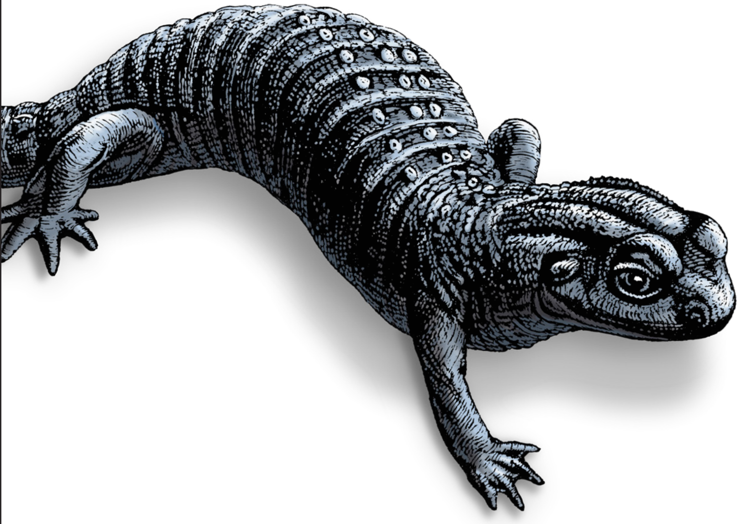
O'REILLY®

4th Edition

SQL Pocket Guide

A Guide to SQL Usage



Alice Zhao

O'REILLY®

SQL Pocket Guide

If you use SQL in your day-to-day work as a data analyst, data scientist, or data engineer, this popular pocket guide is your ideal on-the-job reference. You'll find many examples that address the language's complexities, along with key aspects of SQL used in Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL, and SQLite.

In this updated edition, author Alice Zhao describes how these database management systems implement SQL syntax for both querying and making changes to a database. You'll find details on data types and conversions, regular expression syntax, window functions, pivoting and unpivoting, and more.

- **Quickly look up how to perform specific tasks using SQL**
- **Apply the book's syntax examples to your own queries**
- **Update SQL queries to work in five different database management systems**
- **New: Connect Python and R to a relational database**
- **New: Look up frequently asked SQL questions in the "How Do I?" chapter**

DATA

Twitter: @oreillymedia
facebook.com/oreilly

US $29.99 CAN $39.99
ISBN: 978-1-492-09040-3

52999
9 781492 090403

FOURTH EDITION

SQL Pocket Guide

Alice Zhao

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

SQL Pocket Guide

by Alice Zhao

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Andy Kwan
Development Editor: Amelia Blevins and Jeff Bleiel
Production Editor: Caitlin Ghegan
Copyeditor: Piper Editorial Consulting, LLC
Proofreader: James Fraleigh
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2021: Fourth Edition

Revision History for the Fourth Edition

2021-08-26: First Release

See *https://oreil.ly/sqlpocketerrata* for release details.

978-1-492-09040-3

[LSI]

Table of Contents

Preface

Why SQL?

Since the last edition of *SQL Pocket Guide* was published, a lot has changed in the data world. The amount of data generated and collected has exploded, and a number of tools and jobs have been created to handle the influx of data. Through all of the changes, SQL has remained an integral part of the data landscape.

Over the past 15 years, I have worked as an engineer, consultant, analyst, and data scientist, and I have used SQL in every one of my roles. Even if my main responsibilities were focused on another tool or skill, I had to know SQL in order to access data at a company.

> *If there was a programming language award for best supporting actor, SQL would take home the prize.*

As new technologies emerge, SQL is still top of mind when it comes to working with data. Cloud-based storage solutions like Amazon Redshift and Google BigQuery require users to write SQL queries to pull data. Distributed data processing frameworks like Hadoop and Spark have sidekicks Hive and Spark SQL, respectively, which provide SQL-like interfaces for users to analyze data.

SQL has been around for almost five decades, and it is not going away anytime soon. It is one of the oldest programming languages still being used widely today, and I am excited to share the latest and greatest with you in this book.

Goals of This Book

There are many existing SQL books out there, ranging from ones that teach beginners how to code in SQL to detailed technical specifications for database administrators. This book is not intended to cover all SQL concepts in depth, but rather to be a simple reference for when:

- You've forgotten some SQL syntax and need to look it up quickly
- You've come across a slightly different set of database tools at a new job and need to look up the nuanced differences
- You've been focusing on another coding language for a while and need a quick refresher on how SQL works

If SQL plays a large supporting role in your job, then this is the perfect pocket guide for you.

Updates to the Fourth Edition

The third edition of the *SQL Pocket Guide* by Jonathan Gennick was published in 2010, and it was well received by readers. I've made the following updates to the fourth edition:

- The syntax has been updated for Microsoft SQL Server, MySQL, Oracle Database, and PostgreSQL. IBM's Db2 has been removed due to its decrease in popularity, and SQLite has been added due to its increase in popularity.
- The third edition of this book was organized alphabetically. I've rearranged the sections in the fourth edition so that similar concepts are grouped together. There is still an

index at the end of this book that lists concepts alphabetically.

- Due to the number of data analysts and data scientists who are now using SQL in their jobs, I've added sections on how to use SQL with Python and R (popular open source programming languages), as well as a SQL crash course for those who need a quick refresher.

