHAR(30));`

SQL Server	`ALTER TABLE my_table`
	`  ADD new_num_column INTEGER,`
	`  new_text_column VARCHAR(30);`

SQLite	`ALTER TABLE my_table`
	`  ADD new_num_column INTEGER;`
	`ALTER TABLE my_table`
	`  ADD new_text_column VARCHAR(30);`

Удаление столбца из таблицы

В коде, приведенном в табл. 5.14, показано, как удалить столбец из таблицы в каждой из РСУБД.

Если на столбец наложены какие-либо ограничения, то перед его удалением необходимо их удалить.

Таблица 5.14. Код для удаления столбца из таблицы

РСУБД	Код

MySQL, PostgreSQL	`ALTER TABLE my_table`
	`  DROP COLUMN new_num_column,`
	`  DROP COLUMN new_text_column;`

Oracle	`ALTER TABLE my_table`
	`  DROP COLUMN new_num_column;`
	`ALTER TABLE my_table`
	`  DROP COLUMN new_text_column;`

SQL Server	`ALTER TABLE my_table`
	`  DROP COLUMN new_num_column,`
	`  new_text_column;`

SQLite	См. действия по внесению изменений в SQLite в ручном режиме

ИЗМЕНЕНИЯ В SQLITE В РУЧНОМ РЕЖИМЕ

SQLite не поддерживает некоторые изменения таблиц, такие как удаление столбцов или добавление/изменение/удаление ограничений.

В качестве обходного пути вы можете либо использовать графический интерфейс пользователя, чтобы создать код, с помощью которого можно изменить таблицу, либо вручную создать новую таблицу и скопировать в нее данные (см. следующие этапы).

1. Создайте новую таблицу с нужными столбцами и ограничениями.

```
CREATE TABLE my_table_2 (
   id INTEGER NOT NULL,
```

```
    country VARCHAR(2),
    name VARCHAR(30)
);
```

2. Скопируйте данные из старой таблицы в новую.

```
INSERT INTO my_table_2
SELECT id, country, name
FROM my_table;
```

3. Убедитесь, что данные находятся в новой таблице.

```
SELECT * FROM my_table_2;
```

4. Удалите старую таблицу.

```
DROP TABLE my_table;
```

5. Переименуйте новую таблицу.

```
ALTER TABLE my_table_2 RENAME TO my_table;
```

Отображение, добавление и удаление строк

После того как таблица будет создана, можно ее просматривать, добавлять в нее строки и удалять их.

Отображение строк таблицы

Чтобы отобразить на экране строки таблицы, достаточно написать оператор `SELECT`:

```
SELECT * FROM my_table;
```

Добавление строк в таблицу

Для добавления строк данных в таблицу используйте оператор `INSERT INTO`:

```
INSERT INTO my_table
   (id, country, name)
VALUES (9, 'US', 'Charlie');
```

Удаление строк из таблицы

Для удаления строк данных из таблицы используйте оператор `DELETE FROM`:

```
DELETE FROM my_table
WHERE id = 9;
```

Опустите предложение `WHERE`, чтобы удалить все строки из таблицы:

```
DELETE FROM my_table;
```

Удаление строк из таблицы также называется *усечением* (truncating), при котором все данные в таблице удаляются, но ее определение не меняется. Таким образом, хоть имена столбцов и ограничения таблицы по-прежнему существуют, теперь она пуста.

Чтобы полностью избавиться от таблицы, можно выполнить оператор `DROP`.

Отображение, добавление, изменение и удаление ограничений

Ограничение — это правило, определяющее, какие данные могут быть вставлены в таблицу. Более подробно о различных типах ограничений можно прочитать ранее в этой главе, в подразделе «Создание таблицы с ограничениями».

Отображение ограничений таблицы

В коде, приведенном в табл. 5.15, показано, как отобразить ограничения таблицы в каждой из РСУБД.

Oracle хранит имена таблиц и столбцов заглавными буквами, если только имя столбца не взято в двойные кавычки. При обращении к имени таблицы или столбца в операторе SQL необходимо писать это имя заглавными буквами (MY_TABLE).

Таблица 5.15. Код для отображения ограничений таблицы

РСУБД	Код
MySQL	`SHOW CREATE TABLE my_table;`

Oracle	`SELECT *`
	`FROM user_cons_columns`
	`WHERE table_name = 'MY_TABLE';`

PostgreSQL	`\d my_table`

SQL Server	`-- Перечислить ограничения (кроме тех, которые заданы по умолчанию)`
	`SELECT table_name,`
	`constraint_name,`
	`constraint_type`
	`FROM information_schema.table_constraints`
	`WHERE table_name = 'my_table';`
	
	`-- Список всех ограничений по умолчанию`
	`SELECT OBJECT_NAME(parent_object_id),`
	`COL_NAME(parent_object_id,`
	`parent_column_id),`
	`definition`
	`FROM sys.default_constraints`
	`WHERE OBJECT_NAME(parent_object_id) =`
	`'my_table';`

SQLite	`.schema my_table`

Добавление ограничения

Начнем с оператора `CREATE TABLE`:

```
CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2) DEFAULT 'CA',
   name VARCHAR(15),
   lower_name VARCHAR(15)
);
```

В коде, приведенном в табл. 5.16, добавляется ограничение, которое гарантирует, что столбец `lower_name` представляет собой версию столбца `name` в нижнем регистре в каждой РСУБД.

Таблица 5.16. Код для добавления ограничения

РСУБД	Код

MySQL, PostgreSQL, SQL Server	`ALTER TABLE my_table`
	`  ADD CONSTRAINT chk_lower_name`
	`  CHECK (lower_name = LOWER(name));`

Oracle	`ALTER TABLE my_table ADD (`
	`  CONSTRAINT chk_lower_name`
	`  CHECK (lower_name = LOWER(name)));`

SQLite	См. действия по внесению изменений в SQLite в ручном режиме

Изменение ограничения

Начнем со следующего оператора `CREATE TABLE`:

```
CREATE TABLE my_table (
   id INTEGER NOT NULL,
   country VARCHAR(2) DEFAULT 'CA',
   name VARCHAR(15),
   lower_name VARCHAR(15)
);
```

В коде, приведенном в табл. 5.17, изменяются следующие ограничения:

- ограничение столбца `country` меняется со значения по умолчанию `'CA'` на значение по умолчанию `NULL`;
- ограничение столбца `name` меняется с разрешения длины до 15 символов на разрешение длины до 30 символов.

Таблица 5.17. Код для изменения ограничений в таблице

РСУБД	Код

MySQL	`ALTER TABLE my_table`
	`  MODIFY country VARCHAR(2) NULL,`
	`  MODIFY name VARCHAR(30);`

Продолжение ⇗

Таблица 5.17 *(продолжение)*

РСУБД	Код

Oracle	`ALTER TABLE my_table MODIFY (`
	`country DEFAULT NULL,`
	`name VARCHAR2(30)`
	`);`

PostgreSQL	`ALTER TABLE my_table`
	`ALTER country DROP DEFAULT,`
	`ALTER name TYPE VARCHAR(30);`

SQL Server	`ALTER TABLE my_table`
	`ALTER COLUMN страна`
	`VARCHAR(2) NULL;`
	`ALTER TABLE my_table`
	`ALTER COLUMN name`
	`VARCHAR(30) NULL;`

SQLite	См. действия по внесению изменений в SQLite в ручном режиме

Удаление ограничения

В коде, приведенном в табл. 5.18, показано, как удалить ограничение из таблицы в каждой из РСУБД.

Таблица 5.18. Код для удаления ограничения из таблицы

РСУБД	Код
MySQL	`ALTER TABLE my_table` `DROP CHECK chk_lower_name;`
Oracle, PostgreSQL, SQL Server	`ALTER TABLE my_table` `DROP CONSTRAINT chk_lower_name;`
SQLite	См. действия по внесению изменений в SQLite в ручном режиме

В MySQL CHECK можно заменить на DEFAULT, INDEX (для ограничения UNIQUE), PRIMARY KEY и FOREIGN KEY. Чтобы удалить ограничение NOT NULL, необходимо изменить ограничение с помощью оператора MODIFY.

Обновление столбца данных

Для обновления значений в столбце данных используется оператор `UPDATE ... SET ...`.

Рассмотрим пример таблицы:

```
SELECT *
FROM my_table;

id   country   name      awards
---  --------  --------  -------
  2  CA        Celine    5
  3  CA        Michael   4
  4  US        Stefani   9
```

Чтобы выполнить предварительный просмотр изменений, которые вы хотите внести, используйте этот код:

```
SELECT LOWER(name)
FROM my_table;

LOWER(name)
------------
celine
michael
stefani
```

Обновить значения в столбце данных можно с помощью этого кода:

```
UPDATE my_table
SET name = LOWER(name);

SELECT * FROM my_table;


id   country   name      awards
---  --------  --------  -------
  2  CA        celine          5
  3  CA        michael         4
  4  US        stefani         9
```

Обновление строк данных

Оператор `UPDATE .. SET .. WHERE ..` используется для обновления значений в строке или нескольких строках данных.

Рассмотрим пример таблицы:

```
SELECT *
FROM my_table;

id   country   name      awards
---  --------  --------  -------
  2  CA        celine          5
  3  CA        michael         4
  4  US        stefani         9
```

Чтобы выполнить предварительный просмотр изменений, которые вы хотите внести, используйте этот код:

```
SELECT awards + 1
FROM my_table
WHERE country = 'CA';

awards + 1
-----------
          6
          5
```

Чтобы обновить значения в нескольких строках данных, введите этот код:

```
UPDATE my_table
SET awards = awards + 1
WHERE country = 'CA';

SELECT * FROM my_table;

id   country   name      awards
---  --------  --------  -------
  2  CA        celine          6
  3  CA        michael         5
  4  US        stefani         9
```

Очень важно добавлять в код предложение WHERE вместе с предложением SET при обновлении определенных строк данных. Если WHERE не будет, то обновится вся таблица.

Обновление строк данных с помощью результатов запроса

Вместо того чтобы обновлять таблицу с помощью вручную введенных значений, можно задать новое значение на основе результатов запроса.

Рассмотрим пример таблицы:

```
SELECT * FROM my_table;

id   country   name      awards
---  --------  --------  -------
  2  CA        celine          5
  3  CA        michael         4
  4  US        stefani         9
```

Чтобы выполнить предварительный просмотр изменений, которые вы хотите внести, используйте этот код:

```
SELECT MIN(awards) FROM my_table;

MIN(awards)
------------
           4
```

Обновить значения на основе запроса можно с помощью такого кода:

```
UPDATE my_table
SET awards = (SELECT MIN(awards) FROM my_table)
WHERE country = 'CA';

SELECT * FROM my_table;
```

```
id   country   name      awards
---  --------  --------  -------
  2  CA        celine          4
  3  CA        michael         4
  4  US        stefani         9
```

MySQL не позволяет обновлять таблицу с помощью запроса к ней же. В предыдущем примере нельзя выполнить UPDATE my_table и FROM my_table. Оператор будет выполнен, если добавить запрос FROM another_table.

Результаты запроса всегда должны возвращать один столбец и либо ноль, либо одну строку. Если возвращается *ноль* строк, то значение устанавливается в `NULL`.

Удаление таблицы

Если таблица больше не нужна, то ее можно удалить с помощью оператора `DROP TABLE`:

```
DROP TABLE my_table;
```

Кроме того, в MySQL, PostgreSQL, SQL Server и SQLite можно добавить `IF EXISTS`, чтобы избежать сообщения об ошибке, если таблица не существует:

```
DROP TABLE IF EXISTS my_table;
```

Если вы удалите таблицу, то все данные в ней будут потеряны. Отменить эту команду невозможно, если только не была создана резервная копия. Я не рекомендую выполнять эту команду, если вы не уверены на 100 %, что таблица вам не нужна.

Удаление таблицы со ссылками на внешний ключ

Если в других таблицах имеются внешние ключи, ссылающиеся на удаляемую таблицу, то вместе с ней необходимо удалить ограничения и этих ключей. В коде, приведенном в табл. 5.19,

показано, как удалить таблицу со ссылками на внешний ключ в каждой из РСУБД.

Таблица 5.19. Код для удаления таблицы, содержащей ссылки на внешний ключ

РСУБД	Код
Oracle	`DROP TABLE my_table CASCADE CONSTRAINTS;`
PostgreSQL	`DROP TABLE my_table CASCADE;`
MySQL, SQL Server, SQLite	Ключевое слово `CASCADE` отсутствует, поэтому перед удалением таблицы необходимо вручную удалить все ограничения внешнего ключа, ссылающиеся на нее

Использовать CASCADE, не зная точно, что именно вы удаляете, опасно. Пожалуйста, будьте осторожны. Я не рекомендую выполнять эту команду, если вы не уверены на 100 %, что ограничения вам не нужны.

Индексы

Представьте, что у вас есть таблица с 10 миллионами строк. Вы пишете запрос к ней, чтобы вернуть значения, которые были зарегистрированы в журнале 1 января 2021 года:

```
SELECT *
FROM my_table
WHERE log_date = '2021-01-01';
```

Выполнение этого запроса займет много времени. Причина в том, что «за кулисами» каждая строка проверяется на то, соответствует ли значение `log_date` дате `'2021-01-01'`. Это 10 миллионов проверок.

Ускорить эту работу можно, создав *индекс* для столбца `log_date`. Это действие выполняется один раз, и результат его будет полезным для всех последующих запросов.

Сравнение книжного указателя и индекса SQL

Чтобы лучше понять, как работает индекс SQL, полезно использовать аналогию. В табл. 5.20 сравнивается книжный указатель с индексом в таблице SQL.

Таблица 5.20. Сравнение книжного указателя с индексом SQL

	Книга	Таблица SQL
Условия	Книга состоит из множества страниц. У каждой страницы есть *атрибуты*, например количество слов, рассматриваемые понятия и т. д.	Таблица состоит из множества строк. У каждой строки есть *столбцы*, например `customer_id`, `log_date` и т. д.
Сценарий	Вы читаете книгу и хотите найти все страницы, посвященные понятию *«подзапросы»*	Вы выполняете запрос к таблице и хотите найти все строки, в которых дата `log_date` равна `'2021-01-01'`
Медленный подход	Можно начать со страницы 1 и пролистать все страницы, чтобы понять, упоминаются ли в книге подзапросы. Это заняло бы много времени	Можно начать со строки 1 и просмотреть каждую строку на предмет того, равна ли дата `log_date` значению `'2021-01-01'`. Это заняло бы много времени
Создание индекса	Все понятия, представленные в книге, были внесены в указатель. Название каждого понятия приведено вместе с номерами страниц, на которых оно рассматривается	Для столбца `log_date` в таблице был создан индекс. Каждая дата `log_date` приведена в индексе вместе с номерами строк, содержащих дату журнала
Быстрый подход	Искать страницы, посвященные *подзапросам*, можно с помощью указателя, чтобы быстро найти номера страниц, на которых упоминаются подзапросы, и перейти к этим страницам	Чтобы найти строки с датой `log_date`, равной `'2021-01-01'`, ваш запрос использует индекс для быстрого поиска номеров строк, содержащих дату, и возвращает эти строки

При выполнении того же запроса в таблице `my_table` (в которой теперь индексируется столбец `log_date`):

```
SELECT *
FROM my_table
WHERE log_date = '2021-01-01';
```

запрос будет выполняться намного быстрее, поскольку вместо проверки каждой строки в таблице он увидит `log_date='2021-01-01'`, обратится к индексу и быстро извлечет все строки, имеющие это значение `log_date`.

Хорошей идеей является создание индекса для нескольких столбцов, по которым часто выполняется фильтрация. Например, столбец первичного ключа, столбец даты и т. д.

Однако не стоит создавать индекс для слишком большого количества столбцов, поскольку он занимает много места. Кроме того, при каждом добавлении или удалении строк индекс необходимо будет перестраивать, что отнимает много времени.

Создание индекса для ускорения запросов

Этот код создает новый индекс `my_index` для столбца `log_date` в таблице `my_table`:

```
CREATE INDEX my_index ON my_table (log_date);
```

При создании индекса в Oracle имя столбца необходимо писать в верхнем регистре и брать его в кавычки:

```
CREATE INDEX my_index ON my_table
  ('LOG_DATE');
```

Oracle автоматически создает индекс для столбцов PRIMARY KEY и UNIQUE при создании таблицы.

Создание индексов может занять много времени. Однако это задача, выполняемая один раз, и в долгосрочной перспективе она оправдает себя при выполнении многих более быстрых запросов в будущем.

Можно также создать многостолбцовый или *составной индекс*. В этом коде создается индекс по двум столбцам — `log_date` и `team`:

```
CREATE INDEX my_index ON my_table (log_date, team);
```

Здесь имеет значение порядок следования столбцов. Если написать запрос, который фильтрует по:

- обоим столбцам — индекс ускорит запрос;
- первому столбцу (`log_date`) — индекс ускорит запрос;
- второму столбцу (`team`) — индекс не поможет, так как сначала упорядочивает данные по `log_date`, а затем по столбцу `team`.

Для создания индекса необходимы привилегии CREATE. Если при выполнении предыдущего кода вы получаете ошибку, значит, у вас нет на это прав и вам необходимо обратиться к администратору базы данных.

Удаление индекса

В коде, приведенном в табл. 5.21, показано, как удалить индекс в каждой РСУБД.

Таблица 5.21. Код для удаления индекса

РСУБД	Код
MySQL, SQL Server	`DROP INDEX my_index ON my_table;`
Oracle, PostgreSQL, SQLite	`DROP INDEX my_index;`

Удаление индекса невозможно отменить. Прежде чем удалять индекс, убедитесь на 100 %, что хотите его удалить.

Положительным моментом является то, что потери данных не происходит. Данные в таблице остаются нетронутыми, и индекс всегда можно воссоздать.

Представления

Допустим, у вас есть длинный и сложный SQL-запрос, в котором множество соединений, фильтров, агрегаций и т. д. Его результаты могут быть полезны для вас, и вы захотите обратиться к ним позже.

Это отличная ситуация для создания *представления* (view) или присвоения имени выходным данным запроса. Помните, что они представляют собой одну таблицу, поэтому представление выглядит так же, как и таблица. Разница заключается в том, что оно на самом деле не хранит никаких данных, как таблица, а просто ссылается на них.

Иногда администраторы баз данных создают представления, чтобы ограничить доступ к таблицам. Допустим, имеется таблица customer. Большинство людей должны иметь возможность только читать данные в ней, но не вносить в нее изменения.

Администратор базы может создать представление customer, содержащее данные, идентичные таблице клиентов. Теперь все могут запрашивать представление customer, и только администратор сможет редактировать данные в таблице customer.

Этот код представляет собой сложный запрос, который мы не хотим писать снова и снова:

```
-- Количество водопадов, принадлежащих каждому владельцу
SELECT o.id, o.name,
```

```
       COUNT(w.id) AS num_waterfalls
FROM owner o LEFT JOIN waterfall w
     ON o.id = w.owner_id
GROUP BY o.id, o.name;

id     name              num_waterfalls
-----  ----------------  --------------
    1  Pictured Rocks                 3
    2  Michigan Nature                3
    3  AF LLC                         1
    4  MI DNR                         1
    5  Horseshoe Falls                0
```

Допустим, мы хотим найти среднее количество водопадов для каждого владельца. Это можно сделать, используя либо подзапрос, либо представление:

```
-- Подход с использованием подзапросов
SELECT AVG(num_waterfalls) FROM
(SELECT o.id, o.name,
        COUNT(w.id) AS num_waterfalls
FROM owner o LEFT JOIN waterfall w
     ON o.id = w.owner_id
GROUP BY o.id, o.name) my_subquery;

AVG(num_waterfalls)
--------------------
                 1.6


--- Подход с использованием представления
CREATE VIEW owner_waterfalls_vw AS
SELECT o.id, o.name,
       COUNT(w.id) AS num_waterfalls
FROM owner o LEFT JOIN waterfall w
     ON o.id = w.owner_id
GROUP BY o.id, o.name;

SELECT AVG(num_waterfalls)
FROM owner_waterfalls_vw;

AVG(num_waterfalls)
--------------------
                 1.6
```

Для создания представления необходимы привилегии CREATE. Если при выполнении предыдущего кода вы получаете ошибку, значит, у вас нет соответствующих прав и вам необходимо обратиться к администратору базы данных.

ПОДЗАПРОСЫ В СРАВНЕНИИ С ПРЕДСТАВЛЕНИЯМИ

Как подзапросы, так и представления представляют собой результаты запроса, которые впоследствии могут быть использованы для запросов.

- *Подзапрос* является временным. Он существует только на время выполнения запроса и отлично подходит для однократного применения.
- *Представление* сохраняется. После того как представление будет создано, можно продолжать писать ссылающиеся на него запросы.

Создание представления для сохранения результатов запроса

Используем `CREATE VIEW` для сохранения результатов запроса в виде представления. Затем к этому представлению можно обращаться как к таблице.

Используем этот запрос:

```
SELECT *
FROM my_table
WHERE country = 'US';

id   country   name
---  --------  ------
  1  US        Anna
  2  US        Emily
  3  US        Molly
```

Создаем представление:

```
CREATE VIEW my_view AS
SELECT *
FROM my_table
WHERE country = 'US';
```

Запрашиваем представление:

```
SELECT * FROM my_view;

id    country  name
---   --------  ------
  1   US        Anna
  2   US        Emily
  3   US        Molly
```

Отображение существующих представлений

В коде, приведенном в табл. 5.22, показывается, как отобразить все существующие представления в каждой РСУБД.

Таблица 5.22. Код для отображения существующих представлений

РСУБД	Код
MySQL	`SHOW FULL TABLES` `WHERE table_type = 'VIEW';`
Oracle	`SELECT view_name` `FROM user_views;`

PostgreSQL	`SELECT table_name`
	`FROM information_schema.views`
	`WHERE table_schema NOT IN`
	`('information_schema', 'pg_catalog');`

SQL Server	`SELECT table_name` `FROM information_schema.views;`

SQLite	`SELECT name`
	`FROM sqlite_master`
	`WHERE type = 'view';`

Обновление представления

Обновить представление означает, иначе говоря, перезаписать представление. В коде, приведенном в табл. 5.23, показывается, как обновить представление в каждой РСУБД.

Таблица 5.23. Код для обновления представления

РСУБД	Код

MySQL, Oracle, PostgreSQL	`CREATE OR REPLACE VIEW my_view AS`
	`SELECT *`
	`FROM my_table`
	`WHERE country = 'CA';`

SQL Server	`CREATE OR ALTER VIEW my_view AS`
	`SELECT *`
	`FROM my_table`
	`WHERE country = 'CA';`

SQLite	`DROP VIEW IF EXISTS my_view;`
	`CREATE VIEW my_view AS`
	`SELECT * FROM my_table WHERE country = 'CA';`

Удаление представления

Если представление вам больше не нужно, то его можно удалить с помощью оператора `DROP VIEW`:

```
DROP VIEW my_view;
```

Отменить удаление представления невозможно. Убедитесь на 100 %, что хотите удалить представление, прежде чем удалять его.

Положительный момент — данные не теряются. Они по-прежнему находятся в исходной таблице, и представление всегда можно воссоздать.

Управление транзакциями

Транзакция позволяет более безопасно обновлять базу данных. Она состоит из последовательности операций, которые выполняются как единое целое. Выполняются либо все операции, либо ни одна из них, что также известно как *атомарность*.

Код ниже запускает транзакцию до внесения изменений в таблицы. После выполнения этих операторов никакие обновления не будут постоянно производиться в базе данных до тех пор, пока изменения не будут зафиксированы:

```
START TRANSACTION;

INSERT INTO page_views (user_id, page)
   VALUES (525, 'home');
INSERT INTO page_views (user_id, page)
   VALUES (525, 'contact us');
DELETE FROM new_users WHERE user_id = 525;
UPDATE page_views SET page = 'request info'
   WHERE page = 'contact us';

COMMIT;
```

ПОЧЕМУ БЕЗОПАСНЕЕ ИСПОЛЬЗОВАТЬ ТРАНЗАКЦИЮ

После начала транзакции:

- *все четыре оператора рассматриваются как единое целое.* Представьте, что выполняете первые три оператора, а в это время кто-то другой редактирует базу таким образом, что четвертый оператор не выполняется. Это чревато проблемами, поскольку для правильного обновления базы необходимо, чтобы все четыре оператора выполнялись вместе. Транзакция делает именно это — она требует, чтобы четыре оператора действовали как единое целое, поэтому выполняются либо все они, либо ни один;
- *при необходимости изменения можно отменить.* После запуска транзакции можно запустить каждый из операторов и посмотреть, как они повлияют на таблицы. Если все выглядит правильно, то можно завершить транзакцию и зафиксировать изменения с помощью оператора `COMMIT`. Если что-то покажется неправильным и вы захотите вернуть все, что было до транзакции, то сделать это можно с помощью оператора `ROLLBACK`.

В общем случае, если вы обновляете базу данных, рекомендуется использовать транзакцию.

В следующих подразделах рассматриваются два сценария, в которых использовать транзакцию полезно: один заканчи-

вается оператором COMMIT, применяемым для подтверждения изменений, а второй — оператором ROLLBACK, служащим для отмены изменений.

Двойная проверка изменений перед использованием оператора COMMIT

Представьте, что хотите окончательно удалить несколько строк данных из таблицы, но перед этим хотите дважды проверить, что будут удалены именно те строки, которые нужно. В коде ниже показывается поэтапный процесс использования транзакции в SQL для решения этой задачи.

1. Начните транзакцию.

```
-- MySQL и PostgreSQL
START TRANSACTION;
or
BEGIN;

-- SQL Server и SQLite
BEGIN TRANSACTION;
```

   В Oracle вы, по сути, всегда находитесь в транзакции. Она начинается с момента выполнения первого SQL-оператора. После завершения одной транзакции (с помощью COMMIT или ROLLBACK) при выполнении следующего SQL-оператора начинается другая.

2. Просмотрите таблицу, которую планируете изменить.

   В этот момент вы находитесь в режиме транзакции, значит, в базу данных не будут внесены никакие изменения.

```
SELECT * FROM books;

+------+--------------+
| id   | title        |
+------+--------------+
|    1 | Becoming     |
|    2 | Born a Crime |
|    3 | Bossypants   |
+------+--------------+
```

3. Протестируйте изменение и посмотрите, как оно повлияет на таблицу.

   Вы хотите удалить все названия книг, состоящие из нескольких слов.

   Оператор `SELECT` позволяет просмотреть в таблице все названия книг, состоящие из нескольких слов:

```
SELECT * FROM books WHERE title LIKE '% %';

+------+--------------+
| id   | title        |
+------+--------------+
|    2 | Born a Crime |
+------+--------------+
```

   Оператор `DELETE` использует то же самое предложение `WHERE` для удаления из таблицы названий книг, состоящих из нескольких слов:

```
DELETE FROM books WHERE title LIKE '% %';

SELECT * FROM books;

+------+--------------+
| id   | title        |
+------+--------------+
|    1 | Becoming     |
|    3 | Bossypants   |
+------+--------------+
```

   На данный момент вы все еще находитесь в режиме транзакции, поэтому изменение не стало постоянным.

4. Подтвердите изменения с помощью оператора `COMMIT`.

   Для фиксации изменений используем `COMMIT`. Сделав это, вы выходите из режима транзакций.

```
COMMIT;
```

Отменить (или откатить) транзакцию после ее фиксации невозможно.

Отмена изменений с помощью оператора ROLLBACK

Транзакции особенно полезны для тестирования изменений и их отмены в случае необходимости.

1. Начните транзакцию.

```
-- MySQL и PostgreSQL
START TRANSACTION;
or
BEGIN;

-- SQL Server и SQLite
BEGIN TRANSACTION;
```

   В Oracle вы, по сути, всегда находитесь в транзакции. Она начинается с момента выполнения первого SQL-оператора. После завершения одной транзакции (с помощью `COMMIT` или `ROLLBACK`) при выполнении следующего SQL-оператора начинается другая.

2. Просмотрите таблицу, которую планируете изменить.

   На данный момент вы находитесь в режиме транзакции, то есть никакие изменения в базу данных вноситься не будут.

```
SELECT * FROM books;

+------+--------------+
| id   | title        |
+------+--------------+
| 1    | Becoming     |
| 2    | Born a Crime |
| 3    | Bossypants   |
+------+--------------+
```

3. Протестируйте изменение и посмотрите, как оно повлияет на таблицу.

   Вы хотите удалить все названия книг, состоящие из нескольких слов. Оператор `DELETE` случайно удаляет всю

таблицу (вы забыли о пробеле в `'%%'`). Вы не хотели, чтобы это произошло!

```
DELETE FROM books WHERE title LIKE '%%';

SELECT * FROM books;

+------+--------------+
| id   | title        |
+------+--------------+
```

Хорошо, что в этот момент вы все еще находитесь в режиме транзакции, поэтому изменение не стало постоянным.

4. Отмените изменение с помощью оператора `ROLLBACK`.

   Вместо `COMMIT` выполните `ROLLBACK`. Таблица не будет удалена. Сделав это, вы больше не находитесь в режиме транзакции и можете продолжить работу, используя другие операторы.

```
ROLLBACK;
```

ГЛАВА 6

Типы данных

В таблице SQL каждый столбец может содержать значения только одного типа данных. В этой главе мы поговорим о часто используемых типах данных, а также о том, как и когда их следует применять.