Frequently Asked (SQL) Questions

The last chapter of this book is called "How Do I…?" and it includes frequently asked questions by SQL beginners or those who haven't used SQL in a while.

It's a good place to start if you don't remember the exact keyword or concept that you're looking for. Example questions include:

- How do I find the rows containing duplicate values?
- How do I select rows with the max value for another column?
- How do I concatenate text from multiple fields into a single field?

Navigating This Book

This book is organized into three sections.

I. Basic Concepts

- Chapters 1 through 3 introduce basic keywords, concepts, and tools for writing SQL code.
- Chapter 4 breaks down each clause of a SQL query.

II. Database Objects, Data Types, and Functions

- Chapter 5 lists common ways to create and modify objects within a database.
- Chapter 6 lists common data types that are used in SQL.
- Chapter 7 lists common operators and functions in SQL.

III. Advanced Concepts

- Chapters 8 and 9 explain advanced querying concepts including joins, case statements, window functions, etc.
- Chapter 10 walks through solutions to some of the most commonly searched for SQL questions.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user, or values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*SQL Pocket Guide, 4th ed.* by Alice Zhao (O'Reilly). Copyright 2021 Alice Zhao, 978-1-492-209040-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/jreAj*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*.

Follow us on Twitter: *http://twitter.com/oreillymedia*.

Watch us on YouTube: *http://youtube.com/oreillymedia*.

Acknowledgments

Thank you to Jonathan Gennick for creating this pocket guide from scratch and writing the first three editions, and to Andy Kwan for trusting me to continue on with the publication.

I couldn't have completed this book without the help of my editors Amelia Blevins, Jeff Bleiel, and Caitlin Ghegan, and my technical reviewers Alicia Nevels, Joan Wang, Scott Haines, and Thomas Nield. I truly appreciate the time you've spent reading each page of this book. Your feedback has been invaluable.

To my parents—thank you for fostering my love for learning and creating. To my kids Henry and Lily—your excitement for this book warms my heart. Finally, to my husband, Ali—thank you for all of your notes on this book, for your encouragement, and for being my biggest fan.

CHAPTER 1

SQL Crash Course

This short chapter is intended to quickly get you up to speed on basic SQL terminology and concepts.

What Is a Database?

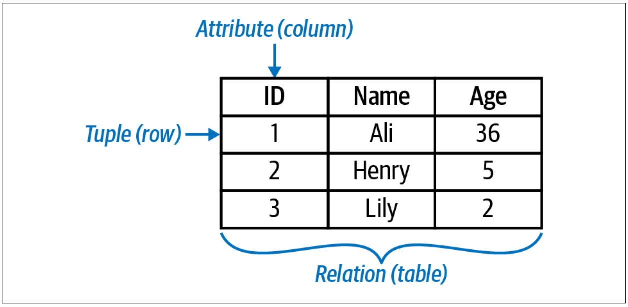
Let's start with the basics. A *database* is a place to store data in an organized way. There are many ways to organize data, and as a result, there are many databases to choose from. The two categories that databases fall into are *SQL* and *NoSQL*.

SQL

SQL is short for *Structured Query Language*. Imagine you have an app that remembers all of your friend's birthdays. SQL is the most popular language you would use to talk to that app.

> *English*: "Hey app. When is my husband's birthday?"
> *SQL*: `SELECT * FROM birthdays`
> `WHERE person = 'husband';`

SQL databases are often called *relational databases* because they are made up of relations, which are more commonly referred to as tables. Many tables connected to each other make up a database. Figure 1-1 shows a picture of a relation in a SQL database.



ID	Name	Age
1	Ali	36
2	Henry	5
3	Lily	2

Figure 1-1. A relation (also known as a table) in a SQL database

The main thing to note about SQL databases is that they require predefined *schemas*. You can think of a schema as the way that data in a database is organized or structured. Let's say you'd like to create a table. Before loading any data into the table, the structure of the table must first be decided on, including things like what columns are in the table, whether those columns hold integer or decimal values, etc.

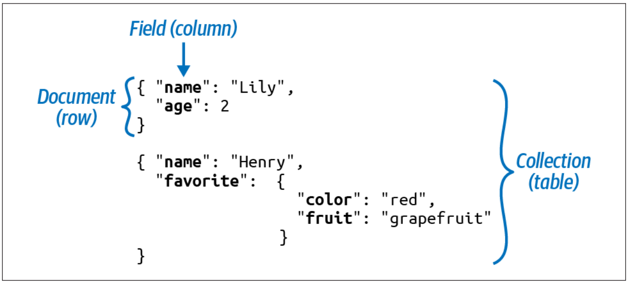
There comes a time, though, when data cannot be organized in such a structured way. Your data may have varying fields or you may need a more effective way of storing and accessing a large amount of data. That's where NoSQL comes in.

NoSQL

NoSQL stands for *not only SQL*. It will not be covered in detail in this book, but I wanted to point it out because the term has grown a lot in popularity since the 2010s and it's important to understand there are ways to store data beyond just tables.

NoSQL databases are often referred to as *non-relational databases*, and they come in all shapes and sizes. Their main characteristics are that they have dynamic schemas (meaning the schema doesn't have to be locked in up front) and they allow for horizontal scaling (meaning the data can spread across multiple machines).