Этот оператор задает три столбца и тип данных для каждого из них: `id` содержит целочисленные значения, `name` — значения, содержащие до 30 символов, а `dt` — значения даты:

```
CREATE TABLE my_table (
   id INT,
   name VARCHAR(30),
   dt DATE
);
```

`INT`, `VARCHAR` и `DATE` — лишь три из множества типов данных в SQL. В табл. 6.1 перечислены четыре категории типов данных, а также широко используемые подкатегории. Синтаксис типов данных сильно различается в зависимости от РСУБД, и об этих различиях мы подробно поговорим в каждом разделе текущей главы.

В табл. 6.2 приведены примеры значений каждого типа данных, чтобы показать, как они представлены в SQL. Эти значения часто называют литералами или константами.

Таблица 6.1. Типы данных в SQL

Числовой	Строковый	Дата и время	Другое
Целое число (`123`) Десятичное число (`1.23`) Число с плавающей запятой (`1.23e10`)	Символ (`'hello'`) Unicode (`'西瓜'`)	Дата (`'2021-12-01'`) Время (`'2:21:00'`) Дата и время (`'2021-12-01 2:21:00'`)	Булево значение (`TRUE`) Бинарные данные (изображения, документы и т. д.)

Таблица 6.2. Литералы в SQL

Категория	Подкатегория	Пример значений

Числовой	Целое число	`123`
		`+123`
		`-123`

	Десятичное число	`123.45`
		`+123.45`
		`-123.45`

	Число с плавающей запятой	`123.45E+23` `123.45e-23`
Строковый	Символ	`'Thank you!'` `'The combo is 39-6-27.'`
	Unicode	`N'Amelie'` `N'♥♥♥'`
Дата и время	Дата	`'2022-10-15'` `'15-OCT-2022'` (Oracle)

	Время	`'10:30:00'`
		`'10:30:00.123456'`
		`'10:30:00 -6:00'`

	Дата и время	`'2022-10-15 10:30:00'` `'15-OCT-2022 10:30:00'` (Oracle)

Категория	Подкатегория	Пример значений
Другое	Булево значение	`TRUE` `FALSE`
	Бинарные данные (примерные значения отображаются в шестнадцатеричном виде)	`X'AB12'` (MySQL, PostgreSQL) `x'AB12'` (MySQL, PostgreSQL) `0xAB12` (MySQL, SQL Server, SQLite)

ЛИТЕРАЛ NULL

Ячейки, не имеющие значения, представляются ключевым словом `NULL` (оно же литерал `NULL`), которое не зависит от регистра (`NULL` = `Null` = `null`).

В таблице часто встречаются нулевые значения, однако сам по себе `null` не является типом данных. Любой числовой, строковый, временной или другой столбец может содержать нулевые значения.

Как выбрать тип данных

При выборе типа данных для столбца важно соблюсти баланс между размером хранилища и гибкостью.

В табл. 6.3 приведено несколько примеров целочисленных типов данных. Обратите внимание, что каждый тип допускает различный диапазон значений и требует разного объема памяти.

Таблица 6.3. Пример целочисленных типов данных

Тип данных	Диапазон допустимых значений	Размер хранения
`INT`	От –2 147 483 648 до 2 147 483 647	4 байта
`SMALLINT`	От –32 768 до 32 767	2 байта
`TINYINT`	От 0 до 255	1 байт

Представьте, что у вас есть столбец данных, содержащий информацию о количестве студентов в аудитории:

```
15
25
50
70
100
```

Этот столбец содержит числовые данные — точнее, целые числа. Для этого столбца можно выбрать любой из трех целочисленных типов данных, приведенных в табл. 6.3.

- *Выбор* `INT`. Если место для хранения данных не является проблемой, то выберите `INT` — этот тип прост и надежен и работает во всех РСУБД.
- *Выбор* `TINYINT`. Поскольку все значения находятся в диапазоне от `0` до `255`, то благодаря типу `TINYINT` вы сможете сэкономить место в памяти.
- *Выбор* `SMALLINT`. Если впоследствии в столбец может быть добавлен большой объем информации о студентах, то выберите `SMALLINT` — этот тип обеспечивает бóльшую гибкость и при этом занимает меньше места, чем `INT`.

Единственно верного ответа нет. Выбор оптимального типа данных для столбца зависит как от объема памяти, так и от требуемой гибкости.

Если вы уже создали таблицу, но хотите изменить тип данных для столбца, это можно сделать, изменив ограничение столбца с помощью оператора ALTER TABLE. Более подробную информацию можно найти в пункте «Изменение ограничения» в главе 5.

Числовые данные

Здесь мы поговорим о числовых значениях и выясним, какова форма их представления в SQL, а затем рассмотрим целые и десятичные типы, а также типы данных с плавающей запятой.

Столбцы с числовыми данными можно вводить в числовые функции, такие как `SUM()` и `ROUND()`, которые рассматриваются в разделе «Числовые функции» главы 7.

Числовые значения

К числовым значениям относятся целые и десятичные числа, а также числа с плавающей запятой.

Целочисленные значения

Числа без десятичной дроби рассматриваются как целые. Знак + является необязательным.

```
123    +123    -123
```

Десятичные значения

Десятичные числа содержат десятичную запятую и хранятся как точные значения. Знак + является необязательным.

```
123.45    +123.45    -123.45
```

Значения с плавающей запятой

Для значений с плавающей запятой используется научная нотация.

```
123.45E+23    123.45e-23
```

Эти значения интерпретируются как $123{,}45 \times 10^{23}$ и $123{,}45 \times 10^{-23}$ соответственно.

В Oracle допускается наличие завершающего символа F, f, D или d для обозначения FLOAT или DOUBLE (более точное значение FLOAT):

```
123F    +123f    -123.45D    123.45d
```

Целочисленные типы данных

Этот код создает целочисленный столбец:

```
CREATE TABLE my_table (
   my_integer_column INT
);

INSERT INTO my_table VALUES
   (25),
   (-525),
   (2500252);

SELECT * FROM my_table;

+------------------+
| my_integer_column |
+------------------+
|                25 |
|              -525 |
|           2500252 |
+------------------+
```

В табл. 6.4 перечислены варианты целочисленных типов данных для каждой РСУБД.

Таблица 6.4. Целочисленные типы данных

РСУБД	Тип данных	Диапазон допустимых значений	Размер хранения
MySQL	`TINYINT`	От –128 до 127, от 0 до 255 (без знака)	1 байт
	`SMALLINT`	От –32 768 до 32 767, от 0 до 65 535 (без знака)	2 байта
	`MEDIUMINT`	От –8 388 608 до 8 388 607, от 0 до 16 777 215 (без знака)	3 байта
	`INT` или `INTEGER`	От –2 147 483 648 до 2 147 483 647, от 0 до 4 294 967 295 (без знака)	4 байта

РСУБД	Тип данных	Диапазон допустимых значений	Размер хранения
	`BIGINT`	От -2^{63} до $2^{63} - 1$, от 0 до 264 – 1 (без знака)	8 байт
Oracle	`NUMBER`	От -10^{125} до $10^{125} - 1$	От 1 до 22 байт
PostgreSQL	`SMALLINT`	От –32 768 до 32 767	2 байта
	`INT` или `INTEGER`	От –2 147 483 648 до 2 147 483 647	4 байта
	`BIGINT`	От -2^{63} до $2^{63} - 1$	8 байт
SQL Server	`TINYINT`	От 0 до 255	1 байт
	`SMALLINT`	От –32 768 до 32 767	2 байта
	`INT` или `INTEGER`	От –2 147 483 648 до 2 147 483 647	4 байта
	`BIGINT`	От -2^{63} до $2^{63} - 1$	8 байт
SQLite	`INTEGER`	От -2^{63} до $2^{63} - 1$ (если больше, то происходит переход к типу данных REAL)	1, 2, 3, 4, 6 или 8 байт

В MySQL допускается использование как диапазонов целых чисел со знаком (положительные и отрицательные целые числа), так и диапазонов чисел без знака (только положительные целые числа). По умолчанию используется диапазон целых чисел со знаком. Чтобы указать беззнаковый диапазон, используйте следующую команду:

```
CREATE TABLE my_table (
   my_integer_column INT UNSIGNED
);
```

В PostgreSQL существует тип данных SERIAL, который создает в столбце автоинкрементное целое число (1, 2, 3 и т. д.). В табл. 6.5 перечислены параметры SERIAL, каждый из которых имеет свой диапазон.

Таблица 6.5. Варианты типа данных SERIAL в PostgreSQL

Тип данных	Диапазон генерируемых значений	Размер хранения
`SMALLSERIAL`	От 1 до 32 767	2 байта
`SERIAL`	От 1 до 2 147 483 647	4 байта
`BIGSERIAL`	От 1 до 9 223 372 036 854 775 807	8 байт

Десятичные типы данных

Десятичные числа также известны как числа *с фиксированной запятой*. Они содержат десятичную запятую и хранятся как точное значение. Денежные данные (например, `799.95`) часто хранятся в виде десятичных чисел. Этот код создает десятичный столбец:

```
CREATE TABLE my_table (
   my_decimal_column DECIMAL(5,2)
);

INSERT INTO my_table VALUES
   (123.45),
   (-123),
   (12.3);

SELECT * FROM my_table;

+------------------+
| my_decimal_column |
+------------------+
|           123.45 |
|          -123.00 |
|            12.30 |
+------------------+
```

При определении типа данных `DECIMAL(5,2)`:

- `5` — максимальное *общее количество* сохраняемых цифр. Эта величина называется *точностью*;
- `2` — количество цифр *справа от десятичной запятой*. Эта величина называется *масштабом*.

В табл. 6.6 перечислены варианты десятичного типа данных для каждой РСУБД.

Таблица 6.6. Десятичные типы данных

РСУБД	Тип данных	Максимально допустимое количество цифр	По умолчанию
MySQL	`DECIMAL` или `NUMERIC`	Всего: 65. После десятичной запятой: 30	`DECIMAL(10,0)`
Oracle	`NUMBER`	Всего: 38. После десятичной запятой: от –84 до 127 (отрицательные значения означают, что это до десятичной запятой)	0 цифр после десятичной запятой
PostgreSQL	`DECIMAL` или `NUMERIC`	До десятичной запятой: 131,072. После десятичной запятой: 16,383	`DECIMAL(30,6)`
SQL Server	`DECIMAL` или `NUMERIC`	Всего: 38. После запятой: 38	`DECIMAL(18,0)`
SQLite	`NUMERIC`	Нет информации	Нет значения по умолчанию

Типы данных с плавающей запятой

Числа с плавающей запятой — это концепция информатики. Когда число имеет много цифр до или после десятичной запятой, то вместо того, чтобы хранить все цифры, числа с плавающей запятой хранят только ограниченное их количество в целях экономии места.

- Число: `1234.56789`.
- Условные обозначения с плавающей запятой: `1.23 x 10^3`.

Можно заметить, что десятичная запятая «уплыла» на несколько знаков влево и вместо полного исходного значения (`1234.56789`) было сохранено *приближенное* (`1.23`). Есть два типа данных с плавающей запятой:

- *число с одинарной точностью:* представлено не менее чем шестью цифрами, полный диапазон — от `1E-38` до `1E+38`;
- *число с двойной точностью:* представлено не менее чем 15 цифрами, полный диапазон — от `1E-308` до `1E+308`.

Этот код создает столбец с плавающей точкой как одинарной точности (`FLOAT`), так и двойной точности (`DOUBLE`):

```
CREATE TABLE my_table (
   my_float_column FLOAT,
   my_double_column DOUBLE
);

INSERT INTO my_table VALUES
   (123.45, 123.45),
   (-12345.6789, -12345.6789),
   (1234567.890123456789, 1234567.890123456789);

SELECT * FROM my_table;

+-----------------+--------------------+
| my_float_column | my_double_column   |
+-----------------+--------------------+
|          123.45 |             123.45 |
|        -12345.7 |        -12345.6789 |
|         1234570 | 1234567.8901234567 |
+-----------------+--------------------+
```

Поскольку в данных с плавающей запятой хранятся приближенные значения, то сравнения и вычисления могут несколько отличаться от ожидаемых.

Если ваши данные всегда будут содержать одинаковое количество десятичных цифр, то для хранения точных значений лучше использовать тип данных с фиксированной запятой, например DECIMAL, а не тип данных с плавающей запятой.

В табл. 6.7 перечислены варианты типов данных с плавающей запятой для каждой РСУБД.

Таблица 6.7. Типы данных с плавающей запятой

РСУБД	Тип данных	Диапазон данных	Размер хранения
MySQL	`FLOAT`	От 0 до 23 бит	4 байта
	`FLOAT`	От 24 до 53 бит	8 байт
	`DOUBLE`	От 0 до 53 бит	8 байт
Oracle	`BINARY_FLOAT`	Нет информации	4 байта
	`BINARY_DOUBLE`	Нет информации	8 байт
PostgreSQL	`REAL`	Нет информации	4 байта
	`DOUBLE PRECISION`	Нет информации	8 байт
SQL Server	`REAL`	Нет информации	4 байта
	`FLOAT`	От 1 до 24 бит	4 байта
	`FLOAT`	От 25 до 53 бит	8 байт
SQLite	`REAL`	Нет информации	8 байт

Тип данных FLOAT в Oracle НЕ является числом с плавающей запятой. Вместо этого FLOAT эквивалентен NUMERIC, который представляет собой десятичное число. Для типа данных с плавающей запятой вместо него следует использовать BINARY_FLOAT или BINARY_DOUBLE.

Биты в сравнении с байтами и цифрами

Один *бит* — наименьшая единица хранения информации. Он может иметь значение 0 или 1.

Один *байт* состоит из восьми бит. Пример байта: 10101010.

Каждый символ представлен байтом. *Цифра* 7 = 00000111 в форме байта.

Строковые данные

В этом разделе мы поговорим о строковых значениях и выясним, какова форма их представления в SQL, а затем подробно рассмотрим символьные и типы данных в Unicode.

Столбцы со строковыми данными могут быть входными данными строковых функций, таких как `LENGTH()` и `REGEXP()` (регулярное выражение), которые рассматриваются в разделе «Строковые функции» главы 7.

Строковые значения

Строковые значения представляют собой последовательности символов, содержащие буквы, цифры и специальные символы.

Основы работы со строками

По стандарту строковые значения берутся в одинарные кавычки:

```
'This is a string.'
```

Когда необходимо вставить одинарную кавычку в строку, нужно использовать две соседние одинарные кавычки:

```
'You''re welcome.'
```

SQL будет рассматривать две такие кавычки как одну внутри строки и возвращать ее:

```
'You're welcome.'
```

Рекомендуется для обозначения строковых значений использовать одинарные кавычки ('), а для идентификаторов (имен таблиц, столбцов и т. д.) — двойные ("").

Альтернативы одинарным кавычкам

Если ваш текст содержит много одинарных кавычек и для обозначения строки необходимо использовать другой символ, то Oracle и PostgreSQL позволяют это сделать.

В Oracle можно начинать строку с символа `Q` или `q`, за которым следует любой символ, затем строка и, наконец, снова тот же символ:

```
Q'[This is a string.]'
q'[This is a string.]'
Q'|This is a string.|'
```

В PostgreSQL можно обрамлять текст двумя знаками доллара и необязательным именем тега:

```
$$This is a string.$$
$mytag$This is a string.$mytag$
```

Управляющие последовательности

MySQL и PostgreSQL поддерживают *управляющие последовательности* (escape-последовательности), или последовательности текста, имеющие специальный смысл. В табл. 6.8 перечислены часто используемые управляющие последовательности.

Таблица 6.8. Часто используемые управляющие последовательности

Последовательность	Описание
`\'`	Одинарная кавычка
`\t`	Знак табуляции
`\n`	Знак новой строки
`\r`	Разрыв строки
`\b`	Возврат на символ (backspace)
`\\`	Обратная косая черта (обратный слеш)

В MySQL можно добавлять в строку управляющие последовательности с помощью символа \:

```
SELECT 'hello', 'he\'llo', '\thello';

+-------+--------+------------+
| hello | he'llo |      hello |
+-------+--------+------------+
```

В PostgreSQL можно добавлять в строки управляющие последовательности, если вся строка начинается с буквы E или e:

```
SELECT 'hello', E'he\'llo', e'\thello';
----------+----------+--------------
hello     | he'llo   | hello
```

Управляющие последовательности применяются только к строкам, взятым в одинарные кавычки, но не к строкам, обрамленным знаками доллара.

Символьные типы данных

Наиболее распространенный способ хранения строковых значений — использование символьных типов данных. В этом коде создается символьный столбец переменной длины, позволяющий хранить до 50 символов:

```
CREATE TABLE my_table (
   my_character_column VARCHAR(50)
);

INSERT INTO my_table VALUES
   ('Here is some text.'),
   ('And some numbers - 1 2 3 4 5'),
   ('And some punctuation! :)');

SELECT * FROM my_table;

+------------------------------+
| my_character_column          |
+------------------------------+
| Here is some text.           |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)     |
+------------------------------+
```

Есть три основных типа символьных данных.

- VARCHAR (строки символов переменной длины). Это наиболее популярный строковый тип данных. Если тип данных — VARCHAR(50), то в столбце может быть до 50 символов. Другими словами, длина строки является переменной.
- CHAR (строки символов постоянной длины). Если тип данных — CHAR(5), то каждое значение в столбце будет содержать ровно пять символов. Другими словами, длина строки фиксирована. Данные будут дополнены пробелами справа, чтобы их длина была точно равна указанной. Например, 'hi' будет храниться как 'hi '.
- TEXT. Этот тип, в отличие от VARCHAR и CHAR, не требует ввода данных, то есть не нужно указывать длину текста. Это удобный вариант для хранения длинных строк, например одного или нескольких абзацев текста.

В табл. 6.9 перечислены варианты символьных типов данных для каждой РСУБД.

Таблица 6.9. Символьные типы данных

РСУБД	Тип данных	Диапазон данных	По умолчанию	Размер хранения
MySQL	CHAR	От 0 до 255 символов	CHAR(1)	Варьируется
	VARCHAR	От 0 до 65 535 символов	Требуется ввод данных	Варьируется
	TINY TEXT	Нет входных данных	Нет входных данных	255 байт
	TEXT	Нет входных данных	Нет входных данных	65 535 байт
	MEDIUM TEXT	Нет входных данных	Нет входных данных	16 777 215 байт

Продолжение ⇗

Таблица 6.9 *(продолжение)*

РСУБД	Тип данных	Диапазон данных	По умолчанию	Размер хранения
	LARGE TEXT	Нет входных данных	Нет входных данных	4 294 967 295 байт
Oracle	CHAR	От 1 до 2000 символов	CHAR(1)	Варьируется
	VARCHAR2	От 1 до 4000 символов	Требуется ввод данных	Варьируется
	LONG	Нет входных данных	Нет входных данных	2 Гбайт
PostgreSQL	CHAR	От 1 до 10 485 760 символов	CHAR(1)	Варьируется
	VARCHAR	От 1 до 10 485 760 символов	Требуется ввод данных	Варьируется
	TEXT	Нет входных данных	Нет входных данных	Варьируется
SQL Server	CHAR	От 1 до 8000 байт	Требуется ввод данных	Варьируется
	VARCHAR	От 1 до 8000 байт или максимум	Требуется ввод данных	Варьируется или до 2 Гбайт
	TEXT	Нет входных данных	Нет входных данных	2 147 483 647 байт
SQLite	TEXT	Нет входных данных	Нет входных данных	Варьируется

Вместо VARCHAR в Oracle обычно используется VARCHAR2. С точки зрения функциональности эти типы идентичны, но VARCHAR может быть когда-нибудь изменен, поэтому безопаснее использовать VARCHAR2.

Типы данных Unicode

Символьные типы данных обычно хранятся в виде данных ASCII, но также могут храниться и в виде данных *Unicode*, если требуется более крупная библиотека символов.

КОДИРОВКА ASCII В СРАВНЕНИИ С UNICODE

Есть множество способов *кодирования* данных, или, другими словами, преобразования данных в 0 и 1, чтобы их мог понимать компьютер. Кодировка, используемая в SQL по умолчанию, называется *ASCII (American Standard Code for Information Interchange)*.

В ASCII $2^8 = 128$ символов, которые превращаются в серию из восьми 0 и 1. Например, символ ! соответствует 00100001. Эти восемь 0 и 1 называются *байтом* данных.

Помимо ASCII, существуют и другие типы кодировок, такие как UTF *(Unicode Transformation Format)*. В Unicode 221 символ:

- первые 28 символов совпадают с ASCII (! = 100001);
- к другим символам относятся азиатские иероглифы, математические символы, эмодзи и т. д.;
- еще не всем символам присвоены значения.

Этот код демонстрирует разницу между типами данных VARCHAR и NVARCHAR (Unicode):

```
CREATE TABLE my_table (
   ascii_text VARCHAR(10),
   unicode_text NVARCHAR(10)
);

INSERT INTO my_table VALUES
   ('abc', 'abc'),
   (N'赵欣婉', N'赵欣婉');

SELECT * FROM my_table;
```

```
+------------+--------------+
| ascii_text | unicode_text |
+------------+--------------+
| abc        | abc          |
| ???        | 赵欣婉       |
+------------+--------------+
```

При вставке данных Unicode из текстового файла в столбец NVARCHAR значения Unicode в текстовом файле не нуждаются в префиксе N.

В табл. 6.10 перечислены варианты типов данных Unicode для каждой РСУБД.

Таблица 6.10. Типы данных Unicode

РСУБД	Тип данных	Описание
MySQL	NCHAR	Как CHAR, но для данных Unicode
	NVARCHAR	Как VARCHAR, но для данных Unicode
Oracle	NCHAR	Как CHAR, но для данных Unicode
	NVARCHAR2	Как VARCHAR2, но для данных Unicode
PostgreSQL	CHAR	Поддерживает данные Unicode
	VARCHAR	Поддерживает данные Unicode
SQL Server	NCHAR	Как CHAR, но для данных Unicode
	NVARCHAR	Как VARCHAR, но для данных Unicode
SQLite	TEXT	Поддерживает данные Unicode

Данные даты и времени

В этом разделе мы поговорим о значениях даты и времени и выясним, какова форма их представления в SQL, а затем подробно рассмотрим типы данных даты и времени в каждой из РСУБД.

Столбцы с данными даты и времени могут быть входными параметрами в функциях даты и времени, таких как DATEDIFF() и EXTRACT(), которые рассматриваются в разделе «Функции даты и времени» главы 7.

Значения даты и времени

Значения времени могут быть представлены в виде дат, времени или даты и времени (DATETIME).

Значения даты

Столбец даты должен содержать значения даты в формате YYYYMM-DD. Например, в Oracle по умолчанию используется формат DD-MON-YYYY.

Дата 15 октября 2022 года записывается так:

```
'2022-10-15'
```

В Oracle 15 октября 2022 года записывается так:

```
'15-OCT-2022'
```

При ссылке на значение даты в запросе перед строкой необходимо поставить ключевое слово DATE или CAST, чтобы сообщить SQL, что это дата, как показано в табл. 6.11.

Таблица 6.11. Ссылка на дату в запросе

РСУБД	Код

MySQL	`SELECT DATE '2021-02-25';`
	`SELECT DATE('2021-02-25');`
	`SELECT CAST('2021-02-25' AS DATE);`

Oracle	`SELECT DATE '2021-02-25' FROM dual;` `SELECT CAST('25-FEB-2021' AS DATE) FROM dual;`

PostgreSQL	`SELECT DATE '2021-02-25';`
	`SELECT DATE('2021-02-25');`
	`SELECT CAST('2021-02-25' AS DATE);`

SQL Server	`SELECT CAST('2021-02-25' AS DATE);`
SQLite	`SELECT DATE('2021-02-25');`

В Oracle формат даты после ключевого слова DATE отличается от формата даты внутри функции CAST.

Кроме того, в Oracle при выполнении вычислений или поиске системной переменной, содержащей только предложение SELECT, в конце запроса необходимо добавить FROM dual. Словом dual обозначается фиктивная таблица, содержащая одно значение.

```
SELECT DATE '2021-02-25' FROM dual;
SELECT CURRENT_DATE FROM dual;
```

Если столбец содержит даты в другом формате, например `ММ/ДД/ГГ`, то распознать дату в SQL можно с помощью функции преобразования строки в дату.

Значения времени

Столбец времени должен содержать значения времени в формате `hh:mm:ss`. Например, 10:30 утра записывается так:

```
'10:30:00'
```

Можно добавить и более детальные значения времени, включая секунды, вплоть до шести знаков после запятой:

```
'10:30:12.345678'
```

Можно добавить часовой пояс. Центральное стандартное время (Central Standard Time) также известно как `UTC-06:00`:

```
'10:30:12.345678 -06:00'
```

При ссылке на значение времени в запросе перед строкой необходимо поставить ключевое слово `TIME` или `CAST`, чтобы сообщить SQL, что это время, как показано в табл. 6.12.

В Oracle формат времени после ключевого слова TIME должен содержать и секунды.

Если столбец содержит время в другом формате, например mmss, то можно применить функцию преобразования строки в формат времени, чтобы SQL распознал его как время.

Таблица 6.12. Ссылка на время в запросе

РСУБД	Код

MySQL	`SELECT TIME '10:30';`
	`SELECT TIME('10:30');`
	`SELECT CAST('10:30' AS TIME);`

Oracle	`SELECT TIME '10:30:00' FROM dual;` `SELECT CAST('10:30' AS TIME) FROM dual;`
PostgreSQL	`SELECT TIME '10:30';` `SELECT CAST('10:30' AS TIME);`
SQL Server	`SELECT CAST('10:30' AS TIME);`
SQLite	`SELECT TIME('10:30');`

Значения даты и времени

Столбец даты и времени должен содержать значения DATETIME в формате YYYY-MM-DD hh:mm:ss. В Oracle по умолчанию используется формат DDMON-YYYY hh:mm:ss.

Дата 15 октября 2022 года, 10:30, записывается так:

```
'2022-10-15 10:30'
```

В Oracle 15 октября 2022 года, 10:30, записывается так:

```
'15-OCT-2022 10:30'
```

При ссылке на значение даты и времени в запросе перед строкой необходимо поставить ключевое слово DATETIME, TIMESTAMP или CAST, чтобы сообщить SQL, что это дата и время, как показано в табл. 6.13.

Таблица 6.13. Ссылка на дату и время (DATETIME) в запросе

РСУБД	Код

MySQL	`SELECT TIMESTAMP '2021-02-25 10:30';`
	`SELECT TIMESTAMP('2021-02-25 10:30');`
	`SELECT CAST('2021-02-25 10:30' AS DATETIME);`

Oracle	`SELECT TIMESTAMP '2021-02-25 10:30:00'`
	`FROM dual;`
	`SELECT CAST('25-FEB-2021 10:30'`
	`AS TIMESTAMP) FROM dual;`

PostgreSQL	`SELECT TIMESTAMP '2021-02-25 10:30';` `SELECT CAST('2021-02-25 10:30' AS TIMESTAMP);`
SQL Server	`SELECT CAST('2021-02-25 10:30' AS DATETIME);`
SQLite	`SELECT DATETIME('2021-02-25 10:30');`

В MySQL ключевым словом является TIMESTAMP, но тип данных в функции CAST — DATETIME.

В Oracle формат даты после ключевого слова TIMESTAMP отличается от формата даты в функции CAST. Кроме того, формат времени после ключевого слова TIMESTAMP должен содержать секунды, но это не обязательно для функции CAST.

Если столбец содержит дату и время другого формата, например `MM/DD/YY mm:ss`, то можно применить функцию преобразования строки в формат даты и времени, чтобы SQL распознал его как дату и время.

Типы данных DATETIME

Есть множество способов хранения значений даты и времени. Поскольку типы данных очень сильно различаются, то в этом подразделе каждой РСУБД посвящен отдельный пункт.

Типы данных DATETIME в MySQL

Этот код создает пять различных столбцов DATETIME:

```
CREATE TABLE my_table (
   dt DATE,
   tm TIME,
   dttm DATETIME,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   yr YEAR
);

INSERT INTO my_table (dt, tm, dttm, yr)
   VALUES ('21-7-4', '6:30',
           2021, '2021-12-25 7:00:01');

+------------+----------+---------------------+
| dt         | tm       | dttm                |
+------------+----------+---------------------+
| 2021-07-04 | 06:30:00 | 2021-12-25 07:00:01 |
+------------+----------+---------------------+

+---------------------+------+
| ts                  | yr   |
+---------------------+------+
| 2021-01-29 12:56:20 | 2021 |
+---------------------+------+
```

В табл. 6.14 перечислены варианты типа данных DATETIME, широко используемые в MySQL.

Таблица 6.14. Типы данных DATETIME в MySQL

Тип данных	Формат	Диапазон
DATE	YYYY-MM-DD	С 1000-01-01 по 9999-12-31
TIME	hh:mm:ss	С –838:59:59 по 838:59:59
DATETIME	YYYY-MM-DD hh:mm:ss	С 1000-01-01 00:00:00 по 9999-12-31 23:59:59
TIMESTAMP	YYYY-MM-DD hh:mm:ss	С 1970-01-01 00:00:01 UTC по 2038-01-19 03:14:07 UTC
YEAR	YYYY	С 0000 по 9999

И DATETIME, и TIMESTAMP хранят информацию о дате и времени. Разница заключается в том, что DATETIME не имеет привязки к часовому поясу, а TIMESTAMP хранит значения Unix (определенный момент времени) и часто используется для обозначения момента создания или обновления записи.