The most popular NoSQL database is *MongoDB*, which is more specifically a document database. Figure 1-2 shows a picture of how data is stored in MongoDB. You'll notice that the data is no longer in a structured table and the number of fields (similar to a column) varies for each document (similar to a row).



```
{ "name": "Lily",
  "age": 2
}

{ "name": "Henry",
  "favorite":  {
                 "color": "red",
                 "fruit": "grapefruit"
               }
}
```

Figure 1-2. A collection (a variant of a table) in MongoDB, a NoSQL database

That all said, the focus of this book is on SQL databases. Even with the introduction of NoSQL, most companies still store the majority of their data in tables in relational databases.

Database Management Systems (DBMS)

You may have heard terms like *PostgreSQL* or *SQLite*, and be wondering how they are different from SQL. They are two types of *Database Management Systems* (DBMS), which is software used to work with a database.

This includes things like figuring out how to import data and organize it, as well as things like managing how users or other programs access the data. A *Relational Database Management System* (RDBMS) is software that is specifically for relational databases, or databases made up of tables.

Each RDBMS has a different implementation of SQL, meaning that the syntax varies slightly from software to software. For example, this is how you would output 10 rows of data in 5 different RDBMSs:

MySQL, PostgreSQL, and SQLite

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

Microsoft SQL Server

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

Oracle Database

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

Googling SQL Syntax

When searching for SQL syntax online, always include the RDBMS you are working with in the search. When I first learned SQL, I could not for the life of me figure out why my copy-pasted code from the internet didn't work and this was the reason!

Do this.

Search: *create table datetime* ***postgresql***

→ Result: `timestamp`

Search: *create table datetime* ***microsoft sql server***

→ Result: `datetime`

Not this.

Search: *create table datetime*

→ Result: syntax could be for any RDBMS

This book covers SQL basics along with the nuances of five popular database management systems: Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL and SQLite.

Some are proprietary, meaning they are owned by a company and cost money to use, and others are open source, meaning they are free for anyone to use. Table 1-1 details the differences between the RDBMSs.

Table 1-1. RDBMS comparison table

RDBMS	Owner	Highlights
Microsoft SQL Server	Microsoft	- Popular proprietary RDBMS - Often used alongside other Microsoft products including Microsoft Azure and the .NET framework - Common on the Windows platform - Also referred to as *MSSQL* or *SQL Server*
MySQL	Open Source	- Popular open source RDBMS - Often used alongside web development languages like HTML/CSS/Javascript - Acquired by Oracle, though still open source
Oracle Database	Oracle	- Popular proprietary RDBMS - Often used at large corporations given the amount of features, tools, and support available - Also referred to simply as *Oracle*
PostgreSQL	Open Source	- Quickly growing in popularity - Often used alongside open source technologies like Docker and Kubernetes - Efficient and great for large datasets
SQLite	Open Source	- World's most used database engine - Common on iOS and Android platforms - Lightweight and great for a small database

NOTE

Going forward in this book:

- Microsoft SQL Server will be referred to as *SQL Server*.
- Oracle Database will be referred to as *Oracle*.

Installation instructions and code snippets for each RDBMS can be found in RDBMS Software in Chapter 2.

A SQL Query

A common acronym in the SQL world is *CRUD*, which stands for "Create, Read, Update, and Delete." These are the four major operations that are available within a database.

SQL Statements

People who have *read and write access* to a database are able to perform all four operations. They can create and delete tables, update data in tables, and read data from tables. In other words, they have all the power.

They write *SQL statements*, which is general SQL code that can be written to perform any of the CRUD operations. These people often have titles like *database administrator* (DBA) or *database engineer*.

SQL Queries

People who have *read access* to a database are only able to perform the read operation, meaning they can look at data in tables.

They write *SQL queries*, which are a more specific type of SQL statement. Queries are used for finding and displaying data, otherwise known as "reading" data. This action is sometimes referred to as *querying tables*. These people often have titles like *data analyst* or *data scientist*.

The next two sections are a quick-start guide for writing SQL queries, since it is the most common type of SQL code that you'll see. More details on creating and updating tables can be found in Chapter 5.

The SELECT Statement

The most basic SQL query (which will work in any SQL software) is:

```
SELECT * FROM my_table;
```

which says, show me all of the data within the table named `my_table`—all of the columns and all of the rows.

While SQL is case-insensitive (`SELECT` and `select` are equivalent), you'll notice that some words are in all caps and others are not.

- The uppercase words in the query are called *keywords*, meaning that SQL has reserved them to perform some sort of operation on the data.
- All other words are lowercase. This includes table names, column names, etc.

The uppercase and lowercase formats are not enforced, but it is a good style convention to follow for readability's sake.

Let's go back to this query:

```
SELECT * FROM my_table;
```

Let's say that instead of returning all of the data in its current state, I want to:

- Filter the data
- Sort the data

This is where you would modify the `SELECT` statement to include a few more *clauses*, and the result would look something like this:

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

More details on all of the clauses can be found in Chapter 4, but the main thing to note is this: the clauses must always be listed in the same order.

Memorize This Order

All SQL queries will contain some combination of these clauses. If you remember nothing else, remember this order!

```
SELECT       -- columns to display
FROM         -- table(s) to pull from
WHERE        -- filter rows
GROUP BY     -- split rows into groups
HAVING       -- filter grouped rows
ORDER BY     -- columns to sort
```

NOTE

The `--` is the start of a comment in SQL, meaning the text after it is just for documentation's sake and the code will not be executed.