Типы данных DATETIME в Oracle

Этот код создает четыре различных столбца DATETIME:

```
CREATE TABLE my_table (
   dt DATE,
   ts TIMESTAMP,
   ts_tz TIMESTAMP WITH TIME ZONE,
   ts_lc TIMESTAMP WITH LOCAL TIME ZONE
);

INSERT INTO my_table VALUES (
   '4-Jul-21', '4-Jul-21 6:30',
   '4-Jul-21 6:30:45AM CST', '4-Jul-21 6:30'
);

DT            TS
------------  ------------------------------
04-JUL-21     04-JUL-21 06.30.00.000000 AM

TS_TZ
-------------------------------------
04-JUL-21 06.30.45.000000 AM CST

TS_LC
------------------------------
04-JUL-21 06.30.00.000000 AM
```

В табл. 6.15 перечислены опции типа данных DATETIME, широко используемые в Oracle.

Таблица 6.15. Типы данных DATETIME в Oracle

Тип данных	Описание
DATE	Может хранить либо только дату, либо дату и время, если установлен NLS_DATE_FORMAT

Тип данных	Описание
TIMESTAMP	Аналогичен DATE, но с добавлением дробных секунд (по умолчанию — шесть цифр, но после десятичной точки может содержать до девяти цифр)
TIMESTAMP WITH TIME ZONE	Как TIMESTAMP, но добавляется часовой пояс
TIMESTAMP WITH LOCAL TIME ZONE	Аналогичен TIMESTAMP WITH TIME ZONE, но настраивается в зависимости от местного часового пояса пользователя

Проверка форматов даты и времени в Oracle

Этот код проверяет текущие форматы даты и временной метки:

```
SELECT value
FROM nls_session_parameters
WHERE parameter in ('NLS_DATE_FORMAT',
                    'NLS_TIMESTAMP_FORMAT');

VALUE
---------------------------
DD-MON-RR
DD-MON-RR HH.MI.SSXFF A
```

Чтобы изменить формат даты или метки времени, можно изменить параметр NLS_DATE_FORMAT или NLS_TIMESTAMP_FORMAT. В этом коде изменяется текущий NLS_DATE_FORMAT = DD-MON-RR, чтобы можно было добавить и время:

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH:MI:SS';
```

Другие широко используемые символы даты и времени, такие как YYYY для года и HH для часа, приведены в табл. 7.27.

Типы данных DATETIME в PostgreSQL

Этот код создает пять различных столбцов DATETIME:

```
CREATE TABLE my_table (
   dt DATE,
   tm TIME,
```

```
   tm_tz TIME WITH TIME ZONE,
   ts TIMESTAMP,
   ts_tz TIMESTAMP WITH TIME ZONE
);

INSERT INTO my_table VALUES (
   '2021-7-4', '6:30', '6:30 CST',
   '2021-12-25 7:00:01', '2021-12-25 7:00:01 CST'
);

    dt      |   tm     |    tm_tz    |
------------+----------+-------------+
2021-07-04  | 06:30:00 | 06:30:00-06 |

        ts           |         ts_tz
---------------------+-----------------------
2021-12-25 07:00:01  | 2021-12-25 07:00:01-06
```

В табл. 6.16 перечислены варианты типа данных `DATETIME`, широко используемые в PostgreSQL.

Таблица 6.16. Типы данных DATETIME в PostgreSQL

Тип данных	Формат	Диапазон
`DATE`	`YYYY-MM-DD`	С 4713 года до н. э. до 5 874 897 года н. э.
`TIME`	`hh:mm:ss`	С 00:00:00 до 24:00:00
`TIME WITH TIME ZONE`	`hh:mm:ss+tz`	С 00:00:00+1459 до 24:00:00–1459
`TIMESTAMP`	`YYYY-MM-DD hh:mm:ss`	С 4713 года до н. э. до 294 276 года н. э.
`TIMESTAMP WITH TIME ZONE`	`YYYY-MM-DD hh:mm:ss+tz`	С 4713 года до н. э. до 294 276 года н. э.

Типы данных DATETIME в SQL Server

Этот код создает шесть различных столбцов `DATETIME`:

```
CREATE TABLE my_table (
   dt DATE,
```

```
   tm TIME,
   dttm_sm SMALLDATETIME,
   dttm DATETIME,
   dttm2 DATETIME2,
   dttm_off DATETIMEOFFSET
);

INSERT INTO my_table VALUES (
   '2021-7-4', '6:30', '2021-12-25 7:00:01',
   '2021-12-25 7:00:01', '2021-12-25 7:00:01',
   '2021-12-25 7:00:01-06:00'
);

dt              tm
--------------  ---------------------
2021-07-04      06:30:00.00000000

dttm_sm
-----------------------
2021-12-25 07:00:00

dttm
---------------------------
2021-12-25 07:00:01.000

dttm2
-----------------------------
2021-12-25 07:00:01.0000000

dttm_off
------------------------------------
2021-12-25 07:00:01.0000000 -06:00
```

В табл. 6.17 перечислены варианты типов данных `DATETIME`, широко используемые в SQL Server.

Таблица 6.17. Типы данных DATETIME в SQL Server

Тип данных	Формат	Диапазон
`DATE`	`YYYY-MM-DD`	С 01.01.0001 по 31.12.9999
`TIME`	`hh:mm:ss`	С 00:00:00.0000000 до 23:59:59.9999999

Продолжение ⇩

Таблица 6.17 *(продолжение)*

Тип данных	Формат	Диапазон
SMALLDATETIME	YYYY-MM-DD hh:mm:ss	Дата: с 01.01.1900 по 06.06.2079. Время: с 0:00:00 до 23:59:59
DATETIME	YYYY-MM-DD hh:mm:ss	Дата: с 01.01.1753 по 31.12.9999. Время: с 00:00:00 до 23:59:59.999
DATETIME2	YYYY-MM-DD hh:mm:ss	Дата: с 01.01.0001 по 31.12.9999. Время: с 00:00:00 до 23:59:59.9999999
DATETIMEOFFSET	YYYY-MM-DD hh:mm:ss +hh:mm	Смещение часового пояса варьируется от –12:00 до +14:00

Типы данных DATETIME в SQLite

В SQLite нет типа данных DATETIME. Вместо него для хранения значений даты и времени можно использовать TEXT, REAL или INTEGER.

Несмотря на то что в SQLite нет специальных типов данных DATETIME, функции DATETIME, в том числе DATE(), TIME() и DATETIME(), позволяют работать с датами и в этой РСУБД.

Более подробную информацию можно найти в разделе «Функции даты и времени» главы 7.

В этом коде показаны три способа хранения значений даты и времени в SQLite:

```
CREATE TABLE my_table (
   dt_text TEXT,
   dt_real REAL,
   dt_integer INTEGER
);

INSERT INTO my_table VALUES (
   '2021-12-25 7:00:01',
```

```
   '2021-12-25 7:00:01',
   '2021-12-25 7:00:01'
);

dt_text|dt_real
2021-12-25 7:00:01|2021-12-25 7:00:01

dt_integer
2021-12-25 7:00:01
```

В табл. 6.18 перечислены варианты типа данных DATETIME в SQLite.

Таблица 6.18. Типы данных DATETIME в SQLite

Тип данных	Описание
TEXT	Хранится в виде строки в формате YYYY-MM-DD HH:MM:SS.SSS
REAL	Хранится как число дней по юлианскому календарю, то есть количество дней, прошедших с полудня в Гринвиче 24 ноября 4714 года до н. э.
INTEGER	Хранится как время Unix, которое представляет собой количество секунд с 01.01.1970 00:00:00 UTC

Другие данные

В SQL существует множество других типов данных, в том числе характерных для каждой РСУБД.

Одни из них попадают в одну из существующих категорий типов данных, но содержат более подробные данные, например числовой тип MONEY или DATETIME-тип INTERVAL.

Другие собирают более сложную информацию, например геопространственные данные, отмечающие конкретное место на Земле, или веб-данные, хранящиеся в форматах JSON/XML.

В этом разделе рассматриваются два дополнительных типа данных: булевы данные и данные из внешних файлов.

Булевы данные

Два булевых значения — `TRUE` и `FALSE`. Они нечувствительны к регистру и должны быть записаны без кавычек:

```
SELECT TRUE, True, FALSE, False;
+------+------+-------+-------+
|    1 |    1 |     0 |     0 |
+------+------+-------+-------+
```

Булевы типы данных

MySQL, PostgreSQL и SQLite поддерживают типы данных `Boolean`. Этот код создает столбец типа `Boolean`:

```
CREATE TABLE my_table (
   my_boolean_column BOOLEAN
);

INSERT INTO my_table VALUES
   (TRUE),
   (false),
   (1);

SELECT * FROM my_table;

+------------------+
| my_boolean_column |
+------------------+
|                1 |
|                0 |
|                1 |
+------------------+
```

В Oracle и SQL Server нет булевых типов данных, но есть обходные пути:

- в Oracle для хранения значений `'T'` и `'F'` служит тип данных `CHAR(1)`, а для хранения значений `1` и `0` — тип данных `NUMBER(1)`;
- в SQL Server следует использовать тип данных `BIT`, который содержит значения `1`, `0` и `NULL`.

Внешние файлы (изображения, документы и т. д.)

Если в столбец данных планируется добавить изображения (`.jpg`, `.png` и т. д.) или документы (`.doc`, `.pdf` и т. д.), то это можно сделать с помощью двух подходов: хранить ссылки на файлы (более распространенный) или хранить файлы в виде бинарных значений.

Подход 1. Хранение ссылок на файлы. Обычно такой подход рекомендуется, если размер файлов превышает 1 Мбайт. Для справки: средний размер фотографии на iPhone составляет несколько мегабайт.

Файлы будут храниться вне базы данных, что снижает нагрузку на нее и часто приводит к повышению производительности.

Чтобы хранить ссылки на файлы, выполните следующие действия.

1. Запишите пути к файлам в файловой системе (`/Users/images/img_001.jpg`).
2. Создайте столбец, в котором будут храниться строки, например `VARCHAR(100)`.
3. Вставьте имена путей в столбец.

Подход 2. Хранение файлов в виде бинарных значений. Такой подход рекомендуется, если размер файлов небольшой.

Файлы будут храниться внутри базы данных, что упрощает такие задачи, как резервное копирование данных.

Чтобы хранить двоичные значения, выполните следующие действия.

1. Преобразуйте файлы в бинарный формат (если открыть бинарный файл, то он будет выглядеть как случайная

последовательность символов, например `Z™/≈jhJcE Ät, ÷mfPfõrà`).

2. Создайте столбец, в котором будут храниться бинарные значения, например `BLOB`.
3. Вставьте бинарные значения файлов в столбец.

Бинарные и шестнадцатеричные значения

Бинарные данные представляют собой необработанные значения, которые интерпретируются компьютером. Часто они отображаются в более компактной, удобной для восприятия человеком форме, называемой *шестнадцатеричной*.

- Символ: `a`.
- Эквивалентное двоичное значение: `01100001`.
- Эквивалентное шестнадцатеричное значение: `61`.

Шестнадцатеричные числа преобразуют 1 и 0 в систему счисления из 16 символов (0–9 и A–F). Шестнадцатеричные числа обозначаются символами `X`, `x` или `0x`:

```
SELECT X'AF12', x'AF12', 0xAF12;
+----------+----------+--------+
| 0xAF12   | 0xAF12   | 0xAF12 |
+----------+----------+--------+
```

MySQL поддерживает все три формата. PostgreSQL поддерживает первые два. SQL Server и SQLite поддерживают третий формат.

В Oracle шестнадцатеричное значение нельзя просто отобразить, но вместо этого можно использовать функцию `TO_NUMBER`, позволяющую отобразить шестнадцатеричное значение в виде числа: `SELECT TO_NUMBER('AF12', 'XXXX') FROM dual`; при этом `X` означает шестнадцатеричную систему счисления.

Бинарные типы данных

Этот код создает столбец бинарных данных:

```
CREATE TABLE my_table (
   my_binary_column BLOB
);

INSERT INTO my_table VALUES
   ('a'),
   ('aaa'),
   ('ae$ iou');

SELECT * FROM my_table;

+-------------------------------------+
| my_binary_column                    |
+-------------------------------------+
| 0x61                                |
| 0x616161                            |
| 0x61652420696F75                    |
+-------------------------------------+
```

В MySQL, Oracle и SQLite наиболее распространенным бинарным типом данных является `BLOB`.

В PostgreSQL вместо `BLOB` следует использовать `bytea`, а в SQL Server — `VARBINARY` (например, `VARBINARY(100)`).

В Oracle и SQL Server строка ae$ iou не распознается автоматически как бинарное значение и должна быть преобразована в такое значение перед вставкой в таблицу.

```
-- Oracle
SELECT RAWTOHEX('ae$ iou') FROM dual;

-- SQL Server
SELECT CONVERT(VARBINARY, 'ae$ iou');
```

В табл. 6.19 перечислены варианты бинарных типов данных для каждой РСУБД.

Таблица 6.19. Бинарные типы данных

РСУБД	Тип данных	Описание	Диапазон данных	Размер хранения
MySQL	`BINARY`	Бинарная строка фиксированной длины, в которой значения дополняются справа нулями для получения точного размера	От 0 до 255 байт	Варьируется
	`VARBINARY`	Бинарная строка переменной длины	От 0 до 65 535 байт	Варьируется
	`TINY BLOB`	Крошечный бинарный объект	Нет входных данных	255 байт
	`BLOB`	Бинарный объект	Нет входных данных	65 535 байт
	`MEDIUM BLOB`	Средний бинарный объект	Нет входных данных	16 777 215 байт
	`LARGE BLOB`	Большой бинарный объект	Нет входных данных	4 294 967 295 байт
Oracle	`RAW`	Бинарная строка переменной длины	от 1 до 32 767 байт	Варьируется
	`LONG RAW`	Бинарный объект	Нет входных данных	2 Гбайт
	`BLOB`	Большой бинарный объект	Нет входных данных	4 Гбайт
PostgreSQL	`BYTEA`	Бинарная строка переменной длины	Нет входных данных	1 или 4 байта плюс фактическая бинарная строка

РСУБД	Тип данных	Описание	Диапазон данных	Размер хранения
SQL Server	`BINARY`	Бинарная строка фиксированной длины, в которой значения дополняются нулями для достижения точного размера	От 1 до 8000 байт	Варьируется
	`VARBINARY`	Бинарная строка переменной длины	От 1 до 8000 байт или максимум	Варьируется или до 2 Гбайт
SQLite	`BLOB`	Большой бинарный объект	Нет входных данных	Хранится в том виде, в котором был введен

ГЛАВА 7

Операции и функции

Операции и *функции* используются для выполнения вычислений, сравнений и преобразований в операторе SQL. В этой главе приведены примеры кода для часто используемых операций и функций.

В этом запросе выделены пять операций (+, =, OR, BETWEEN, AND) и две функции (UPPER, YEAR):

```
-- Повышение оплаты труда работников
SELECT name, pay_rate + 5 AS new_pay_rate
FROM employees
WHERE UPPER(title) = 'ANALYST'
      OR YEAR(start_date) BETWEEN 2016 AND 2018;
```

ОПЕРАЦИИ В СРАВНЕНИИ С ФУНКЦИЯМИ

Операции — это символы или ключевые слова, которые выполняют вычисления или сравнения. Операции можно найти в предложениях SELECT, ON, WHERE и HAVING запроса.

Функции принимают ноль или более входных данных, применяют вычисления или преобразования и выдают значение. Функции можно найти в предложениях SELECT, WHERE и HAVING запроса.

Помимо операторов SELECT, операции и функции могут использоваться в операторах INSERT, UPDATE и DELETE. Эта глава

содержит один раздел, посвященный операциям, и пять разделов о различных функциях: агрегатных, числовых, строковых, функциях даты и времени и Null-функциях.

Ниже перечислены широко используемые операции (табл. 7.1) и функции (табл. 7.2).

Таблица 7.1. Широко используемые операции

Логические операции	Операции сравнения (символы)	Операции сравнения (ключевые слова)	Математические операции
AND	=	BETWEEN	+
OR	!=, <>	EXISTS	–
NOT	<	IN	*
	<=	IS NULL	/
	>	LIKE	%
	>=		

Таблица 7.2. Широко используемые функции

Агрегатные функции	Числовые функции	Строковые функции	Функции даты и времени	Null-функции
COUNT()	ABS()	LENGTH()	CURRENT_DATE	COALESCE()
SUM()	SQRT()	TRIM()	CURRENT_TIME	
AVG()	LOG()	CONCAT()	DATEDIFF()	
MIN()	ROUND()	SUBSTR()	EXTRACT()	
MAX()	CAST()	REGEXP()	CONVERT()	

Операции

Операции могут быть символами или ключевыми словами. Они могут выполнять вычисления (+) или сравнения (BETWEEN). В данном разделе описаны операции, имеющиеся в SQL.

Логические операции

Логические операции используются для изменения условий, результатом которых являются значения TRUE, FALSE или NULL. В этом блоке кода логические операции (NOT, AND, OR) выделены жирным шрифтом:

```
SELECT *
FROM employees
WHERE start_date IS NOT NULL
      AND (title = 'analyst' OR pay_rate < 25);
```

При использовании AND и OR для объединения нескольких условных операторов следует четко указывать порядок выполнения операций с помощью круглых скобок: ().

В табл. 7.3 перечислены логические операции в SQL.

Таблица 7.3. Логические операции

Операция	Описание
AND	Возвращает TRUE, если оба условия равны TRUE. Возвращает FALSE, если одно из условий является FALSE. В противном случае возвращает NULL
OR	Возвращает TRUE, если одно из условий равно TRUE. Возвращает FALSE, если оба условия FALSE. В противном случае возвращает NULL
NOT	Возвращает TRUE, если условие равно FALSE. Возвращает FALSE, если условие равно TRUE. В противном случае возвращает NULL

Представьте столбец name. В табл. 7.4 показано, как значения в нем будут определяться в условном операторе без NOT и с NOT.

Представьте два столбца name и age. В табл. 7.5 показано, как значения в них будут определяться в условном операторе с помощью операций AND и OR.

Таблица 7.4. Пример NOT

name	name IN ('Henry', 'Harper')	name NOT IN ('Henry', 'Harper')
Henry	TRUE FALSE	FALSE
Lily	FALSE TRUE	TRUE
NULL	NULL	NULL

Таблица 7.5. Пример AND и OR

name	age	name 'Henry'	age > 3	name = 'Henry' AND age > 3	name = 'Henry' OR age > 3
Henry	5	TRUE	TRUE	TRUE	TRUE
Henry	1	TRUE	FALSE	FALSE	TRUE
Lily	2	FALSE	FALSE	FALSE	FALSE
Henry	NULL	TRUE	NULL	NULL	TRUE
Lily	NULL	FALSE	NULL	FALSE	NULL

Операции сравнения

Операции сравнения используются в предикатах.

ОПЕРАЦИИ В СРАВНЕНИИ С ПРЕДИКАТАМИ

Предикаты — это сравнения, содержащие операцию:

- предикат `age = 35` содержит операцию `=`;
- предикат `COUNT(id) < 20` содержит операцию `<`.

Предикаты также известны как условные операторы. Эти сравнения оцениваются для каждой строки таблицы и приводят к значению `TRUE`, `FALSE` или `NULL`.

В этом блоке кода операции сравнения (`IS NULL`, `=`, `BETWEEN`) выделены жирным шрифтом:

```
SELECT *
FROM employees
WHERE start_date IS NOT NULL
      AND (title = 'analyst'
      OR pay_rate BETWEEN 15 AND 25);
```

В табл. 7.6 перечислены операции сравнения, являющиеся символами, а в табл. 7.7 — операции сравнения, являющиеся ключевыми словами.

Таблица 7.6. Операции сравнения (символы)

Операция	Описание
=	Тесты на равенство
!=, <>	Тесты на неравенство
<	Тесты на меньше чем
<=	Тесты на меньше или равно
>	Тесты на больше чем
>=	Тесты на больше или равно

В MySQL также допускается использование символа `<=>`, который является нуль-безопасным тестом на равенство.

При использовании =, если сравниваются два значения и одно из них равно NULL, итоговым значением будет NULL.

При использовании <=>, если сравниваются два значения и одно из них равно NULL, итоговым значением будет 0. Если оба значения равны NULL, то итоговым будет 1.

Таблица 7.7. Операции сравнения (ключевые слова)

Операция	Описание
BETWEEN	Проверяет, находится ли значение в заданном диапазоне
EXISTS	Проверяет, существуют ли строки в подзапросе

Операция	Описание
`IN`	Проверяет, содержится ли значение в списке значений
`IS NULL`	Проверяет, является ли значение `NULL`
`LIKE`	Проверяет, соответствует ли значение простому шаблону

Операция LIKE используется для сопоставления простых шаблонов, например для поиска текста, начинающегося с буквы A. Более подробную информацию можно найти ниже, в пункте, посвященном операции LIKE.

Регулярные выражения используются для сопоставления более сложных шаблонов, например для извлечения любого текста, расположенного между двумя знаками препинания. Более подробную информацию можно найти в подразделе «Использование регулярных выражений» далее в текущей главе.

Поговорим о каждой операции сравнения ключевых слов более подробно.

BETWEEN

С помощью `BETWEEN` можно проверить, попадает ли значение в определенный диапазон. Эта операция представляет собой комбинацию `>=` и `<=`. Меньшее из двух значений всегда должно быть записано первым, а операция `AND` разделяет их.

Найдем все строки, в которых значение столбца `age` больше или равно 35 и меньше или равно 44:

```
SELECT *
FROM my_table
WHERE age BETWEEN 35 AND 44;
```

Найдем все строки, в которых значение столбца `age` меньше 35 или больше 44:

```
SELECT *
FROM my_table
WHERE age NOT BETWEEN 35 AND 44;
```

EXISTS

Операция EXISTS используется для проверки того, возвращает ли подзапрос результаты. Как правило, подзапрос ссылается на другую таблицу.

Этот запрос возвращает данные о сотрудниках, которые также являются клиентами:

```
SELECT e.id, e.name
FROM employees e
WHERE EXISTS (SELECT *
              FROM customers c
              WHERE c.email = e.email);
```

EXISTS В СРАВНЕНИИ С JOIN

Запрос EXISTS также может быть составлен с использованием JOIN:

```
SELECT *
FROM employees e INNER JOIN customers c
     ON e.email = c.email;
```

JOIN более предпочтителен, когда требуется вернуть значения из обеих таблиц (SELECT *).

EXISTS лучше использовать тогда, когда требуется вернуть значения из одной таблицы (SELECT e.id, e.name). Такой тип запроса иногда называют *полусоединением* (semi-join). Кроме того, EXISTS полезен, когда во второй таблице есть повторяющиеся строки, а вас интересует только то, существует ли строка.

Этот запрос возвращает данные о клиентах, которые никогда не совершали покупок:

```
SELECT c.id, c.name
FROM customers c
WHERE NOT EXISTS (SELECT *
                  FROM orders o
                  WHERE o.email = c.email);
```

IN

Операция IN используется для проверки того, попадает ли значение в список значений. Этот запрос возвращает значения для нескольких сотрудников:

```
SELECT *
FROM employees
WHERE e.id IN (10001, 10032, 10057);
```

Этот запрос возвращает данные о сотрудниках, не использовавших дни отпуска:

```
SELECT e.id
FROM employees e
WHERE e.id NOT IN (SELECT v.emp_id
                   FROM vacations v);
```

При использовании NOT IN, если в столбце подзапроса (в данном случае v.emp_id) есть хотя бы одно значение NULL, подзапрос никогда не будет иметь значение TRUE, а значит, ни одна строка не будет возвращена.

Если в столбце в подзапросе потенциально возможно значение NULL, то лучше использовать NOT EXISTS:

```
SELECT e.id
FROM employees e
WHERE NOT EXISTS (SELECT *
                  FROM vacations v
                  WHERE v.emp_id = e.id);
```

IS NULL

Операции IS NULL или IS NOT NULL используются для проверки того, является ли значение NULL.

Этот запрос возвращает данные о сотрудниках, у которых нет руководителя:

```
SELECT *
FROM employees
WHERE manager IS NULL;
```

А этот возвращает данные о сотрудниках, у которых руководитель есть:

```
SELECT *
FROM employees
WHERE manager IS NOT NULL;
```

LIKE

Операция `LIKE` используется для определения соответствия простому шаблону. Знак процента (%) — подстановочный, означает один или несколько символов.

Рассмотрим пример таблицы:

```
SELECT * FROM my_table;
```

```
+------+--------------------+
| id   | txt                |
+------+--------------------+
|    1 | You are great.     |
|    2 | Thank you!         |
|    3 | Thinking of you.   |
|    4 | I'm 100% positive. |
+------+--------------------+
```

Найдем все строки, *содержащие* выражение you:

```
SELECT *
FROM my_table
WHERE txt LIKE '%you%';

-- Результаты работы с MySQL, SQL Server и SQLite
+------+------------------+
| id   | txt              |
+------+------------------+
|    1 | You are great.   |
|    2 | Thank you!       |
|    3 | Thinking of you. |

-- Результаты работы с Oracle и PostgreSQL
+------+------------------+
| id   | txt              |
+------+------------------+
| 2    | Thank you!       |
| 3    | Thinking of you. |
+------+------------------+
```

В MySQL, SQL Server и SQLite шаблон нечувствителен к регистру. Значения `You` и `you` соответствуют шаблону `'%you%'`.

В Oracle и PostgreSQL шаблон чувствителен к регистру. Шаблону `'%you%'` соответствует только значение `you`.

Найдем все строки, которые *начинаются* с выражения `You`:

```
SELECT *
FROM my_table
WHERE txt LIKE 'You%';

+------+---------------+
| id   | txt           |
+------+---------------+
| 1    | You are great. |
+------+---------------+
```

Используем `NOT LIKE` для возврата строк, не содержащих указанных символов.

Вместо знака процента (`%`) для соответствия одному или нескольким символам можно использовать знак подчеркивания (`_`), чтобы соответствовать ровно одному символу.

Поскольку % и _ имеют особое значение при использовании с LIKE, то, если вы хотите найти эти символы, необходимо добавить ключевое слово ESCAPE.

Этот код находит все строки, содержащие символ %:

```
SELECT *
FROM my_table
WHERE txt LIKE '%!%%' ESCAPE '!';

+------+--------------------+
| id   | txt                |
+------+--------------------+
| 4    | I'm 100% positive. |
+------+--------------------+
```

После ключевого слова ESCAPE мы объявили символ ! в качестве escape-символа, поэтому, когда символ ! ставится перед средним знаком процента в %!%%, сочетание !% интерпретируется как %.

Операция `LIKE` полезна при поиске определенной строки символов. Для более сложного поиска по шаблону можно использовать регулярные выражения, которые рассматриваются в подразделе «Использование регулярных выражений» далее в этой главе.

Математические операции

Математические операции — это математические символы, которые можно использовать в SQL. В этом блоке кода математическая операция (`/`) выделена жирным шрифтом:

```
SELECT salary / 52 AS weekly_pay
FROM my_table;
```

В табл. 7.8 перечислены математические операции в SQL.

Таблица 7.8. Математические операции

Операция	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление
% (MOD в Oracle)	Остаток от деления одного числа на другое

В PostgreSQL, SQL Server и SQLite при делении целого числа на целое число получается целое число:

```
SELECT 15/2;
7
```

Если нужно, чтобы результат включал десятичные числа, то можно либо разделить на десятичную дробь, либо использовать функцию CAST:

```
SELECT 15/2.0;
7.5

 -- PostgreSQL и SQL Server
SELECT CAST(15 AS DECIMAL) /
       CAST(2 AS DECIMAL);
       7.5

-- SQLite
SELECT CAST(15 AS REAL) /
       CAST(2 AS REAL);
7.5
```

Другие математические операции:

- побитовые операции, такие как `&` (`AND`), `|` (`OR`) и `^` (`XOR`), для работы с битами (значениями `0` и `1`);
- операции присваивания, такие как `+=` (сложение с присваиванием) и `-=` (вычитание с присваиванием), для обновления значений в таблице.

Агрегатные функции

Агрегатная функция выполняет вычисления над многими строками данных, в результате чего получается одно значение. В табл. 7.9 перечислены пять основных агрегатных функций в SQL.

Таблица 7.9. Основные агрегатные функции

Функция	Описание
`COUNT()`	Подсчитывает количество значений
`SUM()`	Вычисляет сумму столбца
`AVG()`	Вычисляет среднее значение столбца
`MIN()`	Находит минимальное значение столбца
`MAX()`	Находит максимальное значение столбца

Агрегатные функции применяют вычисления к значениям в столбце, не равным NULL. Единственное исключение — функция COUNT(*), которая подсчитывает *все* строки, включая нулевые значения.

Кроме того, вы можете объединить (агрегировать) несколько строк в один список с помощью таких функций, как ARRAY_AGG, GROUP_CONCAT, LISTAGG и STRING_AGG. Более подробную информацию можно найти в подразделе «Агрегирование строк в одно значение или список» в главе 8.

Oracle поддерживает дополнительные агрегатные функции, такие как медиана (MEDIAN), режим (STATS_MODE) и стандартное отклонение (STDDEV).

Агрегатные функции (в примере выделены жирным шрифтом) расположены в предложениях SELECT и HAVING следующего запроса:

```
SELECT COUNT(*) AS total_rows,
       AVG(age) AS average_age
FROM my_table;

SELECT region, MIN(age), MAX(age)
FROM my_table
GROUP BY region
HAVING MIN(age) < 18;
```

Если в операторе SELECT присутствуют как агрегированные, так и неагрегированные столбцы, то необходимо добавить все неагрегированные столбцы в предложение GROUP BY (в предыдущем примере это region).

Одни РСУБД выдают ошибку, если этого не сделать. Другие системы (например, SQLite) не выдают ошибку и позволяют выполнить оператор, даже если возвращаемые результаты будут неточными. Рекомендуется перепроверять результаты, чтобы убедиться, что они имеют смысл.

MIN/MAX В СРАВНЕНИИ С LEAST/GREATEST

Функции MIN и MAX находят наименьшее и наибольшее значения в столбце.

Функции LEAST и GREATEST находят наименьшее и наибольшее значения в строке. Входными данными могут быть числовые, строковые или значения даты и времени. Если одно из значений равно NULL, то функция возвращает NULL.

В таблице ниже показано общее количество миль, пройденных в каждом квартале, а запрос находит наибольшее количество миль, пройденных в конкретном квартале:

```
SELECT * FROM goat;

+--------+------+------+------+------+
| name   | q1   | q2   | q3   | q4   |
+--------+------+------+------+------+
| Ali    | 100  | 200  | 150  | NULL |
| Bolt   | 350  | 400  | 380  | 300  |
| Jordan | 200  | 250  | 300  | 320  |
+--------+------+------+------+------+

SELECT name, GREATEST(q1, q2, q3, q4)
             AS most_miles
FROM goat;

+--------+------------+
| name   | most_miles |
+--------+------------+
| Ali    | NULL       |
| Bolt   | 400        |
| Jordan | 320        |
+--------+------------+
```

Числовые функции

Числовые функции могут применяться к столбцам с числовыми типами данных. В этом разделе рассматриваются числовые функции, широко используемые в SQL.

Применение математических функций

В SQL существует несколько типов математических вычислений.

- *Математические операции*. Вычисления, выполняемые с помощью символов `+`, `-`, `*`, `/` и `%`.
- *Агрегатные функции*. Вычисления, которые суммируют весь столбец данных в одно значение, такие как `COUNT`, `SUM`, `AVG`, `MIN` и `MAX`.
- *Математические функции*. Вычисления, выполняемые с использованием ключевых слов, применимых к каждой строке данных, таких как `SQRT`, `LOG` и др., которые перечислены в табл. 7.10.

SQLite поддерживает только функцию ABS. Другие математические функции необходимо добавлять вручную. Более подробную информацию можно найти на странице математических функций на сайте SQLite (https://oreil.ly/0XwjB).

Таблица 7.10. Математические функции

Категория	Функция	Описание	Код	Результат
Положительные и отрицательные значения	`ABS`	Абсолютное значение	`SELECT ABS(-5);`	`5`
	`SIGN`	Возвращает –1, 0 или 1 в зависимости от того, является число отрицательным, нулевым или положительным	`SELECT SIGN(-5);`	`-1`
Экспоненты и логарифмы	`POWER`	*x*, возведенное в степень *y*	`SELECT POWER(5,2);`	`25`
	`SQRT`	Квадратный корень	`SELECT SQRT(25);`	`5`

Категория	Функция	Описание	Код	Результат
	EXP	*e* (=2,71828), возведенное в степень *x*	SELECT EXP(2);	7.389
	LN (LOG в SQL Server)	Логарифм *y* по основанию *x*	SELECT LOG(2,10); SELECT LOG(10,2);	3.322
	LOG (LOG(y,x) в SQL Server)	Натуральный логарифм (по основанию *e*)	SELECT LOG10(100); SELECT LOG(10,100) FROM dual;	2
	LOG10 (LOG(10,x) в Oracle)	Логарифм *x* по основанию 10	SELECT LOG10(100); SELECT LOG(10,100) FROM dual;	2
Другие функции	MOD (x%y в SQL Server)	Остаток от деления *x* на *y*	SELECT MOD(12,5); SELECT 12%5;	2
	PI (отсутствует в Oracle)	Значение числа π	SELECT PI();	3.14159
	COS, SIN, и т. д.	Косинус, синус и другие тригонометрические функции (входные данные в радианах)	SELECT COS(.78);	0.711

Генерация случайных чисел

В табл. 7.11 показано, как генерировать случайное число в каждой РСУБД. В некоторых случаях можно ввести *начальное число*, чтобы генерируемые случайные числа каждый раз были одинаковыми.

Таблица 7.11. Генератор случайных чисел

РСУБД	Код	Диапазон результатов

MySQL, SQL Server	`SELECT RAND();`
	
	`-- Необязательное начальное значение`
	`SELECT RAND(22);`
		От 0 до 1

Oracle	`SELECT DBMS_RANDOM.VALUE FROM dual;`
	
	`SELECT DBMS_RANDOM.RANDOM FROM dual;`
		От 0 до 1
		
		От –2E31 до +2E31

PostgreSQL	`SELECT RANDOM();`	От 0 до 1
SQLite	`SELECT RANDOM();`	От –9E18 до +9E18

Функция случайных чисел иногда используется для возврата нескольких случайных строк таблицы. Хотя это и не самый эффективный запрос (поскольку таблица должна быть отсортирована), такой способ является быстрым:

```
-- Вернуть пять случайных строк
SELECT *
FROM my_table
ORDER BY RANDOM()
LIMIT 5;
```

Oracle и SQL Server позволяют производить выборку таблицы случайных строк:

```
-- Возврат 20 % случайных строк в Oracle
SELECT *
FROM my_table
SAMPLE(20);

-- Возврат произвольных 100 строк в SQL Server
SELECT *
FROM my_table
TABLESAMPLE(100 ROWS);
```

Округление и усечение чисел

В табл. 7.12 приведены различные способы округления чисел в каждой РСУБД.

Таблица 7.12. Параметры округления

Функция	Описание	Код	Результат
`CEIL` (`CEILING` в SQL Server)	Округление до ближайшего целого числа	`SELECT CEIL(98.7654); SELECT CEILING(98.7654);`	`99`
`FLOOR`	Округление до ближайшего целого числа	`SELECT FLOOR(98.7654);`	`98`
`ROUND`	Округляет до определенного количества десятичных знаков, по умолчанию 0 десятичных знаков	`SELECT ROUND(98.7654,2);`	`98,77`
`TRUNC` (`TRUNCATE` в MySQL; `ROUND(x,y,1)` в SQL Server)	Обрезает число до определенного количества знаков после точки, по умолчанию 0 знаков после точки	`SELECT TRUNC(98.7654,2); SELECT TRUNCATE(98.7654,2); SELECT ROUND(98.7654,2,1);`	`98,76`

SQLite поддерживает только функцию ROUND. Другие варианты округления необходимо добавлять вручную. Более подробную информацию можно найти на странице математических функций на сайте SQLite (https://oreil.ly/rF2Rt).

Преобразование данных в числовой тип

Функция `CAST` предназначена для преобразования различных типов данных и часто используется для числовых данных.

В следующем примере мы хотим сравнить строковый столбец с числовым.

Так выглядит таблица со строковым столбцом:

```
+------+---------+
| id   | str_col |
+------+---------+
|    1 | 1.33    |
|    2 | 5.5     |
|    3 | 7.8     |
+------+---------+
```

Попробуем сравнить строковый столбец с числовым значением:

```
SELECT *
FROM my_table
WHERE str_col > 3;

-- Результаты для MySQL, Oracle и SQLite
+------+---------+
| id   | str_col |
+------+---------+
|    2 | 5.5     |
|    3 | 7.8     |
+------+---------+

-- Результаты для PostgreSQL и SQL Server
Error
```

В MySQL, Oracle и SQLite запрос возвращает правильные результаты, поскольку строковый столбец распознается как числовой при использовании оператора >.

В PostgreSQL и SQL Server необходимо явно преобразовать строковый столбец в числовой.

Приведем строковый столбец к десятичному, чтобы сравнить его с числом:

```
SELECT *
FROM my_table
WHERE CAST(str_col AS DECIMAL) > 3;

id  | str_col
----+---------
2   | 5.5
3   | 7.8
```

Использование CAST не приводит к постоянному изменению типа данных столбца. Чтобы изменить тип не только на время выполнения запроса, а навсегда, можно изменить таблицу.

Строковые функции

Строковые функции можно применять к столбцам со строковыми типами данных. В этом разделе рассматриваются строковые операции, широко используемые в SQL.

Нахождение длины строки

Используем функцию `LENGTH`.

В предложении `SELECT`:

```
SELECT LENGTH(name)
FROM my_table;
```

В предложении `WHERE`:

```
SELECT *
FROM my_table
WHERE LENGTH(name) < 10;
```

В SQL Server вместо `LENGTH` используем `LEN`.

В большинстве РСУБД при вычислении длины строки конечные пробелы исключаются, а в Oracle — добавляются.

Пример строки: 'Al '

Длина: 2

Длина в Oracle: 5

Чтобы исключить конечные пробелы в Oracle, используйте функцию TRIM:

```
SELECT LENGTH(TRIM(name))
FROM my_table;
```

Изменение регистра строки

Используем функции `UPPER` или `LOWER`.

`UPPER`:

```
SELECT UPPER(type)
FROM my_table;
```

`LOWER`:

```
SELECT *
FROM my_table
WHERE LOWER(type) = 'public';
```

В Oracle и PostgreSQL также имеется функция `INITCAP(string)` для перевода первой буквы каждого слова в строке в верхний регистр, а остальных букв — в нижний.

Удаление нежелательных символов вокруг строки

С помощью функции `TRIM` можно удалить как начальные, так и конечные символы вокруг строкового значения. В этой таблице есть несколько символов, которые мы хотели бы удалить:

```
SELECT * FROM my_table;

+----------------+
| color          |
+----------------+
| !!red          |
|  .orange!      |
|    ..yellow..  |
+----------------+
```

Удаление пробелов вокруг строки

По умолчанию `TRIM` удаляет пробелы как с левой, так и с правой стороны строки:

```
SELECT TRIM(color) AS color_clean
FROM my_table;

+-------------+
| color_clean |
+-------------+
| !!red       |
| .orange!    |
| ..yellow..  |
+-------------+
```

Удаление других символов вокруг строки

Помимо пробела, можно удалить и другие символы. Этот код удаляет восклицательные знаки вокруг строки:

```
SELECT TRIM('!' FROM color) AS color_clean
FROM my_table;

+-----------------+
| color_clean     |
+-----------------+
| red             |
|  .orange        |
|    ..yellow..   |
+-----------------+
```

В SQLite вместо этого используется функция `TRIM(color, '!')`.

Удаление символов в левой или правой части строки

Есть два варианта удаления символов по обе стороны от строки.

- *Вариант 1.* Использовать `TRIM(LEADING ...)` и `TRIM(TRAILING ...)`. В MySQL, Oracle и PostgreSQL можно удалять символы как из левой, так и из правой части строки с помощью `TRIM(LEADING ...)` и `TRIM(TRAILING ...)` соответственно. Этот код удаляет восклицательные знаки в начале строки:

```
SELECT TRIM(LEADING '!' FROM color) AS color_clean
FROM my_table;
+----------------+
| color_clean    |
+----------------+
| red            |
|  .orange!      |
|    ..yellow..  |
+----------------+
```

- *Вариант 2.* Использовать `LTRIM` и `RTRIM`. Ключевые слова `LTRIM` и `RTRIM` используются для удаления символов из левой или правой части строки соответственно.

  В Oracle, PostgreSQL и SQLite все нежелательные символы могут быть перечислены в одной строке. Этот код удаляет точки, восклицательные знаки и пробелы из начала строки:

```
SELECT LTRIM(color, '.! ') AS color_clean
FROM my_table;

+-------------+
| color_clean |
+-------------+
| red         |
| orange!     |
| yellow..    |
+-------------+
```

  В MySQL и SQL Server с помощью функции `LTRIM(color)` или `RTRIM(color)` можно удалять только пробельные символы.

Конкатенация строк

Используем функцию CONCAT или оператор конкатенации (||).

```
-- MySQL, PostgreSQL и SQL Server
Server SELECT CONCAT(id, '_', name) AS id_name
FROM my_table;

-- Oracle, PostgreSQL и SQLite
SELECT id || '_' || name AS id_name
FROM my_table;

+-----------+
| id_name   |
+-----------+
| 1_Boots   |
| 2_Pumpkin |
| 3_Tiger   |
+-----------+
```

Поиск текста в строке

Есть два подхода к поиску текста в строке.

- *Подход 1. Проверить, содержится ли в строке текст.* Определить наличие или отсутствие текста в строке можно с помощью операции LIKE. При использовании этого запроса будут возвращены только строки, содержащие текст some:

  ```
  SELECT *
  FROM my_table
  WHERE my_text LIKE '%some%';
  ```

  Более подробную информацию можно найти в пункте «LIKE» ранее в этой главе.

- *Подход 2. Определить местоположение текста в строке.* Для этого можно использовать функцию INSTR/POSITION/CHARINDEX.

В табл. 7.13 перечислены параметры, с помощью которых функции определяют местоположение текста в каждой РСУБД.

Таблица 7.13. Функции для поиска местоположения текста в строке

РСУБД	Формат кода
MySQL	`INSTR(string, substring)` `LOCATE(substring, string, position)`
Oracle	`INSTR(string, substring, position, occurrence)`
PostgreSQL	`POSITION(substring IN string)` `STRPOS(string, substring)`
SQL Server	`CHARINDEX(substring, string, position)`
SQLite	`INSTR(string, substring)`

Входными параметрами являются:

- *`string` (обязательный)* — строка, в которой производится поиск (например, имя столбца VARCHAR);
- *`substring` (обязательный)* — искомая строка (то есть символ, слово и т. д.);
- *`position` (необязательный)* — начальная позиция для поиска. По умолчанию поиск начинается с первого символа (1). Если позиция отрицательная, то поиск начинается с конца строки;
- *`occurrence` (необязательный)* — первое/второе/третье и т. д. появление подстроки в строке. По умолчанию используется первое вхождение (1).

Рассмотрим пример таблицы:

```
+-----------------------------+
| my_text                     |
+-----------------------------+
| Here is some text.          |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)    |
+-----------------------------+
```

Найдем местоположение подстроки `some` в строке `my_text`:

```
SELECT INSTR(my_text, 'some') AS some_location
FROM my_table;

+---------------+
| some_location |
+---------------+
|             9 |
|             5 |
|             5 |
+---------------+
```

СЧЕТ В SQL НАЧИНАЕТСЯ С 1

В отличие от других языков программирования, в которых используется нулевая индексация (отсчет начинается с 0), в SQL отсчет начинается с 1.

Символ 9 в предыдущем выводе означает девятый символ.

В Oracle регулярные выражения можно использовать для поиска подстроки с помощью REGEXP_INSTR. Более подробная информация приведена в пункте «Регулярные выражения в Oracle» далее в текущей главе.

Извлечение части строки

Используем функцию `SUBSTR` или `SUBSTRING`. Имя функции и вводимые параметры в каждой РСУБД различаются:

```
-- MySQL, Oracle, PostgreSQL и SQLite
SUBSTR(string, start, length)

-- MySQL
SUBSTR(string FROM start FOR length)

-- MySQL, PostgreSQL и SQL Server
SUBSTRING(string, start, length)

-- MySQL и PostgreSQL
SUBSTRING(string FROM start FOR length)
```

На вход подаются следующие параметры:

- *`string` (обязательный)* — строка, в которой производится поиск (например, имя столбца VARCHAR);
- *`start` (обязательный)* — начальное место поиска. Если значение *`start`* равно `1`, то поиск начнется с первого символа, если `2` — со второго и т. д. Если значение *`start`* равно `0`, то оно будет считаться равным `1`. Если значение *`start`* отрицательное, то поиск будет начинаться с последнего символа;
- *`length` (обязательный)* — длина возвращаемой строки. Если значение *`length`* опущено, то будут возвращены все символы от начала до конца строки. В SQL Server длина является обязательной.

Рассмотрим пример таблицы:

```
+-----------------------------+
| my_text                     |
+-----------------------------+
| Here is some text.          |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)    |
+-----------------------------+
```

Извлечем подстроку:

```
SELECT SUBSTR(my_text, 14, 8) AS sub_str
FROM my_table;

+----------+
| sub_str  |
+----------+
| text.    |
| ers - 1  |
| tuation! |
+----------+
```

В Oracle регулярные выражения можно использовать для извлечения подстроки с помощью REGEXP_SUBSTR. Более подробная информация приведена в пункте «Регулярные выражения в Oracle» далее в текущей главе.

Замена текста в строке

Используем функцию REPLACE. Обратите внимание на порядок входных параметров для функции:

```
REPLACE(string, old_string, new_string)
```

Рассмотрим пример таблицы:

```
+-----------------------------+
| my_text                     |
+-----------------------------+
| Here is some text.          |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)    |
+-----------------------------+
```

Заменим слово some на слово the:

```
SELECT REPLACE(my_text, 'some', 'the')
       AS new_text
FROM my_table;

+-----------------------------+
| new_text                    |
+-----------------------------+
| Here is the text.           |
| And the numbers - 1 2 3 4 5 |
| And the punctuation! :)     |
+-----------------------------+
```

В Oracle и PostgreSQL регулярные выражения можно использовать для замены строки с помощью REGEXP_REPLACE. Более подробную информацию об этом можно найти в пунктах «Регулярные выражения в Oracle» и «Регулярные выражения в PostgreSQL» далее в текущей главе.

Удаление текста из строки

Можно использовать функцию REPLACE, но в качестве значения замены указать пустую строку.

Заменяем слово some на пустую строку:

```
SELECT REPLACE(my_text, 'some ', '')
       AS new_text
FROM my_table;

+-------------------------+
| new_text                |
+-------------------------+
| Here is text.           |
| And numbers - 1 2 3 4 5 |
| And punctuation! :)     |
+-------------------------+
```

Использование регулярных выражений

Регулярные выражения позволяют сопоставлять сложные шаблоны. Например, вы можете найти все слова, состоящие ровно из пяти букв, или все слова, начинающиеся с заглавной буквы.

Представьте, что у вас есть следующий рецепт приправы для тако:

```
- 1 tablespoon chili powder
- .5 tablespoon ground cumin
- .5 teaspoon paprika
- .25 teaspoon garlic powder
- .25 teaspoon onion powder
- .25 teaspoon crushed red pepper flakes
- .25 teaspoon dried oregano
```

Вы хотите исключить количество и получить только список ингредиентов. Для этого можно написать регулярное выражение, которое будет извлекать весь текст, следующий за выражением spoon.

Регулярное выражение будет выглядеть так:

```
(?<=spoon ).*$
```

а результаты — так:

```
chili powder
ground cumin
paprika
garlic powder
```

```
onion powder
crushed red pepper flakes
dried oregano
```

Регулярное выражение просматривает весь текст и извлекает любой текст, который находится между выражением `spoon` и концом строки.

Следует отметить несколько моментов, связанных с регулярными выражениями.

- Синтаксис регулярных выражений не является интуитивно понятным. Полезно разобрать значение каждой части регулярного выражения с помощью онлайн-инструмента, например Regex101 (https://regex101.com).
- Регулярные выражения могут использоваться не только в SQL, но и во многих других языках программирования и текстовых редакторах.
- RegexOne (https://regexone.com) предоставляет краткое вводное руководство. Кроме того, вы можете прочесть статью Томаса Нилда *An introduction to regular expressions* («Введение в регулярные выражения») на сайте O'Reilly (https://oreil.ly/1jJQk).

Вместо того чтобы заучивать синтаксис регулярных выражений, лучше найти существующие регулярные выражения и изменить их в соответствии с вашими потребностями.

Чтобы составить регулярное выражение, приведенное выше, я вводила запрос «текст регулярного выражения после строки».

Второй результат поиска в Google привел меня к регулярному выражению (?<=WORD).*$. Я использовала Regex101 (https://regex101.com), чтобы понять каждую часть регулярного выражения, и в итоге заменила WORD на spoon.

Функции регулярных выражений сильно различаются в зависимости от РСУБД, поэтому ниже поговорим о каждой из них. SQLite не поддерживает регулярные выражения по умолчанию, но их можно реализовать. Более подробную информацию можно найти в документации по SQLite (https://oreil.ly/gmxS6).

Регулярные выражения в MySQL

Используем `REGEXP` для поиска шаблона регулярного выражения в любом месте строки.

Рассмотрим пример таблицы:

```
+--------------------------+-------------+
| title                    | city        |
+--------------------------+-------------+
| 10 Things I Hate About You | Seattle     |
| 22 Jump Street             | New Orleans |
| The Blues Brothers         | Chicago     |
| Ferris Bueller's Day Off   | Chi         |
+--------------------------+-------------+
```

Найдем все варианты написания слова `Chicago`:

```
SELECT *
FROM movies
WHERE city REGEXP '(Chicago|CHI|Chitown)';
```

```
+-------------------------+---------+
| title                   | city    |
+-------------------------+---------+
| The Blues Brothers      | Chicago |
| Ferris Bueller's Day Off | Chi     |
+-------------------------+---------+
```

Регулярные выражения MySQL нечувствительны к регистру символов строк; `CHI` и `Chi` рассматриваются как эквивалентные.

Найдем все фильмы с числами в названии:

```
SELECT *
FROM movies
WHERE title REGEXP '\\d';
```

```
+--------------------------+-------------+
| title                    | city        |
+--------------------------+-------------+
| 10 Things I Hate About You | Seattle     |
| 22 Jump Street             | New Orleans |
+--------------------------+-------------+
```

В MySQL любой одинарный обратный слеш в регулярном выражении (\d = любая цифра) необходимо заменить на двойной.

Регулярные выражения в Oracle

Oracle поддерживает множество функций регулярных выражений, в том числе такие:

- REGEXP_LIKE соответствует шаблону регулярного выражения в тексте;
- REGEXP_COUNT подсчитывает количество раз, когда шаблон встречается в тексте;
- REGEXP_INSTR определяет позиции, в которых шаблон встречается в тексте;
- REGEXP_SUBSTR возвращает подстроки в тексте, соответствующие шаблону;
- REGEXP_REPLACE заменяет подстроки, соответствующие шаблону, на другой текст.

Рассмотрим пример таблицы:

```
 TITLE                        CITY
---------------------------  -------------
 10 Things I Hate About You   Seattle
 22 Jump Street               New Orleans
 The Blues Brothers           Chicago
 Ferris Bueller's Day Off     Chi
```

Найдем все фильмы с числами в названии:

```
SELECT *
FROM movies
WHERE REGEXP_LIKE(title, '\d');

 TITLE                        CITY
---------------------------  -------------
 10 Things I Hate About You   Seattle
 22 Jump Street               New Orleans
```

Следующие выражения эквивалентны:

```
REGEXP_LIKE(title, \d)
REGEXP_LIKE(title, [0-9])
REGEXP_LIKE(title, [[:digit:]])
```

Третий вариант использует синтаксис регулярных выражений POSIX (https://oreil.ly/G3Tkw), который поддерживается Oracle.

Подсчитаем количество заглавных букв в названии:

```
SELECT title, REGEXP_COUNT(title, '[A-Z]')
       AS num_caps
FROM movies;

 TITLE                          NUM_CAPS
----------------------------  ----------
 10 Things I Hate About You            5
 22 Jump Street                        2
 The Blues Brothers                    3
 Ferris Bueller's Day Off              4
```

Найдем местоположение первой гласной в названии:

```
SELECT title, REGEXP_INSTR(title, '[aeiou]')
       AS first_vowel
FROM movies;

 TITLE                          FIRST_VOWEL
----------------------------  -------------
 10 Things I Hate About You              6
 22 Jump Street                          5
 The Blues Brothers                      3
 Ferris Bueller's Day Off                2
```

Выберем все цифры в названии:

```
SELECT title, REGEXP_SUBSTR(title, '[0-9]+')
       AS nums
FROM movies

WHERE REGEXP_SUBSTR(title, '[0-9]+') IS NOT NULL;

 TITLE                          NUMS
----------------------------  ------
 10 Things I Hate About You      10
 22 Jump Street                  22
```

Заменим все цифры в заголовке на число 100:

```
SELECT REGEXP_REPLACE(title, '[0-9]+', '100')
       AS one_hundred_title
FROM movies;

 ONE_HUNDRED_TITLE
----------------------------
 100 Things I Hate About You
 100 Jump Street
```

Более подробную информацию и примеры по регулярным выражениям в Oracle можно найти в книге Oracle Regular Expressions Pocket Reference (https://oreil.ly/5As3T) Джонатана Генника и Питера Линсли (O'Reilly).

Регулярные выражения в PostgreSQL

Используем `SIMILAR TO` или `~` для поиска шаблона регулярного выражения в любом месте строки.

Рассмотрим пример таблицы:

```
           title            |    city
----------------------------+-------------
 10 Things I Hate About You | Seattle
 22 Jump Street             | New Orleans
 The Blues Brothers         | Chicago
 Ferris Bueller's Day Off   | Chi
```

Найдем все варианты написания слова Chicago:

```
SELECT *
FROM movies
WHERE city SIMILAR TO '(Chicago|CHI|Chi|Chitown)';

          title           |  city
--------------------------+---------
 The Blues Brothers       | Chicago
 Ferris Bueller's Day Off | Chi
```

Регулярные выражения PostgreSQL чувствительны к регистру символов строк; `CHI` и `Chi` воспринимаются как разные значения.

SIMILAR TO В СРАВНЕНИИ С ~

Регулярное выражение `SIMILAR TO` обладает ограниченными возможностями и чаще всего используется для предложения нескольких альтернатив (`Chicago|CHI|Chi`). Вместе с `SIMILAR TO` часто используются и другие символы регулярных выражений: * (0 или более), + (1 или более) и { } (точное количество раз).

Тильда (~) должна использоваться для более сложных регулярных выражений наряду с синтаксисом POSIX (https://oreil.ly/Thzdv), который является еще одной разновидностью регулярных выражений, поддерживаемых PostgreSQL.

Полный список поддерживаемых символов можно найти в документации по PostgreSQL (https://oreil.ly/wsB46).

В следующем примере вместо `SIMILAR TO` используется ~.

Найдем все фильмы, в названии которых есть числа:

```
SELECT *
FROM movies
WHERE title ~ '\d';

+---------------------------+-------------+
| title                     | city        |
+---------------------------+-------------+
| 10 Things I Hate About You | Seattle     |
| 22 Jump Street            | New Orleans |
+---------------------------+-------------+
```

PostgreSQL также поддерживает функцию `REGEXP_REPLACE`, которая позволяет заменять символы в строке, соответствующие определенному шаблону.

Заменим все числа в заголовке на 100:

```
SELECT REGEXP_REPLACE(title, '\d+', '100')
FROM movies;

regexp_replace
---------------------------
100 Things I Hate About You
```

```
100 Jump Street
The Blues Brothers
Ferris Bueller's Day Off
```

Регулярное выражение `\d` эквивалентно `[0-9]` и `[[:digit::]]`.

Регулярные выражения в SQL Server

SQL Server поддерживает очень ограниченное количество регулярных выражений с помощью ключевого слова `LIKE`.

Рассмотрим пример таблицы:

```
 title                         city
---------------------------  -------------
 10 Things I Hate About You    Seattle
 22 Jump Street                New Orleans
 The Blues Brothers            Chicago
 Ferris Bueller's Day Off      Chi
```

В SQL Server используется несколько иной синтаксис регулярных выражений, который подробно описан в документации Microsoft (https://oreil.ly/QANyP).

Найдем все фильмы, в названии которых есть числа:

```
SELECT *
FROM movies
WHERE title LIKE '%[0-9]%';
```

```
 title                         city
---------------------------  -------------
 10 Things I Hate About You    Seattle
 22 Jump Street                New Orleans
```

Преобразование данных в строковый тип данных

Когда строковые функции применяются к нестроковым типам данных, это, как правило, не вызывает проблем, даже если имеет место несоответствие типов данных.

В этой таблице есть числовой столбец `numbers`:

```
+---------+
| numbers |
+---------+
| 1.33    |
| 2.5     |
| 3.777   |
+---------+
```

Когда к числовому столбцу применяется строковая функция `LENGTH` (или `LEN` в SQL Server), оператор выполняется без ошибок в большинстве РСУБД:

```
SELECT LENGTH(numbers) AS len_num
FROM my_table;

-- Результаты в MySQL, Oracle, SQL Server и SQLite
+---------+
| len_num |
+---------+
|       4 |
|       3 |
|       5 |
+---------+

-- Результаты в PostgreSQL
Error
```

В PostgreSQL необходимо явно преобразовать числовой столбец в строковый с помощью функции `CAST`:

```
SELECT LENGTH(CAST(numbers AS CHAR(5))) AS len_num
FROM my_table;

 len_num
---------
       4
       3
       5
```

Использование CAST не приводит к постоянному изменению типа данных в столбце. Чтобы изменить тип не только на время выполнения запроса, а навсегда, можно изменить таблицу.

Функции даты и времени

Функции даты и времени могут применяться к столбцам с типом данных DATETIME. В этом разделе рассматриваются функции даты и времени, широко используемые в SQL.

Возврат текущей даты или времени

Эти операторы возвращают текущую дату, текущее время, а также текущие дату и время:

```
-- MySQL, PostgreSQL и SQLite
SELECT CURRENT_DATE;
SELECT CURRENT_TIME;
SELECT CURRENT_TIMESTAMP;