For the most part, the `SELECT` and `FROM` clauses are required and all other clauses are optional. The exception is if you are selecting a particular database function, then only the `SELECT` is required.

The classic mnemonic to remember the order of the clauses is:

Sweaty feet will give horrible odors.

If you don't want to think about sweaty feet each time you write a query, here's one that I made up:

Start Fridays with grandma's homemade oatmeal.

Order of Execution

The order that SQL code is executed is not something typically taught in a beginner SQL course, but I'm including it here because it's a common question I received when I taught SQL to students coming from a Python coding background.

A sensible assumption would be that the order that you *write* the clauses is the same order that the computer *executes* the clauses, but that is not the case. After a query is run, this is the order that the computer works through the data:

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

Compared to the order that you actually write the clauses, you'll notice that the `SELECT` has been moved to the fifth position. The high-level takeaway here is that SQL works in this order:

1. Gathers all of the data with the `FROM`
2. Filters rows of data with the `WHERE`
3. Groups rows together with the `GROUP BY`
4. Filters grouped rows with the `HAVING`
5. Specifies columns to display with the `SELECT`
6. Rearranges the results with the `ORDER BY`

A Data Model

I'd like to spend the final section of the crash course going over a simple *data model* and point out some terms that you'll often hear in fun SQL conversations around the office.

A data model is a visualization that summarizes how all of the tables in a database are related to one another, along with some details about each table. Figure 1-3 is a simple data model of a student grades database.

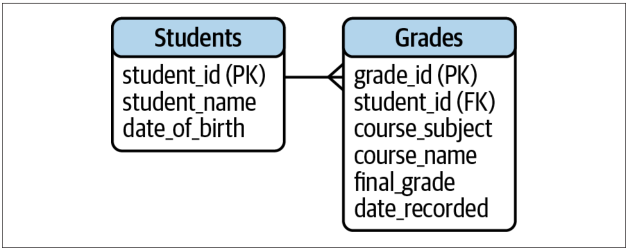


Figure 1-3. A data model of student grades

Table 1-2 lists out the technical terms that describe what's happening in the data model.

Table 1-2. Terms used to describe what's in a data model

Term	Definition	Example
Database	A database is a place to store data in an organized way.	This data model shows all of the data in the student grades database.
Table	A table is made up of rows and columns. In the data model, they are represented by rectangles.	There are two tables in the student grades database: `Students` and `Grades`.

Term	Definition	Example
Column	A table consists of multiple columns, which are sometimes referred to as attributes or fields. Each column contains a particular type of data. In the data model, all of the columns in a table are listed within each rectangle.	In the `Students` table, the columns are `student_id`, `student_name`, and `date_of_birth`.
Primary Key	A *primary key* uniquely identifies each row of data in a table. A primary key can be made up of one or more columns in a table. In a data model, it is flagged as `pk` or with a key icon.	In the `Students` table, the primary key is the `student_id` column, meaning that the `student_id` value is different for each row of data.
Foreign Key	A *foreign key* in a table refers to a primary key in another table. The two tables can be linked together by the common column. A table can have multiple foreign keys. In a data model, it is flagged as `fk`.	In the `Grades` table, `student_id` is a foreign key, meaning that the values in that column match up with values in the corresponding primary key column in the `Students` table.
Relationship	A *relationship* describes how the rows in one table map to the rows in another table. In a data model, it is represented by a line with symbols at the end points. Common types are one-to-one and one-to-many relationships.	In this data model, the two tables have a one-to-many relationship represented by the fork. One student can have many grades, or one row of the `Students` table maps to multiple rows in the `Grades` table.

More details on these terms can be found in "Creating Tables" on page 97 in Chapter 5.

You might be wondering why we're spending so much time reading a data model instead of writing SQL code already! The reason is because you'll often be writing queries that link up a number of tables, so it's a good idea to first get familiar with the data model to know how they all connect.

Data models can typically be found in a documentation repository at a company. You may want to print out the data models that you frequently work with—both for easy reference and easy desk decor.

You can also write queries within an RDBMS to look up information contained in a data model, such as the tables in a database, the columns of a table, or the constraints of a table.

And That Is Your Crash Course!

The remainder of this book is intended to be a reference book and does not need to be read in order. Please use it to look up concepts, keywords, and standards.

CHAPTER 2

Where Can I Write SQL Code?

This chapter covers three places where you can write SQL code:

RDBMS Software
: To write SQL code, you first have to download an RDBMS like MySQL, Oracle, PostgreSQL, SQL Server, or SQLite. The nuances of each RDBMS are highlighted in "RDBMS Software" on page 14.

Database Tools
: Once you've downloaded an RDBMS, the most basic way to write SQL code is through a *terminal window*, which is a text-only black-and-white screen. Most people prefer to use a *database tool* instead, which is a more user-friendly application that connects to an RDBMS behind the scenes.

  A database tool will have a *graphical user interface* (GUI), which allows users to visually explore tables and more easily edit SQL code. "Database Tools" on page 20 goes through how to connect a database tool to an RDBMS.

Other Programming Languages
: SQL can be written within many other programming languages. This chapter focuses on two in particular: Python and R. They are popular open source programming

languages used by data scientists and data analysts, who often need to write SQL code as well.