-- Oracle
SELECT CURRENT_DATE FROM dual;
SELECT CAST(CURRENT_TIMESTAMP AS TIME) FROM dual;
SELECT CURRENT_TIMESTAMP FROM dual;

-- SQL Server
SELECT CAST(CURRENT_TIMESTAMP AS DATE);
SELECT CAST(CURRENT_TIMESTAMP AS TIME);
SELECT CURRENT_TIMESTAMP;
```

Есть множество других функций, эквивалентных этим, в том числе CURDATE() в MySQL, GETDATE() в SQL Server и т. д.

В следующих трех примерах показано, как эти функции используются на практике.

Отобразим текущее время:

```
SELECT CURRENT_TIME;

+--------------+
| current_time |
+--------------+
| 20:53:35     |
+--------------+
```

Создадим таблицу, в которой отмечаются дата и время создания строки:

```
CREATE TABLE my_table
       (id INT,
       creation_datetime TIMESTAMP DEFAULT
                          CURRENT_TIMESTAMP);

INSERT INTO my_table (id)
       VALUES (1), (2), (3);
```

```
 +------+---------------------+
 | id   | creation_datetime   |
 +------+---------------------+
 |    1 | 2021-02-15 20:57:12 |
 |    2 | 2021-02-15 20:57:12 |
 |    3 | 2021-02-15 20:57:12 |
 +------+---------------------+
```

Найдем все строки данных, созданных до определенной даты:

```
SELECT *
FROM   my_table
WHERE  creation_datetime < CURRENT_DATE;
```

```
 +------+---------------------+
 | id   | creation_datetime   |
 +------+---------------------+
 |    1 | 2021-01-15 10:47:02 |
 |    2 | 2021-01-15 10:47:02 |
 |    3 | 2021-01-15 10:47:02 |
 +------+---------------------+
```

Добавление или вычитание интервала даты или времени

Из значений даты и времени можно вычитать различные временные интервалы (годы, месяцы, дни, часы, минуты, секунды и т. д.). Эти же интервалы можно и прибавлять к значениям.

В табл. 7.14 перечислены способы вычитания дней.

Таблица 7.14. Возврат вчерашней даты

РСУБД	Код

MySQL	`SELECT CURRENT_DATE - INTERVAL 1 DAY;`
	`SELECT SUBDATE(CURRENT_DATE, 1);`
	`SELECT DATE_SUB(CURRENT_DATE,`
	`INTERVAL 1 DAY);`

Oracle	`SELECT CURRENT_DATE - INTERVAL '1' DAY` `FROM dual;`
PostgreSQL	`SELECT CAST(CURRENT_DATE -` `INTERVAL '1 day' AS DATE);`

SQL Server	`SELECT CAST(CURRENT_TIMESTAMP - 1 AS DATE);`
	`SELECT DATEADD(DAY, -1, CAST(`
	`CURRENT_TIMESTAMP AS DATE));`

SQLite	`SELECT DATE(CURRENT_DATE, '-1 day');`

В табл. 7.15 перечислены способы добавления трех часов.

Таблица 7.15. Возврат даты и времени через три часа

РСУБД	Код

MySQL	`SELECT CURRENT_TIMESTAMP + INTERVAL 3 HOUR;`
	`SELECT ADDDATE(CURRENT_TIMESTAMP,`
	`INTERVAL 3 HOUR);`
	`SELECT DATE_ADD(CURRENT_TIMESTAMP,`
	`INTERVAL 3 HOUR);`

Oracle	`SELECT CURRENT_TIMESTAMP + INTERVAL '3' HOUR` `FROM dual;`
PostgreSQL	`SELECT CURRENT_TIMESTAMP +` `INTERVAL '3 hours';`
SQL Server	`SELECT DATEADD(HOUR, 3, CURRENT_TIMESTAMP);`
SQLite	`SELECT DATETIME(CURRENT_TIMESTAMP,` `'+3 hours');`

Нахождение разницы между двумя датами или временем

Разницу между двумя датами, временами или датой и временем можно найти в терминах различных временных интервалов (годы, месяцы, дни, часы, минуты, секунды и т. д.).

Нахождение разницы между датами

В табл. 7.16 перечислены способы нахождения количества дней между заданными начальной и конечной датами.

Рассмотрим пример таблицы:

```
+------------+------------+
| start_date | end_date   |
+------------+------------+
| 2016-10-10 | 2020-11-11 |
| 2019-03-03 | 2021-04-04 |
+------------+------------+
```

Таблица 7.16. Количество дней между двумя датами

РСУБД	Код

MySQL	`SELECT DATEDIFF(end_date, start_date)`
	`  AS day_diff`
	`FROM  my_table;`

Oracle	`SELECT (end_date - start_date) AS day_diff` `FROM  my_table;`
PostgreSQL	`SELECT AGE(end_date, start_date) AS day_diff` `FROM  my_table;`

SQL Server	`SELECT DATEDIFF(day, start_date, end_date)`
	`  AS day_diff`
	`FROM  my_table;`

SQLite	`SELECT (julianday(end_date) -`
	`  julianday(start_date)) AS day_diff`
	`FROM  my_table;`

После выполнения кода, приведенного в таблице, были получены следующие результаты:

```
-- MySQL, Oracle, SQL Server и SQLite
+----------+
| day_diff |
+----------+
|     1493 |
|      763 |
+----------+

-- PostgreSQL

       day_diff
--------------------
 4 years 1 mon 1 day
 2 years 1 mon 1 day
```

Нахождение разницы во времени

В табл. 7.17 перечислены способы нахождения секунд между заданными начальными и конечными временны́ми промежутками.

Рассмотрим пример таблицы:

```
+------------+----------+
| start_time | end_time |
+------------+----------+
| 10:30:00   | 11:30:00 |
| 14:50:32   | 15:22:45 |
+------------+----------+
```

Таблица 7.17. Количество секунд между двумя временами

РСУБД	Код

MySQL	`SELECT TIMEDIFF(end_time, start_time)`
	`AS time_diff`
	`FROM my_table;`

Oracle	Тип временных данных `time` отсутствует

PostgreSQL	`SELECT EXTRACT(epoch from end_time`
	`start_time) AS time_diff`
	`FROM my_table;`

Продолжение ⇗

Таблица 7.17 *(продолжение)*

РСУБД	Код

SQL Server	`SELECT DATEDIFF(second, start_time, end_time)`
	`  AS time_diff`
	`FROM  my_table;`

SQLite	`SELECT (strftime('%s',end_time)`
	`  strftime('%s',start_time))`
	`  AS time_diff`
	`FROM  my_table;`

После выполнения кода, приведенного в таблице, были получены следующие результаты:

```
-- MySQL
+-----------+
| time_diff |
+-----------+
| 01:00:00  |
| 00:32:13  |
+-----------+

-- PostgreSQL, SQL Server и SQLite

 time_diff
-----------
 3600
 1933
```

Нахождение разницы в дате и времени

Для заданных значений начальной и конечной дат в табл. 7.18 перечислены способы нахождения количества часов между заданными начальной и конечной датами.

Рассмотрим пример таблицы:

```
+---------------------+---------------------+
| start_dt            | end_dt              |
+---------------------+---------------------+
| 2016-10-10 10:30:00 | 2020-11-11 11:30:00 |
| 2019-03-03 14:50:32 | 2021-04-04 15:22:45 |
+---------------------+---------------------+
```

Таблица 7.18. Количество часов между двумя датами

РСУБД	Код

MySQL	`SELECT TIMESTAMPDIFF(hour, start_dt, end_dt)`
	`  AS hour_diff`
	`FROM  my_table;`

Oracle	`SELECT (end_dt - start_dt) AS hour_diff` `FROM  my_table;`
PostgreSQL	`SELECT AGE(end_dt, start_dt) AS hour_diff` `FROM  my_table;`

SQL Server	`SELECT DATEDIFF(hour, start_dt, end_dt)`
	`  AS hour_diff`
	`FROM  my_table;`

SQLite	`SELECT ((julianday(end_dt) -`
	`  julianday(start_dt))*24) AS hour_diff`
	`FROM  my_table;`

После выполнения кода, приведенного в таблице, были получены следующие результаты:

```
-- MySQL, SQL Server и SQLite
+-----------+
| hour_diff |
+-----------+
|     35833 |
|     18312 |
+-----------+

-- Oracle

 HOUR_DIFF
---------------------------
 +000001493 01:00:00.000000
 +000000763 00:32:13.000000

-- PostgreSQL

           hour_diff
-------------------------------
 4 years 1 mon 1 day 01:00:00
 2 years 1 mon 1 day 00:32:13
```

Результат выполнения кода в PostgreSQL очень длинный:

```
SELECT AGE(end_dt, start_dt)
FROM my_table;

              age
------------------------------
 4 years 1 mon 1 day 01:00:00
 2 years 1 mon 1 day 00:32:13
```

С помощью функции EXTRACT можно извлечь только поле year.

```
SELECT EXTRACT(year FROM
               AGE(end_dt, start_dt))
FROM my_table;

 date_part
-----------
         4
         2
```

Извлечение части даты или времени

Есть несколько способов извлечения единицы времени (месяц, час и т. д.) из значения даты или времени. В табл. 7.19 показано, как это сделать, в частности, для единицы времени `month`.

Таблица 7.19. Извлечение месяца из даты

РСУБД	Код
MySQL	`SELECT EXTRACT(month FROM CURRENT_DATE);` `SELECT MONTH(CURRENT_DATE);`
Oracle	`SELECT EXTRACT(month FROM CURRENT_DATE)` `FROM dual;`
PostgreSQL	`SELECT EXTRACT(month FROM CURRENT_DATE);`
SQL Server	`SELECT DATEPART(month, CURRENT_TIMESTAMP);`
SQLite	`SELECT strftime('%m', CURRENT_DATE);`

И MySQL, и SQL Server поддерживают функции, специфичные для единиц времени, такие как `MONTH()`, как показано в табл. 7.19.

- MySQL поддерживает функции `YEAR()`, `QUARTER()`, `MONTH()`, `WEEK()`, `DAY()`, `HOUR()`, `MINUTE()` и `SECOND()`.
- SQL Server поддерживает функции `YEAR()`, `MONTH()` и `DAY()`.

Значения `month` или `%m` в табл. 7.19 можно заменить другими единицами времени. В табл. 7.20 перечислены единицы времени, принятые в каждой из РСУБД.

Таблица 7.20. Варианты единиц измерения времени

MySQL	Oracle	PostgreSQL	SQL Server	SQLite
Микросекунда, секунда, минута, час, день, неделя, месяц, квартал, год	Секунда, минута, час, день, месяц, год	Микросекунда, миллисекунда, секунда, минута, час, день, день недели, неделя, месяц, квартал, год, десятилетие, век	Наносекунда, микросекунда, миллисекунда, секунда, минута, час, неделя, день недели, день, день года, месяц, квартал, год	`%f` (дробная секунда), `%S` (секунда), `%s` (секунд с момента 1970-01-01), `%M` (минута), `%H` (час), `%J` (номер юлианского дня), `%w` (день недели), `%d` (день месяца), `%j` (день года), `%W` (неделя года), `%m` (месяц), `%Y` (год)

Можно извлечь единицу времени из строкового значения. Код приведен в табл. 7.28 (см. ниже).

Определение дня недели для заданной даты

Для заданной даты необходимо определить день недели.

- Дата: `2020-03-16`.
- Числовой день недели: `2` (воскресенье — первый день).
- День недели: `Monday` (понедельник).

В коде табл. 7.21 возвращается значение порядкового номера дня недели для заданной даты. Воскресенье — первый день, понедельник — второй и т. д.

Таблица 7.21. Извлечение порядкового номера дня недели

РСУБД	Код	Диапазон значений
MySQL	`SELECT DAYOFWEEK('2020-03-16');`	От 1 до 7

Oracle	`SELECT TO_CHAR(`
	`  date '2020-03-16', 'd')`
	`FROM  dual;`
		От 1 до 7

PostgreSQL	`SELECT DATE_PART('dow',` `  date '2020-03-16');`	От 0 до 6
SQL Server	`SELECT DATEPART(weekday,` `  '2020-03-16');`	От 1 до 7
SQLite	`SELECT strftime('%w',` `  '2020-03-16');`	От 0 до 6

В коде табл. 7.22 возвращается день недели для заданной даты.

Таблица 7.22. Возврат дня недели

РСУБД	Код
MySQL	`SELECT DAYNAME('2020-03-16');`
Oracle	`SELECT TO_CHAR(date '2020-03-16', 'day')` `FROM  dual;`

РСУБД	Код
PostgreSQL	`SELECT TO_CHAR(date '2020-03-16', 'day');`
SQL Server	`SELECT DATENAME(weekday, '2020-03-16');`
SQLite	Недоступно

Округление даты до ближайшей единицы времени

Oracle и PostgreSQL поддерживают округление и усечение (также известное как округление в меньшую сторону).

Округление в Oracle

Oracle поддерживает округление и усечение даты до ближайшего года, месяца или дня (первого дня недели).

Округлим дату до первого числа месяца:

```
SELECT TRUNC(date '2020-02-25', 'month')
FROM   dual;

01-FEB-20
```

Округлим до ближайшего месяца:

```
SELECT ROUND(date '2020-02-25', 'month')
FROM dual;

01-MAR-20
```

Округление в PostgreSQL

PostgreSQL поддерживает усечение даты до ближайшего года, квартала, месяца, недели (первого дня недели), дня, часа, минуты или секунды. Дополнительные единицы измерения времени можно найти в документации по PostgreSQL (https://oreil.ly/OONv8).

Округлим до первого числа месяца:

```
SELECT DATE_TRUNC('month', DATE '2020-02-25');
```

```
2020-02-01 00:00:00-06
```

Округлим до минуты:

```
SELECT DATE_TRUNC('minute', TIME '10:30:59.12345');
```

```
10:30:00
```

Преобразование строки в тип данных DATETIME

Есть два способа преобразования строки в тип данных `DATETIME`:

- для простого случая используем функцию `CAST`;
- для нестандартного случая используем функции `STR_TO_DATE`/`TO_DATE`/`CONVERT`.

Функция CAST

Если строковый столбец содержит даты в стандартном формате, то с помощью функции `CAST` можно преобразовать его в тип данных `DATE`.

В табл. 7.23 приведен код преобразования к типу данных `DATE`.

Таблица 7.23. Преобразование строки в дату

РСУБД	Требуемый формат даты	Код
MySQL, PostgreSQL, SQL Server	`YYYY-MM-DD`	`SELECT CAST('2020-10-15' AS DATE);`
Oracle	`DD-MON-YYYY`	`SELECT CAST('15-OCT-2020' AS DATE) FROM dual;`
SQLite	`YYYY-MM-DD`	`SELECT DATE('2020-10-15');`

В табл. 7.24 приведен код для преобразования в тип данных TIME.

Таблица 7.24. Преобразование строки в тип данных TIME

РСУБД	Требуемый формат времени	Код
MySQL, PostgreSQL, SQL Server	hh:mm:ss	SELECT CAST('14:30' AS TIME);
Oracle	hh:mm:ss hh:mm:ss AM/PM	SELECT CAST('02:30:00 PM' AS TIME) FROM dual;
SQLite	hh:mm:ss	SELECT TIME('14:30');

В табл. 7.25 приведен код для преобразования к типу данных DATETIME.

Таблица 7.25. Преобразование строки в тип данных DATETIME

РСУБД	Требуемый формат DATETIME	Код
MySQL, SQL Server	YYYY-MM-DD hh:mm:ss	SELECT CAST('2020-10-15 14:30' AS DATETIME);
Oracle	DD-MON-YYYY hh:mm:ss DD-MON-YYYY hh:mm:ss AM/PM	SELECT CAST('15-OCT-20 02:30:00 PM' AS TIMESTAMP) FROM dual;
PostgreSQL	YYYY-MM-DD hh:mm:ss	SELECT CAST('2020-10-15 14:30' AS TIMESTAMP);
SQLite	YYYY-MM-DD hh:mm:ss	SELECT DATETIME('2020-10-15 14:30');

Кроме того, с помощью функции CAST можно преобразовывать даты в числовые и строковые типы данных.

Функции STR_TO_DATE, TO_DATE и CONVERT

Для дат и времени, представленных не в стандартных форматах `YYYY-MM-DD/DD-MONYYY/hh:mm:ss`, можно использовать функцию преобразования строки в дату или строки в тип данных `TIME`.

В табл. 7.26 перечислены функции преобразования строки в дату и строки в тип данных `TIME` для каждой РСУБД. Примеры строк в коде приведены в нестандартных форматах `MM-DD-YY` и `hhmm`.

Таблица 7.26. Функции преобразования строки в дату и строки в тип данных TIME

РСУБД	Строка в дату	Строка в тип данных TIME
MySQL	`SELECT STR_TO_DATE('10-15-22', '%m-%d-%y');`	`SELECT STR_TO_DATE('1030', '%H%i');`
Oracle	`SELECT TO_DATE('10-15-22', 'MM-DD-YY') FROM dual;`	`SELECT TO_TIMESTAMP('1030', 'HH24MI') FROM dual;`
PostgreSQL	`SELECT TO_DATE('10-15-22', 'MM-DD-YY');`	`SELECT TO_TIMESTAMP('1030', 'HH24MI');`
SQL Server	`SELECT CONVERT( VARCHAR, '10-15-22', 105);`	`SELECT CAST( CONCAT(10,':',30) AS TIME);`
SQLite	Нет функции для нестандартной даты	Нет функции для нестандартного времени

SQL Server использует функцию CONVERT для преобразования строкового типа данных в тип данных DATETIME. VARCHAR — исходный тип данных, 10-15-22 — дата, а 105 обозначает формат MM-DDYYYY.

Другие форматы даты выглядят так: MM/DD/YYYY (101), YYYY.MM.DD (102), DD/MM/YYYY (103) и DD.MM.YYYY (104).

Дополнительные форматы перечислены в документации Microsoft (https://oreil.ly/qY0IH).

А вот форматы времени hh:mi:ss (108) и hh:mi:ss:mmm (114). Ни один из них не соответствует формату, приведенному в табл. 7.26, поэтому в SQL Server нельзя считать время с помощью функции CONVERT.

Значения `%H%i` или `HH24MI` в табл. 7.26 можно заменить другими единицами измерения времени. В табл. 7.27 перечислены спецификаторы формата даты и времени, широко используемые в MySQL, Oracle и PostgreSQL.

Таблица 7.27. Спецификаторы формата даты и времени

MySQL	Oracle и PostgreSQL	Описание
%Y	YYYY	Четырехзначный год
%y	YY	Двухзначный год
%m	MM	Порядковый номер месяца (1–12)
%b	MON	Сокращенное название месяца (Jan — Dec)
%M	MONTH	Название месяца (January — December)
%d	DD	День (1–31)
%h	HH или HH12	12-часовой формат времени (1–12)
%H	HH24	24-часовой формат времени (0–23)
%i	MI	Минуты (0–59)
%s	SS	Секунды (0–59)

Применение функции даты к строковому столбцу

Представьте, что у вас есть следующий строковый столбец:

```
str_column
10/15/2022
10/16/2023
10/17/2024
```

Вы хотите извлечь год из каждой даты:

```
year_column
2022
2023
2024
```

- *Проблема.* Нельзя использовать DATETIME-функцию (EXTRACT) для строкового столбца (str_column).
- *Решение.* Сначала преобразуйте строковый столбец в столбец даты. Затем примените функцию DATETIME. В табл. 7.28 перечислены способы выполнения этих действий в каждой РСУБД.

Таблица 7.28. Извлечение года из строки

РСУБД	Код
MySQL	`SELECT YEAR(STR_TO_DATE(str_column, '%m/%d/%Y')) FROM my_table;`
Oracle	`SELECT EXTRACT(YEAR FROM TO_DATE(str_column, 'MM/DD/YYYY')) FROM my_table;`
PostgreSQL	`SELECT EXTRACT(YEAR FROM TO_DATE(str_column, 'MM/DD/YYYY')) FROM my_table;`
SQL Server	`SELECT YEAR(CONVERT(CHAR, str_column, 101)) FROM my_table;`
SQLite	`SELECT SUBSTR(str_column, 7) FROM my_table;`

В SQLite нет функций для работы с датами, но в качестве обходного пути можно использовать функцию SUBSTR (подстрока) для извлечения последних четырех цифр.

Функции NULL

Эти функции могут применяться к столбцам любого типа и срабатывают при обнаружении значения NULL.

Возврат альтернативного значения при наличии значения NULL

Используем функцию COALESCE.

Рассмотрим пример таблицы:

```
+------+----------+
| id   | greeting |
+------+----------+
|    1 | hi there |
|    2 | hello!   |
|    3 | NULL     |
+------+----------+
```

Если столбец greeting имеет значение NULL, то вернем hi:

```
SELECT COALESCE(greeting, 'hi') AS greeting
FROM my_table;

+----------+
| greeting |
+----------+
| hi there |
| hello!   |
| hi       |
+----------+
```

Кроме того, в MySQL и SQLite допускается использование IFNULL(greeting, 'hi'), в Oracle — NVL(greeting, 'hi'), в SQL Server — ISNULL(greeting, 'hi').

ГЛАВА 8

Расширенные концепции запросов

В этой главе мы поговорим о некоторых расширенных возможностях работы с данными с помощью SQL-запросов, помимо шести основных предложений и общих ключевых слов, с которыми мы познакомились в главах 4 и 7 соответственно.

В табл. 8.1 приведены описания и примеры кода четырех концепций, рассматриваемых в этой главе.

Таблица 8.1. Расширенные концепции запросов

Концепция	Описание	Пример кода

Операторы CASE	Если условие выполнено, то вернуть определенное значение. В противном случае вернуть другое значение	`SELECT house_id,`
		`  CASE WHEN flg = 1`
		`  THEN 'for sale'`
		`  ELSE 'sold' END`
		`FROM houses;`

Группировка и агрегирование	Разделить данные на группы, агрегировать данные внутри каждой группы и вернуть значение для каждой *группы*	`SELECT zip, AVG(ft)`
		`FROM houses`
		`GROUP BY zip;`

Оконные функции	Разделить данные на группы, агрегировать или упорядочить данные внутри каждой группы и вернуть значение для каждой *строки*	`SELECT zip,`
		`  ROW_NUMBER() OVER`
		`  (PARTITION BY zip`
		`  ORDER BY price)`
		`FROM houses;`

Концепция	Описание	Пример кода

Операции PIVOT и UNPIVOT	Преобразовать значения в столбце в несколько столбцов или объединить несколько столбцов в один. Поддерживается Oracle и SQL Server	`-- Синтаксис Oracle`
		`SELECT *`
		`FROM listing_info`
		`PIVOT`
		`  (COUNT(*) FOR`
		`  room IN ('bd','br'));`

В этой главе подробно описывается каждая из концепций, представленных в табл. 8.1, а также типичные случаи их использования.

Операторы CASE

Оператор `CASE` используется для применения логики `IF-ELSE` в запросе. Например, с его помощью можно указать значения. Если встречается `1`, то вывести `vip`. В противном случае отобразить `general admission` (общий вход).

```
+--------+       +-------------------+
| ticket |       | ticket            |
+--------+       +-------------------+
|      1 |       | vip               |
|      0 |  -->  | general admission |
|      1 |       | vip               |
+--------+       +-------------------+
```

В Oracle можно также встретить функцию `DECODE`, которая является более старой функцией, работающей аналогично оператору `CASE`.

С помощью оператора CASE можно обновить значения на время выполнения запроса. Для сохранения обновленных значений можно использовать оператор UPDATE.

В следующих двух подразделах рассматриваются два типа операторов CASE:

- *простой оператор* CASE для *одного столбца* данных;
- *поисковый оператор* CASE для *нескольких столбцов* данных.

Отображение значений на основе логики IF-THEN для одного столбца

Равенство в пределах одного столбца данных проверяется с помощью простого синтаксиса оператора CASE.

Наша цель — вместо отображения значений 1/0/NULL отобразить значения vip/reserved seating/general admission:

- если flag = 1, то ticket = vip;
- если flag = 0, то ticket = reserved seating (зарезервированное место);
- в противном случае ticket = general admission (общий вход).

Рассмотрим пример таблицы:

```
SELECT * FROM concert;

+-------+------+
| name  | flag |
+-------+------+
| anton |    1 |
| julia |    0 |
| maren |    1 |
| sarah | NULL |
+-------+------+
```

Реализуем логику IF-THEN с помощью простого оператора CASE:

```
SELECT name, flag,
   CASE flag WHEN 1 THEN 'vip'
   WHEN 0 THEN 'reserved seating'
   ELSE 'general admission' END AS ticket
FROM concert;
```

```
+-------+------+-------------------+
| name  | flag | ticket            |
+-------+------+-------------------+
| anton |    1 | vip               |
| julia |    0 | reserved seating  |
| maren |    1 | vip               |
| sarah | NULL | general admission |
+-------+------+-------------------+
```

Если ни одно из предложений `WHEN` не соответствует указанному и не задано значение `ELSE`, то будет возвращено значение `NULL`.

Отображение значений на основе логики IF-THEN для нескольких столбцов

Любое условие (`=`, `<`, `IN`, `IS NULL` и т. д.) в потенциально нескольких столбцах данных проверяется с помощью синтаксиса искомого оператора `CASE`.

Наша цель — вместо отображения значений `1`/`0`/`NULL` отобразить значения `vip`/`reserved seating`/`general admission`:

- если `name = anton`, то `ticket = vip`;
- если `flag = 0` или `flag = 1`, то `ticket = reserved seating` (зарезервированное место);
- в противном случае `ticket = general admission` (общий вход).

Рассмотрим пример таблицы:

```
SELECT * FROM concert;
```

```
+-------+------+
| name  | flag |
+-------+------+
| anton |    1 |
| julia |    0 |
| maren |    1 |
| sarah | NULL |
+-------+------+
```

Реализуем логику IF-THEN с помощью простого оператора CASE:

```
SELECT name, flag,
   CASE WHEN name = 'anton' THEN 'vip'
   WHEN flag IN (0,1) THEN 'reserved seating'
   ELSE 'general admission' END AS ticket
FROM concert;

+-------+------+-------------------+
| name  | flag | ticket            |
+-------+------+-------------------+
| anton |    1 | vip               |
| julia |    0 | reserved seating  |
| maren |    1 | reserved seating  |
| sarah | NULL | general admission |
+-------+------+-------------------+
```

Если выполняется несколько условий, то приоритет имеет первое из перечисленных условий.

Чтобы заменить все значения NULL в столбце другим значением, можно использовать оператор CASE, но чаще всего вместо него используется NULL-функция COALESCE.

Группировка и агрегирование

SQL позволяет разделять строки на группы и агрегировать строки внутри каждой группы определенным образом, возвращая в конечном итоге только одну строку на группу.

В табл. 8.2 перечислены концепции, связанные с группировкой и агрегированием данных.

Таблица 8.2. Концепции группировки и агрегирования

Категория	Ключевое слово	Описание
Основная концепция	GROUP BY	Используем предложение GROUP BY для разделения строк данных на группы

Категория	Ключевое слово	Описание
Способы агрегирования строк внутри каждой группы	`COUNT`	Эти агрегатные функции агрегируют несколько строк данных в *одно значение*
	`SUM`	
	`MIN`	
	`MAX`	
	`AVG`	
	`ARRAY_AGG`	Эти функции объединяют несколько строк данных в *один список*
	`GROUP_CONCAT`	
	`LISTAGG`	
	`STRING_AGG`	
Расширения предложения `GROUP BY`	`ROLLUP`	Содержит строки для промежуточных итогов, а также для общего итога
	`CUBE`	Содержит агрегаты для всех возможных комбинаций, сгруппированных по столбцам
	`GROUPING SETS`	Позволяет указать конкретные группировки для отображения

Основы работы с GROUP BY

В этой таблице показано количество калорий, сожженных двумя людьми:

```
SELECT * FROM workouts;
```

```
+------+----------+
| name | calories |
+------+----------+
| ally |       80 |
| ally |       75 |
| ally |       90 |
| jess |      100 |
| jess |       92 |
+------+----------+
```

Для создания сводной таблицы необходимо решить, как это сделать:

1) сгруппировать данные: разделить все значения имен на две группы — `ally` и `jess`;
2) агрегировать данные по группам: найти общее количество калорий внутри каждой группы.

Для создания сводной таблицы используем предложение `GROUP BY`:

```
SELECT name,
       SUM(calories) AS total_calories
FROM workouts
GROUP BY name;
```

```
+------+----------------+
| name | total_calories |
+------+----------------+
| ally |            245 |
| jess |            192 |
+------+----------------+
```

Более подробно о том, как работает `GROUP BY`, можно прочитать в разделе «Предложение GROUP BY» главы 4.

Группировка по нескольким столбцам

В этой таблице показано количество калорий, сжигаемых двумя людьми во время ежедневных тренировок:

```
SELECT * FROM daily_workouts;
```

```
+------+------+------------+----------+
| id   | name | date       | calories |
+------+------+------------+----------+
|    1 | ally | 2021-03-03 |       80 |
|    1 | ally | 2021-03-04 |       75 |
|    1 | ally | 2021-03-05 |       90 |
|    2 | jess | 2021-03-03 |      100 |
|    2 | jess | 2021-03-05 |       92 |
+------+------+------------+----------+
```

Если вы пишете запрос с предложением `GROUP BY`, который группирует по нескольким столбцам и/или содержит несколько агрегатов, то:

- предложение SELECT должно содержать все *имена столбцов* и *агрегатов*, которые вы хотите отображать в выходных данных;
- предложение GROUP BY должно содержать те же *имена столбцов*, которые есть и в предложении SELECT.

Используем предложение GROUP BY для суммирования статистики по каждому человеку, возвращая id и name, а также два агрегата:

```
SELECT id, name,
       COUNT(date) AS workouts,
       SUM(calories) AS calories
FROM daily_workouts
GROUP BY id, name;

+------+------+----------+----------+
| id   | name | workouts | calories |
+------+------+----------+----------+
|    1 | ally |        3 |      245 |
|    2 | jess |        2 |      192 |
+------+------+----------+----------+
```

СОКРАЩЕНИЕ СПИСКА GROUP BY ДЛЯ ПОВЫШЕНИЯ ЭФФЕКТИВНОСТИ

Если известно, что каждый идентификатор связан с одним именем, то можно исключить столбец name из предложения GROUP BY и получить те же результаты, которые дал предыдущий запрос:

```
SELECT id,
       MAX(name) AS name,
       COUNT(date) AS workouts,
       SUM(calories) AS calories
FROM daily_workouts
GROUP BY id;
```

Этот механизм работает более эффективно, будучи скрытым, так как GROUP BY должно выполняться только для одного столбца.

Чтобы компенсировать исключение имени из предложения GROUP BY, можно заметить, что к столбцу name в предложении SELECT была применена произвольная агрегатная функция (MAX). Поскольку в каждой группе идентификаторов существует только одно значение имени, MAX(name) просто вернет имя, связанное с каждым идентификатором.

Агрегирование строк в одно значение или список

В предложении GROUP BY необходимо указать, как должны быть агрегированы строки данных в каждой группе:

- *агрегатная функция для объединения строк в одно значение*: COUNT, SUM, MIN, MAX и AVG;
- *агрегатная функция для объединения строк в список* (показана в примере таблицы): GROUP_CONCAT и другие, перечисленные в табл. 8.3 (см. ниже).

Рассмотрим пример таблицы:

```
SELECT * FROM workouts;

+------+----------+
| name | calories |
+------+----------+
| ally |       80 |
| ally |       75 |
| ally |       90 |
| jess |      100 |
| jess |       92 |
+------+----------+
```

Используем GROUP_CONCAT в MySQL, чтобы создать список калорий:

```
SELECT name,
       GROUP_CONCAT(calories) AS calories_list
FROM workouts
GROUP BY name;

+------+---------------+
| name | calories_list |
+------+---------------+
| ally | 80,75,90      |
| jess | 100,92        |
+------+---------------+
```

Функция GROUP_CONCAT в каждой РСУБД имеет различия. В табл. 8.3 приведен синтаксис, поддерживаемый каждой РСУБД.

Таблица 8.3. Агрегирование строк в список в каждой РСУБД

РСУБД	Код	Разделитель по умолчанию
MySQL	`GROUP_CONCAT(calories)` `GROUP_CONCAT(calories SEPARATOR ',')`	Запятая
Oracle	`LISTAGG(calories)` `LISTAGG(calories, ',')`	Нет значения
PostgreSQL	`ARRAY_AGG(calories)`	Запятая
SQL Server	`STRING_AGG(calories, ',')`	Требуется разделитель
SQLite	`GROUP_CONCAT(calories)` `GROUP_CONCAT(calories, ',')`	Запятая

В MySQL, Oracle и SQLite разделитель (`','`) является необязательным. PostgreSQL не принимает разделитель, а SQL Server требует его наличия.

Кроме того, можно вернуть отсортированный список или список уникальных значений.

В табл. 8.4 приведен синтаксис, поддерживаемый каждой РСУБД.

Таблица 8.4. Возврат отсортированного списка или списка уникальных значений в каждой РСУБД

РСУБД	Сортированный список	Уникальный список
MySQL	`GROUP_CONCAT(calories ORDER BY calories)`	`GROUP_CONCAT(DISTINCT calories)`
Oracle	`LISTAGG(calories) WITHIN GROUP (ORDER BY calories)`	`LISTAGG(DISTINCT calories)`
PostgreSQL	`ARRAY_AGG(calories ORDER BY calories)`	`ARRAY_AGG(DISTINCT calories)`
SQL Server	`STRING_AGG(calories, ',') WITHIN GROUP (ORDER BY calories)`	Не поддерживается
SQLite	Не поддерживается	`GROUP_CONCAT(DISTINCT calories)`

ROLLUP, CUBE и GROUPING SETS

В дополнение к предложению GROUP BY можно добавить ключевые слова ROLLUP, CUBE или GROUPING SETS, позволяющие вносить дополнительную итоговую информацию.

В этой таблице перечислены пять покупок, совершенных в течение трех месяцев:

```
SELECT * FROM spendings;

 YEAR  MONTH  AMOUNT
----- ------ -------
 2019      1      20
 2019      1      30
 2019      1      42
 2020      2      37
 2020      2     100
```

Примеры, приведенные в этом подразделе, основаны на этом примере GROUP BY, который возвращает ежемесячные суммарные расходы:

```
SELECT year, month,
       SUM(amount) AS total
FROM spendings
GROUP BY year, month
ORDER BY year, month;

 YEAR  MONTH  TOTAL
----- ------ ------
2019      1     50
2020      1     42
2020      2    137
```

ROLLUP

MySQL, Oracle, PostgreSQL и SQL Server поддерживают функцию ROLLUP, которая расширяет GROUP BY за счет добавления дополнительных строк, предназначенных для показа промежуточных итогов и общего итога.

Используем ROLLUP для отображения годовых и суммарных расходов.

Строки 2019, 2020 и суммарных расходов добавляются с помощью ROLLUP:

```
SELECT year, month,
       SUM(amount) AS total
FROM spendings
GROUP BY ROLLUP(year, month)
ORDER BY year, month;

 YEAR  MONTH  TOTAL
----- ------ ------
 2019      1     50
 2019            50 -- 2019 spendings
 2020      1     42
 2020      2    137
 2020          179 -- 2020 spendings
               229 -- Total spendings
```

Приведенный выше синтаксис работает в Oracle, PostgreSQL и SQL Server. Синтаксис в MySQL выглядит так: `GROUP BY year, month WITH ROLLUP` — и работает в SQL Server.

CUBE

Oracle, PostgreSQL и SQL Server поддерживают функцию CUBE, которая расширяет ROLLUP за счет добавления дополнительных строк, в которых показываются все возможные комбинации столбцов, по которым производится группировка, а также общий итог.

С помощью CUBE также можно отображать ежемесячные расходы (один месяц за несколько лет). Строки расходов за январь и февраль вставляются при добавлении CUBE:

```
SELECT year, month,
       SUM(amount) AS total
FROM spendings
GROUP BY CUBE(year, month)
ORDER BY year, month;

 YEAR  MONTH  TOTAL
----- ------ ------
 2019      1     50
 2019            50
```

```
2020      1     42
2020      2    137
2020           179
          1     92 -- January spendings
          2    137 -- February spendings
               229
```

Приведенный выше синтаксис работает в Oracle, PostgreSQL и SQL Server. Кроме того, SQL Server поддерживает синтаксис `GROUP BY year, month WITH CUBE`.

GROUPING SETS

Oracle, PostgreSQL и SQL Server поддерживают `GROUPING SETS`, позволяющие указать конкретные группы, которые необходимо отобразить.

Эти данные представляют собой подмножество результатов, сгенерированных функцией `CUBE`, и содержат группировки лишь по одному столбцу. В данном случае возвращаются только общие годовые и общие ежемесячные расходы:

```
SELECT year, month,
       SUM(amount) AS total
FROM spendings
GROUP BY GROUPING SETS(year, month)
ORDER BY year, month;

 YEAR  MONTH  TOTAL
----- ------ ------
2019             50
2019            179
          1      92
          2     137
```

Оконные функции

Оконная функция (в Oracle называется *аналитической*) аналогична агрегатной тем, что обе они выполняют вычисления над строками данных. Разница заключается в том, что агрегатная функция возвращает одно значение, а оконная — значение для каждой строки данных.

В этой таблице перечислены сотрудники с указанием их ежемесячных продаж. Показанные ниже запросы используют ее, чтобы показать разницу между агрегатной и оконной функциями.

```
SELECT * FROM sales;
```

```
+-------+-------+-------+
| name  | month | sales |
+-------+-------+-------+
| David |     3 |     2 |
| David |     4 |    11 |
| Laura |     3 |     3 |
| Laura |     4 |    14 |
| Laura |     5 |     7 |
| Laura |     6 |     1 |
+-------+-------+-------+
```

Агрегатная функция

Функция `SUM()` является агрегатной. Этот запрос суммирует продажи для каждого человека и возвращает каждое имя вместе с его значением `total_sales`:

```
SELECT name,
       SUM(sales) AS total_sales
FROM sales
GROUP BY name;
```

```
+-------+-------------+
| name  | total_sales |
+-------+-------------+
| David |          13 |
| Laura |          25 |
+-------+-------------+
```

Оконная функция

Функция `ROW_NUMBER() OVER (PARTITION BY name ORDER BY month)` является оконной. В части этого запроса, выделенной жирным шрифтом, для каждого человека генерируется номер строки, обозначающий первый месяц, второй и т. д., в которые этот

человек что-то продал. Запрос возвращает каждую строку вместе со значением `sale_month`.

```
SELECT name,
       ROW_NUMBER() OVER (PARTITION BY name
       ORDER BY month) AS sale_month
FROM sales;

+-------+------------+
| name  | sale_month |
+-------+------------+
| David |          1 |
| David |          2 |
| Laura |          1 |
| Laura |          2 |
| Laura |          3 |
| Laura |          4 |
+-------+------------+
```

РАЗБОР ОКОННОЙ ФУНКЦИИ

`ROW_NUMBER() OVER (PARTITION BY name ORDER BY month)`

Окно представляет собой группу строк. В предыдущем примере было два окна. Окно имени `David` состояло из двух строк, а окно имени `Laura` — из четырех.

- `ROW_NUMBER()` — функция, которую необходимо применить к каждому окну. Она обязательна. Другие широко используемые функции содержат `RANK()`, `FIRST_VALUE()`, `LAG()` и т. д.
- `OVER` означает, что вы указываете оконную функцию. Это предложение обязательно.
- `PARTITION BY name`. Здесь указывается, как вы хотите разделить данные на окна. Они могут быть разделены по одному или нескольким столбцам. Этот параметр необязателен. Если он исключен, то окно представляет собой всю таблицу.
- `ORDER BY month`. В этом предложении указывается, как должно быть отсортировано каждое окно перед применением функции. В MySQL, PostgreSQL и SQLite это предложение необязательно, в отличие от Oracle и SQL Server.

В следующих подразделах приведены примеры практического использования оконных функций.

Ранжирование строк в таблице

Для добавления номера строки в каждую строку таблицы можно использовать функцию `ROW_NUMBER()`, `RANK()` или `DENSE_RANK()`.

В этой таблице показано количество детей, получивших популярные имена:

```
SELECT * FROM baby_names;
```

```
+--------+--------+--------+
| gender | name   | babies |
+--------+--------+--------+
| F      | Emma   |     92 |
| F      | Mia    |     88 |
| F      | Olivia |    100 |
| M      | Liam   |    105 |
| M      | Mateo  |     95 |
| M      | Noah   |    110 |
+--------+--------+--------+
```

Выполним два следующих запроса:

- ранжирование имен по популярности;
- ранжирование имен по популярности для каждого пола.

Ранжируем имена по популярности:

```
SELECT gender, name,
       ROW_NUMBER() OVER (
       ORDER BY babies DESC) AS popularity
FROM baby_names;
```

```
+--------+--------+------------+
| gender | name   | popularity |
+--------+--------+------------+
| M      | Noah   |          1 |
| M      | Liam   |          2 |
| F      | Olivia |          3 |
| M      | Mateo  |          4 |
| F      | Emma   |          5 |
| F      | Mia    |          6 |
+--------+--------+------------+
```

Ранжируем имена по популярности для каждого пола:

```
SELECT gender, name,
       ROW_NUMBER() OVER (PARTITION BY gender
       ORDER BY babies DESC) AS popularity
FROM baby_names;
```

```
+--------+--------+------------+
| gender | name   | popularity |
+--------+--------+------------+
| F      | Olivia |          1 |
| F      | Emma   |          2 |
| F      | Mia    |          3 |
| M      | Noah   |          1 |
| M      | Liam   |          2 |
| M      | Mateo  |          3 |
+--------+--------+------------+
```

ROW_NUMBER В СРАВНЕНИИ С RANK И DENSE_RANK

Есть три подхода к добавлению номеров строк. Каждый из них по-своему решает проблему связей.

ROW_NUMBER разрывает связь:

```
NAME      BABIES   POPULARITY
-------  -------  ------------
Olivia        99             1
Emma          80             2
Sophia        80             3
Mia           75             4
```

RANK сохраняет ее:

```
NAME      BABIES   POPULARITY
-------  -------  ------------
Olivia        99             1
Emma          80             2
Sophia        80             2
Mia           75             4
```

DENSE_RANK сохраняет связь и не пропускает номера строк:

```
NAME      BABIES   POPULARITY
-------  -------  ------------
Olivia        99             1
Emma          80             2
Sophia        80             2
Mia           75             3
```

Возврат первого значения в каждой группе

С помощью FIRST_VALUE и LAST_VALUE можно вернуть первую и последнюю строки окна соответственно.

В показанных ниже запросах этот двухэтапный процесс разбивается на две части, чтобы можно было получить наиболее популярные имена для каждого пола.

- *Этап 1. Отображение наиболее популярных имен для каждого пола.*

```
SELECT gender, name, babies,
       FIRST_VALUE(name) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS top_name
FROM baby_names;
```

```
+--------+--------+--------+----------+
| gender | name   | babies | top_name |
+--------+--------+--------+----------+
| F      | Olivia |    100 | Olivia   |
| F      | Emma   |     92 | Olivia   |
| F      | Mia    |     88 | Olivia   |
| M      | Noah   |    110 | Noah     |
| M      | Liam   |    105 | Noah     |
| M      | Mateo  |     95 | Noah     |
+--------+--------+--------+----------+
```

Используем полученный результат в качестве подзапроса для следующего шага, который выполняет фильтрацию по подзапросу.

- *Этап 2. Возврат только двух строк, содержащих наиболее популярные имена.*

```
SELECT * FROM

(SELECT gender, name, babies,
       FIRST_VALUE(name) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS top_name
FROM baby_names) AS top_name_table

WHERE name = top_name;
```

```
+--------+--------+--------+----------+
| gender | name   | babies | top_name |
+--------+--------+--------+----------+
| F      | Olivia |    100 | Olivia   |
| M      | Noah   |    110 | Noah     |
+--------+--------+--------+----------+
```

В Oracle необходимо исключить часть `AS top_name_table`.

Возврат второго значения в каждой группе

С помощью `NTH_VALUE` можно вернуть определенный номер ранга в каждом окне. SQL Server не поддерживает `NTH_VALUE`. Вместо этого обратитесь к коду, приведенному в следующем подразделе — «Возврат первых двух значений в каждой группе», но возвращайте только второе значение.

В показанных ниже запросах этот двухступенчатый процесс разбивается на две части, чтобы можно было получить второе по популярности имя для каждого пола.

- *Этап 1. Возврат второго по популярности имени для каждого пола.*

```
SELECT gender, name, babies,
       NTH_VALUE(name, 2) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS second_name
FROM baby_names;
```

```
+--------+--------+--------+-------------+
| gender | name   | babies | second_name |
+--------+--------+--------+-------------+
| F      | Olivia |    100 | NULL        |
| F      | Emma   |     92 | Emma        |
| F      | Mia    |     88 | Emma        |
| M      | Noah   |    110 | NULL        |
| M      | Liam   |    105 | Liam        |
| M      | Mateo  |     95 | Liam        |
+--------+--------+--------+-------------+
```

Второй параметр `NTH_VALUE(name, 2)` определяет второе значение в окне. Этот параметр может быть любым целым положительным числом.

Используем полученный результат в качестве подзапроса для следующего шага, который выполняет фильтрацию по подзапросу.

- *Этап 2. Возврат только двух строк, содержащих вторые по популярности имена.*

```
SELECT * FROM

(SELECT gender, name, babies,
       NTH_VALUE(name, 2) OVER (PARTITION BY gender
       ORDER BY babies DESC) AS second_name
FROM baby_names) AS second_name_table

WHERE name = second_name;

+--------+--------+--------+-------------+
| gender | name   | babies | second_name |
+--------+--------+--------+-------------+
| F      | Emma   |     92 | Emma        |
| M      | Liam   |    105 | Liam        |
+--------+--------+--------+-------------+
```

В Oracle необходимо исключить часть `AS second_name_table`.

Возврат первых двух значений в каждой группе

Используем `ROW_NUMBER` в подзапросе, чтобы вернуть несколько номеров рангов в каждой группе.

В показанных ниже запросах этот двухступенчатый процесс разбивается на две части, чтобы можно было получить первые и вторые по популярности имена для каждого пола.

- *Этап 1. Отображение рейтинга популярности для каждого пола.*

```
SELECT gender, name, babies,
       ROW_NUMBER() OVER (PARTITION BY gender
       ORDER BY babies DESC) AS popularity
FROM baby_names;
```

```
+--------+--------+--------+------------+
| gender | name   | babies | popularity |
+--------+--------+--------+------------+
| F      | Olivia |    100 |          1 |
| F      | Emma   |     92 |          2 |
| F      | Mia    |     88 |          3 |
| M      | Noah   |    110 |          1 |
| M      | Liam   |    105 |          2 |
| M      | Mateo  |     95 |          3 |
+--------+--------+--------+------------+
```

Используем полученный результат в качестве подзапроса для следующего шага, который выполняет фильтрацию по подзапросу.

- *Этап 2. Отфильтруем строки, содержащие ранги 1 и 2.*

```
SELECT * FROM

(SELECT gender, name, babies,
       ROW_NUMBER() OVER (PARTITION BY gender
       ORDER BY babies DESC) AS popularity
FROM baby_names) AS popularity_table

WHERE popularity IN (1,2);

+--------+--------+--------+------------+
| gender | name   | babies | popularity |
+--------+--------+--------+------------+
| F      | Olivia |    100 |          1 |
| F      | Emma   |     92 |          2 |
| M      | Noah   |    110 |          1 |
| M      | Liam   |    105 |          2 |
+--------+--------+--------+------------+
```

В Oracle необходимо исключить часть `AS popularity_table`.

Возврат значения предыдущей строки

С помощью функций `LAG` и `LEAD` можно просмотреть определенное количество строк позади и впереди соответственно.

Для возврата предыдущей строки используем функцию `LAG`:

```
SELECT gender, name, babies,
       LAG(name) OVER (PARTITION BY gender
```

```
      ORDER BY babies DESC) AS prior_name
FROM baby_names;

+--------+--------+--------+------------+
| gender | name   | babies | prior_name |
+--------+--------+--------+------------+
| F      | Olivia |    100 | NULL       |
| F      | Emma   |     92 | Olivia     |
| F      | Mia    |     88 | Emma       |
| M      | Noah   |    110 | NULL       |
| M      | Liam   |    105 | Noah       |
| M      | Mateo  |     95 | Liam       |
+--------+--------+--------+------------+
```

Используем функцию `LAG(name, 2, 'No name')`, чтобы вернуть имена из двух предыдущих строк и заменить значения `NULL` на `No name`:

```
SELECT gender, name, babies,
       LAG(name, 2, 'No name')
       OVER (PARTITION BY gender
       ORDER BY babies DESC) AS prior_name_2
FROM baby_names;

+--------+--------+--------+--------------+
| gender | name   | babies | prior_name_2 |
+--------+--------+--------+--------------+
| F      | Olivia |    100 | No name      |
| F      | Emma   |     92 | No name      |
| F      | Mia    |     88 | Olivia       |
| M      | Noah   |    110 | No name      |
| M      | Liam   |    105 | No name      |
| M      | Mateo  |     95 | Noah         |
+--------+--------+--------+--------------+
```

Функции `LAG` и `LEAD` принимают по три аргумента: `LAG(name, 2, 'None')`.

- `name` — значение, которое вы хотите вернуть. Оно является обязательным.
- `2` — обозначает смещение строки. Оно необязательно и по умолчанию равно 1.
- `'No name'` — имя, которое будет возвращено в случае отсутствия значения. Оно является необязательным и по умолчанию имеет значение `NULL`.

Расчет скользящего среднего

Для расчета скользящего среднего используем комбинацию функции AVG и условия ROWS BETWEEN.

Рассмотрим пример таблицы:

```
SELECT * FROM sales;
```

```
+-------+-------+-------+
| name  | month | sales |
+-------+-------+-------+
| David |     1 |     2 |
| David |     2 |    11 |
| David |     3 |     6 |
| David |     4 |     8 |
| Laura |     1 |     3 |
| Laura |     2 |    14 |
| Laura |     3 |     7 |
| Laura |     4 |     1 |
| Laura |     5 |    20 |
+-------+-------+-------+
```

Для каждого человека найдем трехмесячное скользящее среднее значение продаж за два месяца, предшествующих текущему:

```
SELECT name, month, sales,
       AVG(sales) OVER (PARTITION BY name
       ORDER BY month
       ROWS BETWEEN 2 PRECEDING AND
       CURRENT ROW) three_month_ma
FROM sales;
```

```
+-------+-------+-------+----------------+
| name  | month | sales | three_month_ma |
+-------+-------+-------+----------------+
| David |     1 |     2 |         2.0000 |
| David |     2 |    11 |         6.5000 |
| David |     3 |     6 |         6.3333 |
| David |     4 |     8 |         8.3333 |
| Laura |     1 |     3 |         3.0000 |
| Laura |     2 |    14 |         8.5000 |
| Laura |     3 |     7 |         8.0000 |
| Laura |     4 |     1 |         7.3333 |
| Laura |     5 |    20 |         9.3333 |
+-------+-------+-------+----------------+
```

В примере выше рассматриваются две строки, предшествующие текущей строке:

```
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

Вы можете просмотреть и следующие строки, используя ключевое слово FOLLOWING:

```
ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING
```

Такие диапазоны иногда называют скользящими окнами.

Вычисление промежуточного итога

Чтобы выполнить это действие, используем комбинацию функции SUM и предложения ROWS BETWEEN UNBOUNDED.

Для каждого сотрудника найдем промежуточную сумму продаж до текущего месяца:

```
SELECT name, month, sales,
       SUM(sales) OVER (PARTITION BY name
       ORDER BY month
       ROWS BETWEEN UNBOUNDED PRECEDING AND
       CURRENT ROW) running_total
FROM sales;
```

```
+-------+-------+-------+---------------+
| name  | month | sales | running_total |
+-------+-------+-------+---------------+
| David |     1 |     2 |             2 |
| David |     2 |    11 |            13 |
| David |     3 |     6 |            19 |
| David |     4 |     8 |            27 |
| Laura |     1 |     3 |             3 |
| Laura |     2 |    14 |            17 |
| Laura |     3 |     7 |            24 |
| Laura |     4 |     1 |            25 |
| Laura |     5 |    20 |            45 |
+-------+-------+-------+---------------+
```

Здесь мы вычисляем промежуточную сумму для каждого человека. Чтобы вычислить такую сумму по всей таблице, можно удалить часть кода PARTITION BY name.

ROWS В СРАВНЕНИИ С RANGE

Альтернативой ROWS BETWEEN является RANGE BETWEEN. В запросе ниже вычисляется промежуточная сумма продаж, совершенных всеми сотрудниками; используются ключевые слова ROWS и RANGE:

```
SELECT month, name,
  SUM(sales) OVER (ORDER BY month ROWS BETWEEN
  UNBOUNDED PRECEDING AND CURRENT ROW) rt_rows,
  SUM(sales) OVER (ORDER BY month RANGE BETWEEN
  UNBOUNDED PRECEDING AND CURRENT ROW) rt_range
FROM sales;
```

```
+-------+-------+----------+-----------+
| month | name  | rt_rows  | rt_range  |
+-------+-------+----------+-----------+
|     1 | David |        2 |         5 |
|     1 | Laura |        5 |         5 |
|     2 | David |       16 |        30 |
|     2 | Laura |       30 |        30 |
|     3 | David |       36 |        43 |
|     3 | Laura |       43 |        43 |
|     4 | David |       51 |        52 |
|     4 | Laura |       52 |        52 |
|     5 | Laura |       72 |        72 |
+-------+-------+----------+-----------+
```

Разница между этими ключевыми словами заключается в том, что RANGE будет возвращать одно и то же значение промежуточной суммы для каждого месяца (поскольку данные были упорядочены по месяцам), тогда как ROWS будет иметь разное значение промежуточной суммы для каждой строки.

Операции PIVOT и UNPIVOT

Oracle и SQL Server поддерживают операции `PIVOT` и `UNPIVOT`. Операция `PIVOT` берет один столбец и разбивает его на несколько, а `UNPIVOT` берет несколько столбцов и объединяет их.

Разбиение значений столбца на несколько столбцов

Представьте, что у вас есть таблица, в которой каждая строка содержит имя человека, а затем название фрукта, который человек съел в этот день. Вы хотите взять столбец фруктов и создать отдельный столбец для каждого фрукта.

Рассмотрим пример таблицы:

```
SELECT * FROM fruits;

+------+-------+--------------+
| id   | name  | fruit        |
+------+-------+--------------+
|    1 | Henry | strawberries |
|    2 | Henry | grapefruit   |
|    3 | Henry | watermelon   |
|    4 | Lily  | strawberries |
|    5 | Lily  | watermelon   |
|    6 | Lily  | strawberries |
|    7 | Lily  | watermelon   |
+------+-------+--------------+
```

Ожидаемый результат:

```
+-------+--------------+------------+------------+
| name  | strawberries | grapefruit | watermelon |
+-------+--------------+------------+------------+
| Henry |            1 |          1 |          1 |
| Lily  |            2 |          0 |          2 |
+-------+--------------+------------+------------+
```

Используем операцию `PIVOT` в Oracle и SQL Server:

```
-- Oracle
SELECT *
FROM fruits
PIVOT
(COUNT(id) FOR fruit IN ('strawberries',
                'grapefruit', 'watermelon'));

-- SQL Server
SELECT *
```

```
FROM fruits
PIVOT
(COUNT(id) FOR fruit IN ([strawberries],
                [grapefruit], [watermelon])
) AS fruits_pivot;
```

В предложении PIVOT имеются ссылки на столбцы id и fruit, но нет ссылок на столбец name. Поэтому в итоговом результате столбец name останется в качестве отдельного, а данные о каждом фрукте будут преобразованы в новый столбец.

Значения таблицы представляют собой подсчет количества строк исходной таблицы, содержащих каждую конкретную комбинацию имени человека и фрукта.

АЛЬТЕРНАТИВА PIVOT — CASE

В MySQL, PostgreSQL и SQLite используется ручной способ выполнения операции PIVOT. Он заключается в применении оператора CASE, поскольку эти РСУБД не поддерживают PIVOT.

```
SELECT name,
       SUM(CASE WHEN fruit = 'strawberries'
            THEN 1 ELSE 0 END) AS strawberries,
       SUM(CASE WHEN fruit = 'grapefruit'
            THEN 1 ELSE 0 END) AS grapefruit,
       SUM(CASE WHEN fruit = 'watermelon'
            THEN 1 ELSE 0 END) AS watermelon
FROM fruits
GROUP BY name
ORDER BY name;
```

Перечисление значений нескольких столбцов в одном

Представьте, что у вас есть таблица, где каждая строка представляет собой имя человека, за которым следует несколько столбцов, содержащих данные о его любимых фруктах. Вы хотите изменить порядок данных таким образом, чтобы все данные о фруктах были в одном столбце.

Рассмотрим пример таблицы:

```
SELECT * FROM favorite_fruits;
```

```
+----+-------+-----------+-----------+-----------+
| id | name  | fruit_one | fruit_two | fruit_thr |
+----+-------+-----------+-----------+-----------+
|  1 | Anna  | apple     | banana    |           |
|  2 | Barry | raspberry |           |           |
|  3 | Liz   | lemon     | lime      | orange    |
|  4 | Tom   | peach     | pear      | plum      |
+----+-------+-----------+-----------+-----------+
```

Ожидаемый результат:

```
+----+-------+-----------+------+
| id | name  | fruit     | rank |
+----+-------+-----------+------+
|  1 | Anna  | apple     |    1 |
|  1 | Anna  | banana    |    2 |
|  2 | Barry | raspberry |    1 |
|  3 | Liz   | lemon     |    1 |
|  3 | Liz   | lime      |    2 |
|  3 | Liz   | orange    |    3 |
|  4 | Tom   | peach     |    1 |
|  4 | Tom   | pear      |    2 |
|  4 | Tom   | plum      |    3 |
+----+-------+-----------+------+
```

Используем операцию `UNPIVOT` в Oracle и SQL Server:

```
-- Oracle
SELECT *
FROM favorite_fruits
UNPIVOT
(fruit FOR rank IN (fruit_one AS 1,
    fruit_two AS 2,     fruit_thr AS 3));

-- SQL Server
SELECT *
FROM favorite_fruits
UNPIVOT
(fruit FOR rank IN (fruit_one,
                    fruit_two,
                    fruit_thr)
) AS fruit_unpivot
WHERE fruit <> '';
```

Операция UNPIVOT берет столбцы fruit_one, fruit_two и fruit_thr и образует из них один столбец под названием fruit.