Instead of switching back and forth between Python/R and an RDBMS, you can connect Python/R directly to an RDBMS and write SQL code within Python/R. "Other Programming Languages" on page 24 walks through how to do so step by step.

RDBMS Software

This section includes installation instructions and short code snippets for the five RDBMSs that are covered in this book.

Which RDBMS to Choose?

If you are working at a company that is already using an RDBMS, you will need to use the same one.

If you are working on a personal project, you will need to decide which RDBMS to use. You can refer back to Table 1-1 in Chapter 1 to review the details of some popular ones.

Quick Start with SQLite

Want to start writing SQL code as soon as possible? SQLite is the fastest RDBMS to set up.

Compared to the other RDBMSs in this book, it's less secure and can't handle multiple users, but it provides basic SQL functionality in a compact package.

Because of this, I've moved SQLite up to the front of each section of this chapter since its setup is generally more straightforward than the others.

What Is a Terminal Window?

I'll often refer to a terminal window in this chapter because once you've downloaded an RDBMS, it's the most basic way to interact with the RDBMS.

A *terminal window* is an application on your computer that typically has a black background and only allows text inputs. The application name varies by operating system:

- On Windows, use the Command Prompt application.
- On macOS and Linux, use the Terminal application.

Once you open up a terminal window, you'll see a *command prompt*, which looks like a `>` followed by a flashing box. This means that it's ready to take in text commands from the user.

TIP

The next sections include links to download RDBMS installers for Windows, macOS, and Linux.

On macOS and Linux, an alternative to downloading an installer is to use the Homebrew package manager instead. Once you install Homebrew, you can run simple `brew install` commands from the Terminal to do all of the RDBMS installations.

SQLite

SQLite is free and the most lightweight install, meaning that it doesn't take up much space on your computer and is extremely quick to set up. For Windows and Linux, SQLite Tools can be downloaded from the SQLite Download Page. macOS comes with SQLite already installed.

TIP

The simplest way to start using SQLite is to open a terminal window and type **`sqlite3`**. With this approach, however, everything is done in memory, meaning that changes will not be saved once you close SQLite.

```
> sqlite3
```

If you want your changes to be saved, you should connect to a database upon opening with the following syntax:

```
> sqlite3 my_new_db.db
```

The command prompt for SQLite looks like this:

```
sqlite>
```

Some quick code to test things out:

```
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
```

To show databases, show tables, and exit:

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

TIP

If you want to display column names in your output, type:

```
sqlite> .headers on
```

To hide them again, type:

```
sqlite> .headers off
```

MySQL

MySQL is free, even though it is now owned by Oracle. MySQL Community Server can be downloaded from the MySQL Community Downloads page. On macOS and Linux, alternatively, you can do the installation with Homebrew by typing **`brew install mysql`** in the Terminal.

The command prompt for MySQL looks like this:

```
mysql>
```

Some quick code to test things out:

```
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)
```

To show databases, switch databases, show tables, and exit:

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

Oracle

Oracle is proprietary and works on Windows and Linux machines. Oracle Database Express Edition, the free edition, can be downloaded from the Oracle Database XE Downloads page.

The command prompt for Oracle looks like this:

```
SQL>
```

Some quick code to test things out:

```
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
```

To show databases, show all tables (including system tables), show user-created tables, and exit:

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

PostgreSQL

PostgreSQL is free and often used alongside other open source technologies. PostgreSQL can be downloaded from the PostgreSQL Downloads page. On macOS and Linux, alternatively, you can do the installation with Homebrew by typing **`brew install postgresql`** in the Terminal.

The command prompt for PostgreSQL looks like this:

```
postgres=#
```

Some quick code to test things out:

```
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)
```

To show databases, switch databases, show tables, and exit:

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

TIP

If you ever see `postgres-#`, that means that you've forgotten a semicolon at the end of a SQL statement. Type `;` and you should see `postgres=#` again.

If you ever see `:`, that means you've been automatically switched to the vi text editor, and you can exit by typing `q`.

SQL Server

SQL Server is proprietary (owned by Microsoft) and works on Windows and Linux machines. It can also be installed via Docker. SQL Server Express, the free edition, can be downloaded from the Microsoft SQL Server Downloads page.

The command prompt for SQL Server looks like this:

```
1>
```

Some quick code to test things out:

```
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)
```

To show databases, switch databases, show tables, and exit:

```
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
```

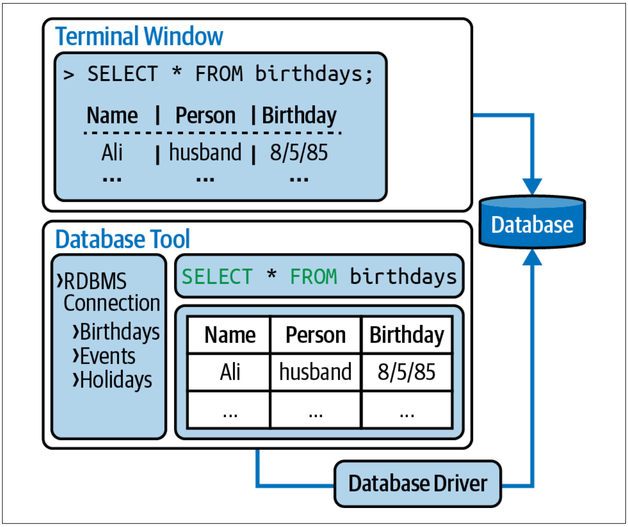
NOTE

In *SQL Server*, SQL code is not executed until you type the `go` command on a new line.