Как только это будет сделано, можно использовать типичный оператор SELECT для извлечения исходных столбцов id и name, а также вновь созданного столбца fruit.

АЛЬТЕРНАТИВА UNPIVOT — UNION ALL

Ручным способом выполнения UNPIVOT является использование вместо него UNION ALL в MySQL, PostgreSQL и SQLite, поскольку эти РСУБД не поддерживают UNPIVOT.

```
WITH all_fruits AS
(SELECT id, name,
        fruit_one as fruit,
        1 AS rank
FROM favorite_fruits
UNION ALL
SELECT id, name,
       fruit_two as fruit,
       2 AS rank
FROM favorite_fruits
UNION ALL
SELECT id, name,
       fruit_three as fruit,
       3 AS rank
FROM favorite_fruits)

SELECT *
FROM all_fruits
WHERE fruit <> ''
ORDER BY id, name, fruit;
```

MySQL не поддерживает вставку константы в столбец внутри запроса (1 AS rank, 2 AS rank и 3 AS rank). Удалите эти строки, чтобы код мог работать.

ГЛАВА 9

Работа с несколькими таблицами и запросами

В этой главе рассматриваются способы объединения нескольких таблиц путем их соединения или использования операций UNION, а также работа с несколькими запросами с помощью обобщенных табличных выражений.

В табл. 9.1 приведены описания и примеры кода трех концепций, рассматриваемых в этой главе.

Таблица 9.1. Работа с несколькими таблицами и запросами

Концепция	Описание	Пример кода

Соединение таблиц (операции JOIN)	Объединение столбцов двух таблиц на основе совпадающих строк	`SELECT c.id, l.city`
		`FROM customers c`
		`  INNER JOIN loc l`
		`  ON c.lid = l.id;`

Операции UNION	Объединение строк двух таблиц на основе совпадающих столбцов	`SELECT name, city`
		`FROM employees;`
		`UNION`
		`SELECT name, city`
		`FROM customers;`

Продолжение ⇗

Таблица 9.1 *(продолжение)*

Концепция	Описание	Пример кода

Обобщенное табличное выражение, или CTE (Common Table Expressions)	Временное сохранение результирующего набора данных запроса, чтобы другой запрос мог на него ссылаться. Содержит также рекурсивные и иерархические запросы	`WITH my_cte AS (`
		`  SELECT name,`
		`  SUM(order_id)`
		`  AS num_orders`
		`  FROM customers`
		`  GROUP BY name)`
		
		`SELECT MAX(num_orders)`
		`FROM my_cte;`

Соединение таблиц

В SQL *соединение* означает объединение данных из нескольких таблиц в рамках одного запроса. В следующих двух таблицах перечислены штат, в котором живет человек, и домашние животные, которые у него есть:

```
-- штаты -- домашние животные
+------+-------+    +------+------+
| name | state |    | name | pet  |
+------+-------+    +------+------+
| Ada  | AZ    |    | Deb  | dog  |
| Deb  | DE    |    | Deb  | duck |
+------+-------+    | Pat  | pig  |
                    +------+------+
```

Используем предложение `JOIN` для соединения двух таблиц в одну:

```
SELECT *
FROM states s INNER JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+
```

В результирующую таблицу включены строки только для Deb, поскольку она есть в обеих таблицах.

Два левых столбца взяты из таблицы `states`, а два правых — из `pets`. На столбцы в выводе можно ссылаться, используя псевдонимы `s.name`, `s.state`, `p.name` и `p.pet`.

РАЗБОР ФРАЗЫ JOIN

```
states s INNER JOIN pets p ON s.name = p.name
```

- *Таблицы* (`states`, `pets`). Таблицы, которые мы хотим объединить.
- *Псевдонимы* (`s`, `p`) *для таблиц.* Они необязательны, но рекомендуются для простоты. Без псевдонимов предложение `ON` можно было бы записать как `states.name = pets.name`.
- *Тип соединения* (`INNER JOIN`). Предложение `INNER` указывает, что должны быть возвращены только совпадающие строки. Если написано только `JOIN`, то по умолчанию используется `INNER JOIN`. Другие типы соединений приведены в табл. 9.2 (см. ниже).
- *Условие соединения* (`ON s.name = p.name`).

  Условие, которое должно быть истинным для того, чтобы две строки считались совпадающими. Чаще всего используется операция равенства (=), но могут применяться и другие, в том числе неравенство (`!=` или `<>`), `>`, `<`, `BETWEEN` и т. д.

В дополнение к `INNER JOIN` в табл. 9.2 перечислены различные типы соединений в SQL. В запросе ниже показан общий формат соединения таблиц:

```
SELECT *
FROM states s [JOIN_TYPE] pets p
     ON s.name = p.name;
```

Замените выделенную жирным шрифтом часть `[JOIN_TYPE]` на ключевые слова в столбце «Ключевое слово», чтобы получить результаты, показанные в столбце «Результирующие строки». Для типа соединения `CROSS JOIN` исключите предложение `ON`, чтобы получить результаты, показанные в табл. 9.2.

Таблица 9.2. Способы соединения таблиц

Ключевое слово	Описание	Результирующие строки

`JOIN`	По умолчанию используется `INNER JOIN`	`nm | st | nm | pt`
		`Deb | DE | Deb | dog`
		`Deb | DE | Deb | duck`

`INNER JOIN`	Возвращает общие строки	`nm | st | nm | pt`
		`Deb | DE | Deb | dog`
		`Deb | DE | Deb | duck`

`LEFT JOIN`	Возвращает строки в левой таблице и соответствующие им строки в другой таблице	`nm | st | nm | pt`
		`Ada | AZ | NULL | NULL`
		`Deb | DE | Deb | dog`
		`Deb | DE | Deb | duck`

`RIGHT JOIN`	Возвращает строки в правой таблице и соответствующие им строки в другой таблице	`nm | st | nm | pt`
		`Deb | DE | Deb | dog`
		`Deb | DE | Deb | duck`
		`NULL | NULL | Pat | pig`

`FULL OUTER JOIN`	Возвращает строки в обеих таблицах	`nm | st | nm | pt`
		`Ada | AZ | NULL | NULL`
		`Deb | DE | Deb | dog`
		`Deb | DE | Deb | duck`
		`NULL | NULL | Pat | pig`

`CROSS JOIN`	Возвращает все комбинации строк в двух таблицах	`nm | st | nm | pt`
		`Ada | AZ | Deb | dog`
		`Ada | AZ | Deb | duck`
		`Ada | AZ | Pat | pig`
		`Deb | DE | Deb | dog`
		`Deb | DE | Deb | duck`
		`Deb | DE | Pat | pig`

В дополнение к соединению таблиц с помощью стандартного синтаксиса `JOIN ... ON ...` в табл. 9.3 перечислены другие способы соединения таблиц в SQL.

Таблица 9.3. Синтаксис для соединения таблиц

Тип	Описание	Код
Синтаксис JOIN ... ON ...	Наиболее используемый синтаксис соединения, который работает с INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN и CROSS JOIN	`SELECT * FROM states s INNER JOIN pets p ON s.name = p.name;`
Использование предложения USING	Используйте вместо предложения ON, если имена столбцов, которые подвергаются соединению, совпадают	`SELECT * FROM states INNER JOIN pets USING (name);`
Использование предложения NATURAL JOIN	Используйте вместо INNER JOIN, если имена всех столбцов, которые подвергаются соединению, совпадают	`SELECT * FROM states NATURAL JOIN pets;`
Старый синтаксис соединения (Old Join)	Возвращает все комбинации строк в двух таблицах. Эквивалент CROSS JOIN	`SELECT * FROM states s, pets p`

Самосоединение (Self Join)	Используйте старый или новый синтаксис JOIN для возврата всех комбинаций строк таблицы, соединенной с самой собой	`SELECT * FROM states s1, states s2 WHERE s1.region = s2.region;`
		
		`SELECT * FROM states s1 INNER JOIN states s2 WHERE s1.region = s2.region;`

В следующих подразделах подробно описаны концепции, приведенные в табл. 9.2 и 9.3.

Основы соединения и INNER JOIN

В этом разделе рассматривается концепция работы соединения, а также базовый синтаксис соединения, в котором используется предложение `INNER JOIN`.

Основы соединения

Соединение таблиц можно представить в виде двух этапов:

1) отображения всех комбинаций строк в таблицах;
2) фильтра по строкам, имеющим совпадающие значения.

Вот две таблицы, которые мы хотели бы соединить:

```
-- штаты             -- домашние животные
+------+-------+     +------+------+
| name | state |     | name | pet  |
+------+-------+     +------+------+
| Ada  | AZ    |     | Deb  | dog  |
| Deb  | DE    |     | Deb  | duck |
+------+-------+     | Pat  | pig  |
                     +------+------+
```

- *Этап 1. Отобразить все комбинации строк в таблицах.* Если в предложении `FROM` указать список двух имен таблиц, то будут возвращены все возможные комбинации строк из этих таблиц.

```
SELECT *
FROM states, pets;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Ada  | AZ    | Deb  | dog  |
| Deb  | DE    | Deb  | dog  |
| Ada  | AZ    | Deb  | duck |
| Deb  | DE    | Deb  | duck |
| Ada  | AZ    | Pat  | pig  |
| Deb  | DE    | Pat  | pig  |
+------+-------+------+------+
```

Синтаксис `FROM states, pets` является старым способом выполнения соединения в SQL. Более современный способ сделать то же самое — использовать `CROSS JOIN`.

- *Этап 2. Отфильтровать строки с совпадающими именами.* Скорее всего, вам нужно отобразить не все комбинации строк в двух таблицах, а только те, в которых столбцы имен обеих таблиц совпадают.

```
SELECT *
FROM states s, pets p
WHERE s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+
```

Строка `Deb/DE` указана дважды, так как соответствует двум значениям `Deb` в таблице `pets`.

Приведенный код эквивалентен `INNER JOIN`.

Описанный двухэтапный процесс является чисто концептуальным. Базы данных редко совершают перекрестное соединение при выполнении `JOIN`, вместо этого они используют более оптимизированные способы.

Однако если вы будете мыслить в этих концептуальных терминах, это поможет вам правильно составлять запросы на соединение и понимать их результаты.

INNER JOIN

Чаще всего две таблицы соединяются с помощью предложения `INNER JOIN`, которое возвращает строки, находящиеся в обеих таблицах.

- *Используем* `INNER JOIN`, *чтобы вернуть только данные о людях, находящихся в обеих таблицах.*

```
SELECT *
FROM states s INNER JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+
```

- *Соединение нескольких таблиц.* Это можно сделать, добавив наборы ключевых слов `JOIN ... ON ...`:

```
SELECT *
FROM states s
     INNER JOIN pets p
        ON s.name = p.name
     INNER JOIN lunch l
        ON s.name = l.name;
```

- *Соединение по нескольким столбцам.* Это можно сделать, добавив дополнительные условия в предложение `ON`. Представьте, что хотите соединить эти таблицы по столбцам `name` и `age`:

```
-- states_ages            -- pets_ages
+------+-------+-----+    +------+-----+-----+
| name | state | age |    | name | pet | age |
+------+-------+-----+    +------+-----+-----+
| Ada  | AK    |  25 |    | Ada  | ant |  30 |
| Ada  | AZ    |  30 |    | Pat  | pig |  45 |
+------+-------+-----+    +------+-----+-----+

SELECT *
FROM states_ages s INNER JOIN pets_ages p
     ON s.name = p.name
     AND s.age = p.age;

+------+-------+------+------+------+------+
| name | state | age  | name | pet  | age  |
+------+-------+------+------+------+------+
| Ada  | AZ    |   30 | Ada  | ant  |   30 |
+------+-------+------+------+------+------+
```

LEFT JOIN, RIGHT JOIN и FULL OUTER JOIN

Используйте предложения `LEFT JOIN`, `RIGHT JOIN` и `FULL OUTER JOIN` для соединения строк из двух таблиц, в том числе и тех, которые не содержатся в обеих таблицах.

LEFT JOIN

Используйте предложение `LEFT JOIN`, чтобы вернуть данные обо всех людях в таблице `states`. Данные о людях из этой таблицы, которых нет в таблице `pets`, возвращаются со значениями `NULL`.

```
SELECT *
FROM states s LEFT JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Ada  | AZ    | NULL | NULL |
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+
```

Предложение `LEFT JOIN` эквивалентно `LEFT OUTER JOIN`.

RIGHT JOIN

Используйте предложение `RIGHT JOIN`, чтобы вернуть данные обо всех людях в таблице `pets`. Данные о людях в этой таблице, которых нет в таблице `states`, возвращаются со значениями `NULL`.

```
SELECT *
FROM states s RIGHT JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
| NULL | NULL  | Pat  | pig  |
+------+-------+------+------+
```

Предложение `RIGHT JOIN` эквивалентно `RIGHT OUTER JOIN`.

SQLite не поддерживает `RIGHT JOIN`.

LEFT JOIN встречается гораздо чаще, чем RIGHT JOIN. Если необходимо выполнить RIGHT JOIN, то поменяйте местами две таблицы в предложении FROM и выполните LEFT JOIN.

FULL OUTER JOIN

Используйте предложение `FULL OUTER JOIN`, чтобы вернуть данные обо всех людях в таблицах `states` и `pets`. Недостающие значения из обеих таблиц возвращаются в виде значений `NULL`.

```
SELECT *
FROM states s FULL OUTER JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Ada  | AZ    | NULL | NULL |
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
| NULL | NULL  | Pat  | pig  |
+------+-------+------+------+
```

Предложение `FULL OUTER JOIN` эквивалентно `FULL JOIN`.

MySQL и SQLite не поддерживают `FULL OUTER JOIN`.

USING и NATURAL JOIN

При соединении таблиц, чтобы сэкономить на вводе текста, можно использовать предложения `USING` или `NATURAL JOIN` вместо стандартного синтаксиса `JOIN ... ON ...`.

USING

MySQL, Oracle, PostgreSQL и SQLite поддерживают предложение `USING`.

Чтобы соединить два столбца с одинаковыми именами, вместо предложения `ON` можно использовать `USING` — при условии, что соединение является эквисоединением (`=` в предложении `ON`).

```
-- С предложением ON
SELECT *
FROM states s INNER JOIN pets p
     ON s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+

-- Эквивалентное сокращение при использовании USING
SELECT *
FROM states INNER JOIN pets
     USING (name);

+------+-------+------+
| name | state | pet  |
+------+-------+------+
| Deb  | DE    | dog  |
| Deb  | DE    | duck |
+------+-------+------+
```

Разница между двумя запросами заключается в том, что первый возвращает четыре столбца, в том числе `s.name` и `p.name`, а второй — три столбца, поскольку два столбца `name` объединяются и называются просто `name`.

Естественное соединение NATURAL JOIN

MySQL, Oracle, PostgreSQL и SQLite поддерживают предложение `NATURAL JOIN`.

Вы можете использовать предложение NATURAL JOIN вместо синтаксиса INNER JOIN ... ON ... для соединения двух таблиц на основе всех столбцов с одинаковыми именами. Чтобы NATURAL JOIN можно было использовать, соединение должно быть эквисоединением (= в предложении ON).

```
-- С предложением INNER JOIN ... ON ... AND ...
SELECT *
FROM states_ages s INNER JOIN pets_ages p
     ON s.name = p.name
     AND s.age = p.age;

+------+-------+------+------+------+------+
| name | state | age  | name | pet  | age  |
+------+-------+------+------+------+------+
| Ada  | AZ    |   30 | Ada  | ant  |   30 |
+------+-------+------+------+------+------+

-- Эквивалентное сокращение при использовании NATURAL JOIN
SELECT *
FROM states_ages NATURAL JOIN pets_ages;

+------+------+-------+------+
| name | age  | state | pet  |
+------+------+-------+------+
| Ada  |   30 | AZ    | ant  |
+------+------+-------+------+
```

Разница между двумя запросами заключается в том, что первый возвращает шесть столбцов, в том числе s.name, s.age, p.name и p.age, а второй — четыре столбца, поскольку повторяющиеся столбцы name и age объединяются и называются просто name и age.

Будьте осторожны при использовании предложения NATURAL JOIN. Оно позволяет сэкономить немного времени на вводе текста, но может выполнить неожиданное соединение, если в таблицу добавляется или удаляется столбец с именем, которое в ней уже есть. Это предложение лучше использовать для быстрых запросов, а не для рабочего кода.

CROSS JOIN и Self Join

Другим способом соединения таблиц является отображение всех комбинаций строк двух таблиц. Это можно сделать с помощью `CROSS JOIN`. Если таблица соединяется с самой собой, то такая операция называется *самосоединением* (Self Join). Оно полезно, когда требуется сравнить строки в одной таблице.

Перекрестное соединение CROSS JOIN

Используйте предложение `CROSS JOIN`, чтобы вернуть все комбинации строк в двух таблицах. Это действие эквивалентно перечислению таблиц в предложении `FROM` (что иногда называют старым синтаксисом соединения).

```
-- С предложением CROSS JOIN
SELECT *
FROM states CROSS JOIN pets;

-- Эквивалентный список таблиц
SELECT *
FROM states, pets;
+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Ada  | AZ    | Deb  | dog  |
| Deb  | DE    | Deb  | dog  |
| Ada  | AZ    | Deb  | duck |
| Deb  | DE    | Deb  | duck |
| Ada  | AZ    | Pat  | pig  |
| Deb  | DE    | Pat  | pig  |
+------+-------+------+------+
```

После того как все комбинации будут перечислены, можно отфильтровать результаты, добавив предложение `WHERE`, чтобы возвращать меньшее количество строк в зависимости от того, что вы ищете.

Самосоединение

Вы можете соединить таблицу с самой собой с помощью самосоединения (Self Join). Как правило, это действие состоит из двух этапов:

1) отображения всех комбинаций строк при соединении таблицы с самой собой;
2) фильтрации полученных строк по некоторым критериям.

Ниже приведены два практических примера самосоединения.

Рассмотрим пример таблицы, в которой содержатся данные о сотрудниках и их руководителях:

```
SELECT * FROM employee;

+------+--------+----------+--------+
| dept | emp_id | emp_name | mgr_id |
+------+--------+----------+--------+
| tech |    201 | lisa     |    101 |
| tech |    202 | monica   |    101 |
| data |    203 | nancy    |    201 |
| data |    204 | olivia   |    201 |
| data |    205 | penny    |    202 |
+------+--------+----------+--------+
```

- *Пример 1. Возврат списка сотрудников и их руководителей.*

```
SELECT e1.emp_name, e2.emp_name as mgr_name
FROM employee e1, employee e2
WHERE e1.mgr_id = e2.emp_id;

+----------+----------+
| emp_name | mgr_name |
+----------+----------+
| nancy    | lisa     |
| olivia   | lisa     |
| penny    | monica   |
+----------+----------+
```

- *Пример 2. Сопоставление каждого сотрудника с другим сотрудником его отдела.*

```
SELECT e.dept, e.emp_name, matching_emp.emp_name
FROM employee e, employee matching_emp
WHERE e.dept = matching_emp.dept
      AND e.emp_name <> matching_emp.emp_name;
```

```
+------+----------+----------+
| dept | emp_name | emp_name |
+------+----------+----------+
| tech | monica   | lisa     |
| tech | lisa     | monica   |
| data | penny    | nancy    |
| data | olivia   | nancy    |
| data | penny    | olivia   |
| data | nancy    | olivia   |
| data | olivia   | penny    |
| data | nancy    | penny    |
+------+----------+----------+
```

В предыдущем запросе имеются повторяющиеся строки (monica/lisa и lisa/monica). Чтобы удалить дубликаты и вернуть только четыре строки вместо восьми, можно добавить строку:

```
AND e.emp_name < matching_emp.emp_name
```

в предложение WHERE, чтобы возвращать только строки, в которых первое имя предшествует второму в алфавитном порядке. Вот результат без дубликатов:

```
+------+----------+----------+
| dept | emp_name | emp_name |
+------+----------+----------+
| tech | lisa     | monica   |
| data | nancy    | olivia   |
| data | nancy    | penny    |
| data | olivia   | penny    |
+------+----------+----------+
```

Операции объединения

Ключевое слово `UNION` используется для объединения результатов двух или более операторов `SELECT`. Разница между предложениями `JOIN` и `UNION` заключается в том, что `JOIN` связывает несколько таблиц в рамках одного запроса, в то время как `UNION` объединяет результаты нескольких запросов:

```
-- пример с JOIN
SELECT *
FROM birthdays b JOIN candles c
     ON b.name = c.name;
```

```
-- пример с UNION
SELECT * FROM writers
UNION
SELECT * FROM artists;
```

На рис. 9.1 показана разница между результатами `JOIN` и `UNION` на основе приведенного выше кода.

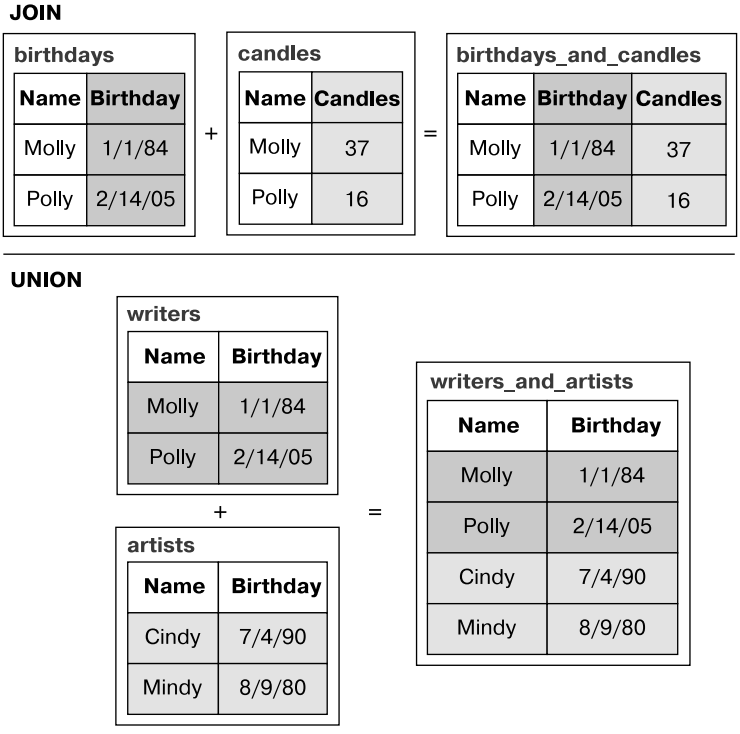


Рис. 9.1. JOIN в сравнении с UNION

Есть три способа объединить строки двух таблиц. Они также известны как *операции объединения*.

- `UNION` — объединяет результаты нескольких операторов.
- `EXCEPT` (`MINUS` в Oracle) — возвращает результаты одного оператора за вычетом набора результатов другого.

- `INTERSECT` — возвращает перекрывающиеся результаты нескольких операторов.

UNION

Ключевое слово `UNION` объединяет результаты двух или более операторов `SELECT` в один вывод.

Вот две таблицы, которые мы хотели бы объединить:

```
-- staff
+---------+---------+
| name    | origin  |
+---------+---------+
| michael | NULL    |
| janet   | NULL    |
| tahani  | england |
+---------+---------+

-- residents
+---------+---------+------------+
| name    | country | occupation |
+---------+---------+------------+
| eleanor | usa     | temp       |
| chidi   | nigeria | professor  |
| tahani  | england | model      |
| jason   | usa     | dj         |
+---------+---------+------------+
```

Используем операцию `UNION` для объединения двух таблиц и удаления повторяющихся строк:

```
SELECT name, origin FROM staff
UNION
SELECT name, country FROM residents;

+---------+---------+
| name    | origin  |
+---------+---------+
| michael | NULL    |
| janet   | NULL    |
| tahani  | england |
| eleanor | usa     |
| chidi   | nigeria |
| jason   | usa     |
+---------+---------+
```

Обратите внимание, что tahani/england присутствует как в таблице staff, так и в таблице residents. Однако в наборе результатов отображается только как одна строка, поскольку операция UNION удаляет повторяющиеся строки из выходных данных.

КАКИЕ ЗАПРОСЫ МОЖНО ОБЪЕДИНИТЬ

При выполнении операции UNION для двух запросов одни характеристики запросов должны совпадать, а для других это не обязательно.

- *Количество столбцов:* ***должно совпадать***. При объединении двух запросов необходимо указать одинаковое количество столбцов в обоих запросах.
- *Имена столбцов:* ***не обязательно должны совпадать***. Чтобы UNION можно было выполнить, имена столбцов в двух запросах не обязательно должны совпадать. Имена столбцов, используемые в первом операторе SELECT в запросе UNION, становятся именами выходных столбцов.
- *Типы данных:* ***должно совпадать***. Чтобы UNION можно было выполнить, типы данных двух запросов должны совпадать. Если это не так, то прежде, чем выполнять UNION, можно с помощью функции CAST привести их к одному и тому же типу данных.

UNION ALL

Используем операцию UNION ALL для объединения двух таблиц и сохранения повторяющихся строк:

```
SELECT name, origin FROM staff
UNION ALL
SELECT name, country FROM residents;
```

```
+---------+---------+
| name    | origin  |
+---------+---------+
| michael | NULL    |
| janet   | NULL    |
| tahani  | england |
| eleanor | usa     |
| chidi   | nigeria |
| tahani  | england |
| jason   | usa     |
+---------+---------+
```

Если вы точно уверены, что повторяющиеся строки невозможны, то используйте UNION ALL, чтобы повысить производительность. UNION выполняет дополнительную скрытую сортировку для выявления дубликатов.

UNION с другими предложениями

При использовании `UNION` можно задействовать и другие предложения, такие как `WHERE`, `JOIN` и т. д. Однако для всего запроса допускается только одно предложение `ORDER BY`, и оно должно находиться в самом конце.

Отфильтруем значения `NULL` и отсортируем результаты запроса `UNION`:

```
SELECT name, origin
FROM staff
WHERE origin IS NOT NULL

UNION

SELECT name, country
FROM residents

ORDER BY name;

+---------+---------+
| name    | origin  |
+---------+---------+
| chidi   | nigeria |
| eleanor | usa     |
| jason   | usa     |
| tahani  | england |
+---------+---------+
```

UNION с несколькими таблицами

Можно объединить несколько таблиц с помощью дополнительных предложений `UNION`.

Объединим строки нескольких таблиц:

```
SELECT name, origin
FROM staff
```

```
UNION

SELECT name, country
FROM residents

UNION

SELECT name, country
FROM pets;
```

С помощью UNION обычно объединяются результаты из нескольких таблиц. Если вы объединяете результаты из одной, то лучше вместо UNION написать один запрос и использовать соответствующее предложение WHERE, оператор CASE и т. д.

EXCEPT и INTERSECT

Помимо использования операции `UNION` для объединения строк нескольких таблиц, можно с помощью операций `EXCEPT` и `INTERSECT` объединять строки различными способами.

EXCEPT

Используем операцию `EXCEPT` для «вычитания» результатов одного запроса из результатов другого.

Вернем данные о сотрудниках, которые не являются резидентами:

```
SELECT name FROM staff
EXCEPT
SELECT name FROM residents;

+---------+
| name    |
+---------+
| michael |
| janet   |
+---------+
```

MySQL не поддерживает операцию EXCEPT. Вместо этого в качестве обходного пути можно использовать ключевые слова NOT IN:

```
SELECT name
FROM staff
WHERE name NOT IN (SELECT name FROM residents);
```

Oracle использует операцию MINUS вместо EXCEPT.

PostgreSQL также поддерживает операцию EXCEPT ALL, которая не удаляет дубликаты. Операция EXCEPT удаляет все вхождения значения, а EXCEPT ALL — отдельные экземпляры.

INTERSECT

Используем операцию INTERSECT для поиска общих строк в двух запросах.

Вернем данные о сотрудниках, которые являются резидентами:

```
SELECT name, origin FROM staff
INTERSECT
SELECT name, country FROM residents;
```

```
+---------+---------+
| name    | origin  |
+---------+---------+
| tahani  | england |
+---------+---------+
```

MySQL не поддерживает операцию INTERSECT. В качестве обходного пути можно использовать предложение INNER JOIN:

```
SELECT s.name, s.origin
FROM staff s INNER JOIN residents r
     ON s.name = r.name;
```

PostgreSQL также поддерживает операцию INTERSECT ALL, которая сохраняет повторяющиеся значения.

ОПЕРАЦИИ ОБЪЕДИНЕНИЯ: ПОРЯДОК ВЫПОЛНЕНИЯ

При написании оператора, который содержит несколько операций объединения (UNION, EXCEPT, INTERSECT), используйте круглые скобки, чтобы указать порядок выполнения действий.

```
SELECT * FROM staff
EXCEPT
(SELECT * FROM residents
UNION
SELECT * FROM pets);
```

Если не указано иное, то операции объединения выполняются в порядке сверху вниз, за исключением того, что INTERSECT имеет приоритет над UNION и EXCEPT.

Обобщенные табличные выражения

Обобщенное табличное выражение (common table expression, CTE) представляет собой временный набор результатов. Другими словами, оно временно сохраняет результаты запроса, чтобы вы могли написать другие запросы, ссылающиеся на него.

Определить CTE можно по наличию ключевого слова WITH. Есть два типа CTE:

- *нерекурсивное CTE* — запрос, на который ссылаются другие запросы (см. подраздел «CTE в сравнении с подзапросами» ниже);
- *рекурсивное CTE* — запрос, который ссылается сам на себя (см. подраздел «Рекурсивные CTE» ниже).