Database Tools

Instead of working with an RDBMS directly, most people will use a database tool to interact with a database. A database tool comes with a nice graphical user interface that allows you to point, click, and write SQL code in a user-friendly setting.

Behind the scenes, a database tool uses a *database driver*, which is software that helps the database tool talk to a database. Figure 2-1 shows the visual differences between accessing a database directly through a terminal window versus indirectly through a database tool.



```
> SELECT * FROM birthdays;

  Name  | Person  | Birthday
  -----------------------------
   Ali  | husband |  8/5/85
   ...      ...       ...
```

```
SELECT * FROM birthdays
```

Name	Person	Birthday
Ali	husband	8/5/85
...	...	...

Figure 2-1. Accessing an RDBMS through a terminal window versus a database tool

There are a number of database tools available. Some work specifically with a single RDBMS, and others work with multiple RDBMSs. Table 2-1 lists each RDBMS along with one of the most popular database tools for that particular RDBMS. All of the database tools in the table are free to download and use, and there are many other proprietary ones out there as well.

Table 2-1. Database tool comparison table

RDBMS	Database Tool	Details
SQLite	DB Browser for SQLite	- Different developer than SQLite - One of many tool options for SQLite
MySQL	MySQL Workbench	- Same developer as MySQL
Oracle	Oracle SQL Developer	- Developed by Oracle
PostgreSQL	pgAdmin	- Different contributors than PostgreSQL - Included with the PostgreSQL install

RDBMS	Database Tool	Details
SQL Server	SQL Server Management Studio	- Developed by Microsoft
Multiple	DBeaver	- One of many tool options for connecting to a variety of RDBMSs (including any of the preceding five listed)

Connect a Database Tool to a Database

When opening up a database tool, the first step is to connect to a database. This can be done in several ways:

Option 1: Create a New Database
: You can create a brand-new database by writing a `CREATE` statement:

```
CREATE DATABASE my_new_db;
```

Afterward, you can create tables to populate the database. More details can be found in "Creating Tables" on page 97 in Chapter 5.

Option 2: Open Up a Database File
: You may have downloaded or been given a file with a *.db* extension:

```
my_new_db.db
```

This *.db* file will already contain a number of tables. You can simply open it up within a database tool and start interacting with the database.

Option 3: Connect to an Existing Database
: You may want to work with a database that is either on your computer or on a *remote server*, meaning that the data is on a computer located elsewhere. This is extremely common these days with *cloud computing*, where people use servers owned by companies like Amazon, Google, or Microsoft.

Database Connection Fields

To connect to a database, you'll need to fill out the following fields within a database tool:

Host
: Where the database is located.

  - If the database is on your computer, then this should be *localhost* or *127.0.0.1*.
  - If the database is on a remote server, then this should be the IP address of that computer example: *123.45.678.90*.

Port
: How to connect to the RDBMS.

  There should already be a default port number in this field, and you shouldn't change it. It will be different for each RDBMS.

  - MySQL: *3306*
  - Oracle: *1521*
  - PostgreSQL: *5432*
  - SQL Server: *1433*

Database
: The name of the database you'd like to connect to.

Username
: Your username for the database.

  There may already be a default username in this field. If you don't remember setting up a username, keep the default value.

Password
: Your password associated with the username.

  If you don't remember setting up a password for your username, try leaving this field blank.

NOTE

For *SQLite*, instead of filling out these five database connection fields, you would enter in the file path of the *.db* database file you are trying to connect to.

Once you fill in the database connection fields correctly, you should have access to the database. You can now use the database tool to find the tables and fields you are interested in, and start writing SQL code.

Other Programming Languages

SQL can be written within a number of other programming languages. This chapter focuses on two popular open source ones: Python and R.

As a data scientist or data analyst, you likely do your analysis in Python or R, and also need to write SQL queries to pull data from a database.

A Basic Data Analysis Workflow

1. Write a SQL query within a database tool.
2. Export the results as a *.csv* file.
3. Import the *.csv* file into Python or R.
4. Continue doing analysis in Python or R.

The preceding approach is fine for doing a quick, one-time export. However, if you need to continuously edit your SQL query or are working with multiple queries, this can get annoying very quickly.

A Better Data Analysis Workflow

1. Connect Python or R to a database.
2. Write SQL queries within Python or R.
3. Continue doing analysis in Python or R.

This second approach allows you to do all of your querying and analysis within one tool, which is helpful if you need to tweak your queries as you are doing analysis. The remainder of this chapter provides code for each step of this second workflow.

Connect Python to a Database

It takes three steps to connect Python to a database:

1. Install a database driver for Python.
2. Set up a database connection in Python.
3. Write SQL code in Python.

Step 1: Install a database driver for Python

A database driver is software that helps Python talk to a database, and there are many driver options to choose from. Table 2-2 includes code for how to install a popular driver for each RDBMS.

This is a one-time installation you'll need to do via either a **`pip install`** or a **`conda install`**. The following code should be run in a terminal window.

Table 2-2. Install a driver for Python using either pip or conda

RDBMS	Option	Code
SQLite	n/a	No install necessary (Python 3 comes with 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`
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`

Step 2: Set up a database connection in Python

To set up a database connection, you first need to know the location and name of the database you are trying to connect to, as well as your username and password. More details can be found in "Database Connection Fields" on page 23.

Table 2-3 contains the Python code you need to run each time you plan on writing SQL code in Python. You can include it at the top of your Python script.

Table 2-3. Python code to set up a database connection

RDBMS	Code

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')`

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')`

TIP

Not all arguments are required. If you exclude an argument completely, then the default value will be used. For example, the default host is *localhost*, which is your computer. If no username and password were set up, then those arguments can be left out.

Keeping Your Passwords Safe in Python

The preceding code is fine for testing out a connection to a database, but in reality, you should not be saving your password within a script for everyone to see.

There are multiple ways to avoid doing so, including:

- generating an SSH key
- setting environment variables
- creating a configuration file

These options, however, all require additional knowledge of computers or file formats.

The recommended approach: create a separate Python file.

The most straightforward approach, in my opinion, is to save your username and password in a separate Python file, and then call that file within your database connection script. While this is less secure than the other options, it is the quickest start.

To use this approach, start by creating a *db_config.py* file with the following code:

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

Import the *db_config.py* file when setting up your database connection. The following example modifies the Oracle code from Table 2-3 to use the *db_config.py* values instead of hardcoded user and password values (changes are bolded):

```
import cx_Oracle
import db_config

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

Step 3: Write SQL code in Python

Once the database connection has been established, you can start writing SQL queries within your Python code.

Write a simple query to test your database connection:

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

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

WARNING

When using `cx_Oracle` in Python, remove the semicolon (`;`) at the end of all queries to avoid getting an error.

Save the results of a query as a pandas dataframe:

```
# 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'>
```

Close the connection when you are done using the database:

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

It is always good practice to close the database connection to save resources.

SQLAlchemy for Python Lovers

Another popular way to connect to a database is using the SQL-Alchemy package in Python. It is an *object relational mapper* (ORM), which turns database data into Python objects, allowing you to code in pure Python instead of using SQL syntax.

Imagine you want to see all the table names in a database. (The following code is PostgreSQL-specific, but SQLAlchemy will work with any RDBMS.)

Without SQLAlchemy:

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

With SQLAlchemy:

```
conn.table_names()
```

When using SQLAlchemy, the `conn` object comes with a `table_names()` Python method, which you may find easier to remember than SQL syntax. While SQLAlchemy provides cleaner Python code, it does slow down performance due to the additional time it spends turning data into Python objects.

To use SQLAlchemy in Python:

1. You must already have a database driver (like `psycopg2`) installed.
2. In a terminal window, type **`pip install sqlalchemy`** or a **`conda install -c conda-forge sqlalchemy`** to install SQLAlchemy.
3. Run the following code in Python to set up a SQLAlchemy connection. (The following code is PostgreSQL-specific.)

The SQLAlchemy documentation provides code for other RDBMSs and drivers:

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

Connect R to a Database

It takes three steps to connect R to a database:

1. Install a database driver for R
2. Set up a database connection in R
3. Write SQL code in R

Step 1: Install a database driver for R

A database driver is software that helps R talk to a database, and there are many driver options to choose from. Table 2-4 includes code for how to install a popular driver for each RDBMS.

This is a one-time installation. The following code should be run in R.

Table 2-4. Install a driver for R

RDBMS	Code
SQLite	`install.packages("RSQLite")`
MySQL	`install.packages("RMySQL")`

Oracle	The `ROracle` package can be downloaded from the Oracle ROracle Downloads page.
	
	`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)`

RDBMS	Code
PostgreSQL	`install.packages("RPostgres")`
SQL Server	On Windows, the `odbc` (Open Database Connectivity) package is pre-installed. On macOS and Linux, it can be downloaded from the Microsoft ODBC page. `install.packages("odbc")`

Step 2: Set up a database connection in R

To set up a database connection, you first need to know the location and name of the database you are trying to connect to, as well as your username and password. More details can be found in "Database Connection Fields" on page 23.

Table 2-5 contains the R code you need to run each time you plan on writing SQL code in R. You can include it at the top of your R script.

Table 2-5. R code to set up a database connection

RDBMS	Code

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")`

RDBMS	Code
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")`

TIP

Not all arguments are required. If you exclude an argument completely, then the default value will be used.

- For example, the default host is *localhost*, which is your computer.
- If no username and password were set up, then those arguments can be left out.

Keeping Your Passwords Safe in R

The preceding code is fine for testing out a connection to a database, but in reality, you should not be saving your password within a script for everyone to see.

There are multiple ways to avoid doing so, including:

- encrypting credentials with the `keyring` package

- creating a configuration file with the `config` package
- setting up environment variables with an *.Renviron* file
- recording the user and password as a global option in R with the `options` command

The recommended approach: prompt the user for a password.

The most straightforward approach, in my opinion, is to have RStudio prompt you for your password instead.

Instead of this:

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

Do this:

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

Step 3: Write SQL code in R

Once the database connection has been established, you can start writing SQL queries within your R code.

Show all tables in the database:

```
dbListTables(con)
```

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

TIP

For *SQL Server*, include the schema name to limit the number of tables displayed—`dbListTables(con, schema="dbo")`. `dbo` stands for database owner and it is the default schema in SQL Server.

Take a look at the `test` table in the database:

```
dbReadTable(con, "test")

  id num
1  1 100
2  2 200
```

NOTE

For *Oracle*, the table name is case-sensitive. Since Oracle automatically converts table names to uppercase, you'll likely need to run the following instead: `dbReadTable(con, "TEST")`.

Write a simple query and output a dataframe:

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

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

Close the connection when you are done using the database.