Нерекурсивные CTE встречаются гораздо чаще, чем рекурсивные. Чаще всего если кто-то упоминает CTE, то имеет в виду нерекурсивные.

Рассмотрим пример нерекурсивного CTE:

```
-- Запрос результатов my_cte
WITH my_cte AS (
   SELECT name, AVG(grade) AS avg_grade
   FROM my_table
   GROUP BY name)

SELECT *
FROM my_cte
WHERE avg_grade < 70;
```

Рассмотрим пример рекурсивного CTE:

```
-- Генерация чисел от 1 до 10
WITH RECURSIVE my_cte(n) AS
(
  SELECT 1 -- Include FROM dual in Oracle
  UNION ALL
  SELECT n + 1 FROM my_cte WHERE n < 10
)

SELECT * FROM my_cte;
```

В MySQL и PostgreSQL ключевое слово `RECURSIVE` является обязательным. В Oracle и SQL Server это слово необходимо опустить. SQLite работает с любым из этих синтаксисов.

В Oracle можно встретить старый код, использующий синтаксис `CONNECT BY` для рекурсивных запросов, но в настоящее время гораздо чаще применяются CTE.

CTE в сравнении с подзапросами

И CTE, и подзапросы позволяют написать запрос, а затем написать другой, который ссылается на первый. В этом подразделе описывается разница между этими двумя подходами.

Представьте, что ваша цель — найти данные об отделе с самой большой средней зарплатой. Это можно сделать в два

этапа: написать запрос, который возвращает данные о средней зарплате для каждого отдела; с помощью CTE или подзапроса написать второй запрос на основе первого, чтобы вернуть данные об отделе с наибольшей средней зарплатой.

- *Этап 1. Запрос, который находит данные о средней зарплате по каждому отделу.*

```
SELECT dept, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept;

+-------+------------+
| dept  | avg_salary |
+-------+------------+
| mktg  |      78000 |
| sales |      61000 |
| tech  |      83000 |
+-------+------------+
```

- *Этап 2. CTE и подзапрос, который, используя предыдущий запрос, находит данные об отделе с наибольшей средней зарплатой.*

```
-- подход с CTE
WITH avg_dept_salary AS (
   SELECT dept, AVG(salary) AS avg_salary
   FROM employees
   GROUP BY dept)

SELECT *
FROM avg_dept_salary
ORDER BY avg_salary DESC
LIMIT 1;

-- Эквивалентный подход с подзапросом
SELECT *
FROM
(SELECT dept, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept) avg_dept_salary

ORDER BY avg_salary DESC
LIMIT 1;
```

```
+------+------------+
| dept | avg_salary |
+------+------------+
| tech |      83000 |
+------+------------+
```

Синтаксис предложения `LIMIT` различается в зависимости от программного обеспечения. Замените `LIMIT 1` на `ROWNUM = 1` в Oracle и `TOP 1` в SQL Server. Более подробную информацию можно найти в разделе «Предложение LIMIT» главы 4.

ПРЕИМУЩЕСТВА CTE ПО СРАВНЕНИЮ С ПОДЗАПРОСОМ

Использование CTE вместо подзапроса имеет несколько преимуществ.

- *Множественные ссылки.* После того как CTE будет определено, можно многократно ссылаться на него по имени в последующих запросах `SELECT`:

  ```
  WITH my_cte AS (...)

  SELECT * FROM my_cte WHERE id > 10
  UNION
  SELECT * FROM my_cte WHERE score > 90;
  ```

  При использовании подзапроса необходимо каждый раз писать полный подзапрос.

- *Несколько таблиц.* Синтаксис CTE более удобен для чтения при работе с несколькими таблицами, поскольку можно перечислить все CTE заранее:

  ```
  WITH my_cte1 AS (...),
       my_cte2 AS (...)

  SELECT *
  FROM my_cte1 m1
       INNER JOIN my_cte2 m2
       ON m1.id = m2.id;
  ```

  При использовании подзапросов они будут разбросаны по всему запросу.

CTE *не поддерживаются* в более старых версиях SQL, поэтому подзапросы широко используются до сих пор.

Рекурсивные CTE

В этом подразделе рассматриваются две практические ситуации, в которых рекурсивное CTE может быть полезным.

Заполнение недостающих строк в последовательности данных

В этой таблице приведены даты и цены. Обратите внимание, что в столбце `date` отсутствуют данные за второе и пятое число месяца.

```
SELECT * FROM stock_prices;

+------------+--------+
| date       | price  |
+------------+--------+
| 2021-03-01 | 668.27 |
| 2021-03-03 | 678.83 |
| 2021-03-04 | 635.40 |
| 2021-03-06 | 591.01 |
+------------+--------+
```

Столбец `date` заполняется в два этапа:

1) используется рекурсивное CTE для генерации последовательности дат;
2) соединяется последовательность дат с исходной таблицей.

Приведенный ниже код работает в MySQL. В табл. 9.4 приведен синтаксис для каждой РСУБД.

- *Этап 1. С помощью рекурсивного CTE создаем последовательность дат под названием* `my_dates`. Таблица `my_dates` начинается с даты `2021-03-01`, и следующая дата добавляется снова и снова, вплоть до даты `2021-03-06`:

```
-- Синтаксис MySQL
WITH RECURSIVE my_dates(dt) AS (
   SELECT '2021-03-01'
   UNION ALL
   SELECT dt + INTERVAL 1 DAY
   FROM my_dates
   WHERE dt < '2021-03-06')

SELECT * FROM my_dates;

+------------+
| dt         |
+------------+
| 2021-03-01 |
| 2021-03-02 |
| 2021-03-03 |
| 2021-03-04 |
| 2021-03-05 |
| 2021-03-06 |
+------------+
```

- *Этап 2. Выполняем левое соединение рекурсивного CTE с исходной таблицей.*

```
-- Синтаксис MySQL
WITH RECURSIVE my_dates(dt) AS (
   SELECT '2021-03-01'
   UNION ALL
   SELECT dt + INTERVAL 1 DAY
   FROM my_dates
   WHERE dt < '2021-03-06')
SELECT d.dt, s.price
FROM my_dates d
     LEFT JOIN stock_prices s
     ON d.dt = s.date;

+------------+--------+
| dt         | price  |
+------------+--------+
| 2021-03-01 | 668.27 |
| 2021-03-02 |   NULL |
| 2021-03-03 | 678.83 |
| 2021-03-04 | 635.40 |
| 2021-03-05 |   NULL |
| 2021-03-06 | 591.01 |
+------------+--------+
```

- *Этап 3 (необязательный). Заполняем значения* NULL *вчерашними ценами.* Заменяем предложение SELECT (SELECT d.dt, s.price) на:

```
SELECT d.dt, COALESCE(s.price,
       LAG(s.price) OVER
       (ORDER BY d.dt)) AS price
 ...

+------------+--------+
| dt         | price  |
+------------+--------+
| 2021-03-01 | 668.27 |
| 2021-03-02 | 668.27 |
| 2021-03-03 | 678.83 |
| 2021-03-04 | 635.40 |
| 2021-03-05 | 635.40 |
| 2021-03-06 | 591.01 |
+------------+--------+
```

В каждой РСУБД синтаксис различается.

Ниже приведен общий синтаксис генерации столбца даты. Выделенные жирным шрифтом фрагменты различаются в зависимости от РСУБД, а код, характерный для конкретного программного обеспечения, приведен в табл. 9.4.

```
[WITH] my_dates(dt) AS (
   SELECT [DATE]
   UNION ALL
   SELECT [DATE PLUS ONE]
   FROM my_dates
   WHERE dt < [LAST DATE])

SELECT * FROM my_dates;
```

Таблица 9.4. Формирование столбца даты в каждой РСУБД

РСУБД	WITH	DATE	DATE PLUS ONE	LAST DATE
MySQL	WITH RECURSIVE	'2021-03-01'	dt + INTERVAL 1 DAY	'2021-03-06'
Oracle	WITH	DATE '2021-03-01'	dt + INTERVAL '1' DAY	DATE '2021-03-06'

РСУБД	WITH	DATE	DATE PLUS ONE	LAST DATE
PostgreSQL	`WITH RECURSIVE`	`CAST('2021-03-01' AS DATE)`	`CAST(dt + INTERVAL '1 day' AS DATE)`	`'2021-03-06'`
SQL Server	`WITH`	`CAST('2021-03-01' AS DATE)`	`DATEADD(DAY, 1, CAST(dt AS DATE))`	`'2021-03-06'`
SQLite	`WITH RECURSIVE`	`-`	`DATE(dt, '1 day')`	`'2021-03-06'`

Возврат всех родителей дочерней строки

В таблице ниже представлены роли различных членов семьи. В крайнем правом столбце указан идентификатор родителя человека.

```
SELECT * FROM family_tree;
```

```
+------+---------+----------+-----------+
| id   | name    | role     | parent_id |
+------+---------+----------+-----------+
|    1 | Lao Ye  | Grandpa  |      NULL |
|    2 | Lao Lao | Grandma  |      NULL |
|    3 | Ollie   | Dad      |      NULL |
|    4 | Alice   | Mom      |         1 |
|    4 | Alice   | Mom      |         2 |
|    5 | Henry   | Son      |         3 |
|    5 | Henry   | Son      |         4 |
|    6 | Lily    | Daughter |         3 |
|    6 | Lily    | Daughter |         4 |
+------+---------+----------+-----------+
```

Приведенный ниже код выполняется в MySQL. В табл. 9.5 (см. ниже) приведен синтаксис для каждой РСУБД.

С помощью рекурсивного CTE можно перечислить родителей, бабушек и дедушек каждого человека:

```
-- Синтаксис MySQL
WITH RECURSIVE my_cte (id, name, lineage) AS (
```

```
   SELECT id, name, name AS lineage
   FROM family_tree
   WHERE parent_id IS NULL
   UNION ALL
   SELECT ft.id, ft.name,
          CONCAT(mc.lineage, ' > ', ft.name)
   FROM family_tree ft
        INNER JOIN my_cte mc
        ON ft.parent_id = mc.id)

SELECT * FROM my_cte ORDER BY id;

+------+---------+-------------------------+
| id   | name    | lineage                 |
+------+---------+-------------------------+
|    1 | Lao Ye  | Lao Ye                  |
|    2 | Lao Lao | Lao Lao                 |
|    3 | Ollie   | Ollie                   |
|    4 | Alice   | Lao Ye > Alice          |
|    4 | Alice   | Lao Lao > Alice         |
|    5 | Henry   | Ollie > Henry           |
|    5 | Henry   | Lao Ye > Alice > Henry  |
|    5 | Henry   | Lao Lao > Alice > Henry |
|    6 | Lily    | Ollie > Lily            |
|    6 | Lily    | Lao Ye > Alice > Lily   |
|    6 | Lily    | Lao Lao > Alice > Lily  |
+------+---------+-------------------------+
```

В приведенном выше коде (известном также как иерархический запрос) `my_cte` содержит два объединенных оператора:

- первый оператор `SELECT` является начальной точкой. Строки, в которых `parent_id` имеет значение `NULL`, рассматриваются как корни дерева;
- второй оператор `SELECT` определяет рекурсивную связь между родительской и дочерней строками. Возвращаются дочерние строки каждого корня дерева и добавляются в столбец `lineage` до тех пор, пока не будет составлена полная родословная.

В каждой РСУБД синтаксис различается.

Ниже приведен общий синтаксис для перечисления всех родителей. Выделенные жирным шрифтом фрагменты раз-

личаются в зависимости от РСУБД, а код, характерный для конкретного программного обеспечения, приведен в табл. 9.5.

```
[WITH] my_cte (id, name, lineage) AS (
   SELECT id, name, [NAME] AS lineage
   FROM family_tree
   WHERE parent_id IS NULL
   UNION ALL
   SELECT ft.id, ft.name, [LINEAGE]
   FROM family_tree ft
        INNER JOIN my_cte mc
        ON ft.parent_id = mc.id)

SELECT * FROM my_cte ORDER BY id;
```

Таблица 9.5. Перечисление всех родителей в каждой РСУБД

РСУБД	WITH	NAME	LINEAGE
MySQL	`WITH RECURSIVE`	`name`	`CONCAT(mc.lineage, ' > ', ft.name)`
Oracle	`WITH`	`name`	`mc.lineage || ' > ' || ft.name`
PostgreSQL	`WITH RECURSIVE`	`CAST(name AS VARCHAR(30))`	`CAST(CONCAT(mc.lineage, ' > ', ft.name) AS VARCHAR(30))`
SQL Server	`WITH`	`CAST(name AS VARCHAR(30))`	`CAST(CONCAT(mc.lineage, ' > ', ft.name) AS VARCHAR(30))`
SQLite	`WITH RECURSIVE`	`name`	`mc.lineage || ' > ' || ft.name`

ГЛАВА 10

Как мне?..

Эта глава предназначена для краткого ознакомления с часто задаваемыми вопросами по SQL, объединяющими несколько концепций:

- поиск строк, содержащих повторяющиеся значения;
- выбор строк с максимальным значением для другого столбца;
- конкатенацию текста из нескольких полей в одно;
- поиск всех таблиц, содержащих определенное имя столбца;
- обновление таблицы, идентификатор в которой совпадает с идентификатором в другой таблице.

Поиск строк, содержащих повторяющиеся значения

В таблице ниже приведены данные о семи видах чая и температуре, при которой их следует заваривать. Обратите внимание: есть два набора дублирующих друг друга значений `tea`/`temperature`, которые выделены жирным шрифтом.

```
SELECT * FROM teas;

+----+--------+-------------+
| id | tea    | temperature |
+----+--------+-------------+
|  1 | green  |         170 |
|  2 | black  |         200 |
```

```
|  3 | black  |         200 |
|  4 | herbal |         212 |
|  5 | herbal |         212 |
|  6 | herbal |         210 |
|  7 | oolong |         185 |
+----+--------+-------------+
```

В данном разделе рассматриваются два различных сценария для поиска повторяющихся значений:

- возврат всех уникальных комбинаций значений `tea`/`temperature`;
- возврат только строки с повторяющимися значениями `tea`/`temperature`.

Возврат всех уникальных комбинаций

Чтобы исключить дублирующиеся значения и вернуть только уникальные строки таблицы, используем ключевое слово `DISTINCT`:

```
SELECT DISTINCT tea, temperature
FROM teas;

+--------+-------------+
| tea    | temperature |
+--------+-------------+
| green  |         170 |
| black  |         200 |
| herbal |         212 |
| herbal |         210 |
| oolong |         185 |
+--------+-------------+
```

Возможные добавления

Чтобы вернуть количество уникальных строк в таблице, можно использовать ключевые слова `COUNT` и `DISTINCT` одновременно. Более подробную информацию см. в подразделе «Ключевое слово DISTINCT» в главе 4.

Возврат только строк с повторяющимися значениями

В этом запросе определяются строки таблицы с повторяющимися значениями:

```
WITH dup_rows AS (
     SELECT tea, temperature,
            COUNT(*) as num_rows
     FROM teas
     GROUP BY tea, temperature
     HAVING COUNT(*) > 1)

SELECT t.id, d.tea, d.temperature
FROM teas t INNER JOIN dup_rows d
     ON t.tea = d.tea
     AND t.temperature = d.temperature;
```

```
+----+--------+-------------+
| id | tea    | temperature |
+----+--------+-------------+
|  2 | black  |         200 |
|  3 | black  |         200 |
|  4 | herbal |         212 |
|  5 | herbal |         212 |
+----+--------+-------------+
```

Объяснение

Основная часть работы выполняется в запросе `dup_rows`. Подсчитываются все комбинации `tea`/`temperature`, а затем с помощью предложения `HAVING` сохраняются только те комбинации, которые встречаются несколько раз. Результат запроса `dup_rows` выглядит так:

```
+--------+-------------+----------+
| tea    | temperature | num_rows |
+--------+-------------+----------+
| black  |         200 |        2 |
| herbal |         212 |        2 |
+--------+-------------+----------+
```

Цель использования `JOIN` во второй половине запроса — вернуть столбец `id` в конечном результате.

Ключевые слова в запросе

- **`WITH dup_rows`** — это начало обобщенного табличного выражения, которое позволяет работать с несколькими операторами `SELECT` в рамках одного запроса.
- **`HAVING COUNT(*) > 1`** использует предложение `HAVING`, которое позволяет фильтровать по агрегату типа `COUNT()`.
- **`teas t INNER JOIN dup_rows d`** использует предложение `INNER JOIN`, которое позволяет объединить таблицу `teas` и запрос `dup_rows`.

Возможные добавления

Для удаления конкретных повторяющихся строк из таблицы используется оператор `DELETE`. Более подробную информацию о нем можно найти в главе 5.

Выбор строк с максимальным значением для другого столбца

В таблице ниже перечислены данные о сотрудниках и количестве совершенных ими продаж. Вам необходимо получить данные о последних продажах для каждого сотрудника, которые выделены жирным шрифтом.

```
SELECT * FROM sales;
```

```
+------+----------+------------+-------+
| id   | employee | date       | sales |
+------+----------+------------+-------+
|    1 | Emma     | 2021-08-01 |     6 |
|    2 | Emma     | 2021-08-02 |    17 |
|    3 | Jack     | 2021-08-02 |    14 |
|    4 | Emma     | 2021-08-04 |    20 |
|    5 | Jack     | 2021-08-05 |     5 |
|    6 | Emma     | 2021-08-07 |     1 |
+------+----------+------------+-------+
```

Решение

Этот запрос возвращает количество продаж, совершенных каждым сотрудником на дату последней продажи (то есть наибольшее значение даты продажи для каждого сотрудника):

```
SELECT s.id, r.employee, r.recent_date, s.sales
FROM (SELECT employee, MAX(date) AS recent_date
      FROM sales
      GROUP BY employee) r
INNER JOIN sales s
           ON r.employee = s.employee
           AND r.recent_date = s.date;

+------+----------+-------------+-------+
| id   | employee | recent_date | sales |
+------+----------+-------------+-------+
|    5 | Jack     | 2021-08-05  |     5 |
|    6 | Emma     | 2021-08-07  |     1 |
+------+----------+-------------+-------+
```

Объяснение

Ключ к решению этой задачи состоит в том, чтобы разбить ее на две части. Первая часть — определить последнюю дату продажи для каждого сотрудника. Вывод подзапроса `r` выглядит так:

```
+----------+-------------+
| employee | recent_date |
+----------+-------------+
| Emma     | 2021-08-07  |
| Jack     | 2021-08-05  |
+----------+-------------+
```

Вторая часть — вернуть столбцы `id` и `sales` в конечный результат, для чего используется `JOIN` во второй половине запроса.

Ключевые слова в запросе

- **`GROUP BY employee`** использует предложение `GROUP BY`, которое разбивает таблицу `employee` и находит значение **`MAX(date)`** для каждого сотрудника.
- **`r INNER JOIN sales s`** использует предложение `INNER JOIN`, что позволяет объединить подзапрос `r` и таблицу `sales`.

Возможные добавления

Альтернативой решению, в котором используется `GROUP BY`, выступает вариант, где оконная функция (`OVER ... PARTITION BY ...`) применяется с функцией `FIRST_VALUE`, возвращающей те же результаты. Более подробную информацию можно найти в разделе «Оконные функции» главы 8.

Конкатенация текста из нескольких полей в одно

В данном разделе рассматриваются два различных сценария:

- конкатенация текста из полей *в одной строке* в одно значение;
- конкатенация текста из полей *в нескольких строках* в одно значение.

Конкатенация текста из полей в одной строке

Эта таблица содержит два столбца, которые необходимо объединить:

```
+----+---------+        +-----------+
| id | name    |        | id_name   |
+----+---------+        +-----------+
|  1 | Boots   |  --->  | 1_Boots   |
|  2 | Pumpkin |        | 2_Pumpkin |
|  3 | Tiger   |        | 3_Tiger   |
+----+---------+        +-----------+
```

Для объединения значений используем функцию `CONCAT` или операцию конкатенации (`||`):

```
-- MySQL, PostgreSQL и SQL Server
SELECT CONCAT(id, '_', name) AS id_name
FROM my_table;

-- Oracle, PostgreSQL, и SQLite
SELECT id || '_' || name AS id_name
FROM my_table;
```

```
+-----------+
| id_name   |
+-----------+
| 1_Boots   |
| 2_Pumpkin |
| 3_Tiger   |
+-----------+
```

Возможные добавления

В главе 7, помимо CONCAT, рассматриваются и другие способы работы со строковыми значениями, в том числе:

- определение длины строки;
- поиск слов в строке;
- извлечение текста из строки.

Конкатенация текста из полей в нескольких строках

В таблице ниже перечислены данные о калориях, сожженных каждым человеком. Вы хотите объединить данные о калориях каждого человека в одну строку.

```
+------+----------+      +------+----------+
| name | calories |      | name | calories |
+------+----------+      +------+----------+
| ally |       80 | ---> | ally | 80,75,90 |
| ally |       75 |      | jess | 100,92   |
| ally |       90 |      +------+----------+
| jess |      100 |
| jess |       92 |
+------+----------+
```

Для создания списка используйте функцию типа GROUP_CONCAT, LISTAGG, ARRAY_AGG или STRING_AGG.

```
SELECT name,
       GROUP_CONCAT(calories) AS calories_list
FROM workouts
GROUP BY name;
```

```
+------+--------------+
| name | calories_list |
+------+--------------+
| ally | 80,75,90      |
| jess | 100,92        |
+------+--------------+
```

Этот код работает в MySQL и SQLite. В других РСУБД замените `GROUP_CONCAT(calories)` на следующие функции.

- Oracle:

  `LISTAGG(calories, ',')`.

- PostgreSQL:

  `ARRAY_AGG(calories)`.

- SQL Server:

  `STRING_AGG(calories, ',')`.

Возможные добавления

В подразделе «Агрегирование строк в одно значение или список» главы 8 содержится подробная информация о том, как использовать другие разделители, помимо запятой (`,`), как сортировать значения и возвращать уникальные значения.

Поиск всех таблиц, содержащих определенное имя столбца

Представьте, что у вас есть база данных с большим количеством таблиц. Вы хотите быстро найти все таблицы, которые содержат имя столбца со словом `city`.

Решение

В большинстве РСУБД есть специальная таблица, содержащая все имена таблиц и названия столбцов. В табл. 10.1 показано, как сделать запрос к ней в каждой системе.

Последняя строка каждого блока кода является необязательной. Вы можете добавить ее, если хотите сузить результаты для конкретной базы данных или пользователя. Если этот параметр исключен, то будут возвращены все таблицы.

Таблица 10.1. Поиск всех таблиц, содержащих определенное имя столбца

РСУБД	Код

MySQL	`SELECT table_name, column_name`
	`FROM information_schema.columns`
	`WHERE column_name LIKE '%city%'`
	`  AND table_schema = 'my_db_name';`

Oracle	`SELECT table_name, column_name`
	`FROM all_tab_columns`
	`WHERE column_name LIKE '%CITY%'`
	`  AND owner = 'MY_USER_NAME';`

PostgreSQL, SQL Server	`SELECT table_name, column_name`
	`FROM information_schema.columns`
	`WHERE column_name LIKE '%city%'`
	`  AND table_catalog = 'my_db_name';`

В результате будут выведены имена всех столбцов, содержащие слово `city`, а также таблицы, в которых они находятся:

```
+------------+-------------+
| TABLE_NAME | COLUMN_NAME |
+------------+-------------+
| customers  | city        |
| employees  | city        |
| locations  | metro_city  |
+------------+-------------+
```

В SQLite нет таблицы, содержащей все имена всех столбцов. Вместо этого можно вручную показать все таблицы, а затем просмотреть имена столбцов в каждой из них:

```
.tables
pragma table_info(my_table);
```

Возможные добавления

В главе 5 рассматриваются дополнительные способы взаимодействия с базами данных и таблицами, в том числе:

- просмотр названий существующих баз данных;
- просмотр названий существующих таблиц;
- просмотр названий столбцов таблицы.

В главе 7, помимо `LIKE`, рассматриваются и другие способы поиска текста, в том числе:

- операция `=` для поиска точного совпадения;
- операция `IN` для поиска по нескольким терминам;
- регулярные выражения для поиска шаблона.

Обновление таблицы, в которой идентификатор совпадает с идентификатором в другой таблице

Представьте, что у вас есть две таблицы: `products` и `deals`. Вы хотите обновить значения в столбце `name` таблицы `deals` значениями в столбце `name` таблицы `products` для строк, которые имеют совпадающие `id`.

```
SELECT * FROM products;

+------+--------------------+
| id   | name               |
+------+--------------------+
|  101 | Mac and cheese mix |
|  102 | MIDI keyboard      |
|  103 | Mother's day card  |
+------+--------------------+

SELECT * FROM deals;
```

```
+------+--------------+
| id   | name         |
+------+--------------+
|  102 | Tech gift    | --> MIDI keyboard
|  103 | Holiday card | --> Mother's day card
+------+--------------+
```

Решение

Можно использовать оператор `UPDATE` для изменения значений в таблице с помощью синтаксиса `UPDATE ... SET ...`. В табл. 10.2 показано, как это сделать в каждой РСУБД.

Таблица 10.2. Обновление таблицы, идентификатор которой совпадает с идентификатором в другой таблице

РСУБД	Код

MySQL	`UPDATE deals d,`
	`products p`
	`SET d.name = p.name`
	`WHERE d.id = p.id;`

Oracle	`UPDATE deals d`
	`SET name = (SELECT p.name`
	`FROM products p`
	`WHERE d.id = p.id);`

PostgreSQL, SQLite	`UPDATE deals`
	`SET name = p.name`
	`FROM deals d`
	`INNER JOIN products p`
	`ON d.id = p.id`
	`WHERE deals.id = p.id;`

SQL Server	`UPDATE d`
	`SET d.name = p.name`
	`FROM deals d`
	`INNER JOIN products p`
	`ON d.id = p.id;`

Теперь в таблицу `deals` добавляются названия из таблицы `products`:

```
SELECT * FROM deals;

+------+-------------------+
| id   | name              |
+------+-------------------+
|  102 | MIDI keyboard     |
|  103 | Mother's day card |
+------+-------------------+
```

После того как оператор UPDATE будет выполнен, его результаты отменить невозможно. Исключением является случай, когда вы начинаете транзакцию до выполнения оператора UPDATE.

Возможные добавления

В главе 5 рассматривались дополнительные способы изменения таблиц, в том числе:

- обновление столбца данных;
- обновление строк данных;
- обновление строк данных результатами запроса;
- добавление столбца в таблицу.

ЗАКЛЮЧИТЕЛЬНЫЕ СЛОВА

В этой книге мы рассмотрели наиболее популярные концепции и ключевые слова SQL, но коснулись лишь малой части возможностей этого языка. SQL можно использовать для выполнения множества задач, применяя самые разные подходы. Я призываю вас продолжать учиться и исследовать.

Вы, наверное, заметили, что синтаксис SQL сильно различается в зависимости от РСУБД. Написание кода SQL требует большой практики, терпения и изучения синтаксиса. Я надеюсь, что этот карманный справочник окажется полезным для вас.

Об авторе

Элис Жао — специалист по анализу данных, которая страстно любит преподавать и делать сложные вещи простыми для понимания. Она преподавала многочисленные курсы по SQL, Python и R в качестве старшего специалиста по данным в компании Metis и была соучредителем Best Fit Analytics. Ее технические учебники на YouTube, получившие высокие оценки, известны своей практичностью, занимательностью и визуальной привлекательностью.

Элис пишет об аналитике и поп-культуре в своем блоге *A Dash of Data*. Ее работы были опубликованы в *Huffington Post*, *Thrillist* и *Working Mother*. Она выступала на различных конференциях, включая Strata в Нью-Йорке и ODSC в Сан-Франциско, по самым разным темам: от обработки естественного языка до визуализации данных. Получила в Северо-Западном университете степень магистра в области аналитики и степень бакалавра в области электротехники.

Иллюстрация на обложке

На обложке изображена альпийская саламандра *(Salamandra atra)*. Она чаще всего встречается в ущельях высоко в Альпах (до 1000 м) и отличается необычной способностью хорошо переносить холодную погоду. Эти блестящие черные существа предпочитают тенистые, влажные места, а также трещины и щели в каменных стенах. Они питаются червями, пауками, улитками и личинками мелких насекомых.

В отличие от других саламандр альпийская саламандра рождает полностью сформировавшуюся молодь. Беременность длится два года, но на больших высотах (1400–1700 м) может продолжаться до трех лет. Эти земноводные находятся под защитой — их относят к видам, рискующим исчезнуть при определенных условиях. Биологи подчеркивают, что важно сохранять для них привычные условия обитания: саламандрам нужны скалистые и не слишком сухие ландшафты с умеренной растительностью.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой исчезновения; все они важны для нашего мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры из «Королевской естественной истории» Лайдеккера.

Элис Жао

SQL. Pocket guide

4-е издание

Перевел с английского В. Дмитрущенков

Изготовлено в России. Изготовитель: ТОО «Спринт Бук».
Место нахождения и фактический адрес:
010000, Казахстан, город Астана, район Алматы,
Проспект Ракымжан Кошкарбаев, дом 10/1, н. п. 18.

Дата изготовления: 02.2024. Наименование: книжная продукция.
Срок годности: не ограничен.

Подписано в печать 26.01.24. Формат 84×108/32. Бумага офсетная. Усл. п. л. 16,800.
Тираж 1500. Заказ 0000.