```
dbDisconnect(con)
```

It is always good practice to close the database connection to save resources.

CHAPTER 3

The SQL Language

This chapter covers SQL fundamentals including its standards, key terms, and sublanguages, along with answers to the following questions:

- What is ANSI SQL and how is it different from SQL?
- What is a keyword versus a clause?
- Do capitalization and whitespace matter?
- What is there beyond the `SELECT` statement?

Comparison to Other Languages

Some people in the technology space don't consider SQL to be a real programming language.

While SQL stands for "Structured Query *Language*," you can't use it in the same way as some other popular programming languages like Python, Java, or C++. With those languages, you can write code to specify the exact steps that a computer should take to get a task done. This is called *imperative programming*.

In Python, if you want to sum up a list of values, you can tell the computer exactly *how* you want to do so. The following example code goes through a list, item by item, and adds each value to a running total, to finally calculate the total sum:

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

With SQL, instead of telling a computer exactly *how* you want to do something, you just describe *what* you want done, which in this case is to calculate the sum. Behind the scenes, SQL figures out how to optimally execute the code. This is called *declarative programming*.

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

The main takeaway here is that SQL is not a *general-purpose programming language* like Python, Java, or C++, which can be used for a variety of applications. Instead, SQL is a *special-purpose programming language*, specifically made for managing data in a relational database.

Extensions for SQL

At its core, SQL is a declarative language, but there are extensions that allow it to do more:

- *Oracle* has *procedural language SQL* (PL/SQL)
- *SQL Server* has *transact SQL* (T-SQL)

With these extensions, you can do things like group together SQL code into procedures and functions, and more. The syntax doesn't follow ANSI standards, but it makes SQL much more powerful.

ANSI Standards

The *American National Standards Institute* (ANSI) is an organization based in the United States that documents standards on everything from drinking water to nuts and bolts.

SQL became an ANSI standard in 1986. In 1989, they published a very detailed document of specifications (think hundreds of pages) on what a database language should be able to do and how it should be done. Every few years, the standards get updated, so that's why you'll hear terms like ANSI-89 and ANSI-92, which were different sets of SQL standards that were added in 1989 and 1992, respectively. The latest standard is ANSI SQL2016.

SQL Versus ANSI SQL Versus MySQL Versus . . .

SQL is the general term for structured query language.

ANSI SQL refers to SQL code that follows the ANSI standards and will run in any relational database management system (RDBMS) software.

MySQL is one of many RDBMS options. Within MySQL, you can write both ANSI code and MySQL-specific SQL code.

Other RDBMS options include *Oracle*, *PostgreSQL*, *SQL Server*, *SQLite*, and others.

Even with the standards, no two RDBMSs are exactly the same. While some aim to be fully ANSI compliant, they are all just partially ANSI compliant. Each vendor ends up choosing which standards to implement and which additional features to build that only work within their software.

Should I follow the standards?

Most of the basic SQL code you write adheres to ANSI standards. If you find code that does something complex using simple yet unfamiliar keywords, then there's a good chance it's outside of the standards.

If you work solely within one RDBMS, like *Oracle* or *SQL Server*, it is absolutely fine to not follow the ANSI standards and take advantage of all of the features of the software.

The issue comes when you have code working in one RDBMS that you want to use in another RDBMS. Non-ANSI code likely won't run in the new RDBMS and would need to be rewritten.

Let's say you have the following query that works in *Oracle*. It does not meet ANSI standards because the `DECODE` function is only available within *Oracle* and not other software. If I copy the query over to *SQL Server*, the code will not run:

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

The following query has the same logic, but uses a `CASE` statement instead, which is an ANSI standard. Because of this, it will work in *Oracle*, *SQL Server*, and other software:

```
-- Code that works in any RDBMS
SELECT item, CASE WHEN flag = 0 THEN 'No'
             ELSE 'Yes' END AS Yes_or_No
FROM items;
```

Which Standard Should I Choose?

The following two code blocks perform a join using two different standards. ANSI-89 was the first widely adopted standard, followed by ANSI-92, which included some major revisions.

```
-- 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;
```

If you're writing new SQL code, I would recommend either using the latest standard (which is currently ANSI SQL2016) or the syntax provided in the documentation of the RDBMS you are working in.

However, it's important to be aware of the earlier standards because you will likely come across older code if your company has been around for a few decades.

SQL Terms

Here is a block of SQL code that shows the number of sales each employee closed in 2021. We'll be using this code block to highlight a number of SQL terms.

```
-- Sales closed in 2021
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;
```

Keywords and Functions

Keywords and functions are terms built into SQL.

Keywords

A *keyword* is text that already has some meaning in SQL. All the keywords in the code block are bolded here:

```sql
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 Is Case-Insensitive

Keywords are typically capitalized for readability. However, SQL is case-insensitive, meaning that an uppercase `WHERE` and a lowercase `where` mean the same thing when the code is run.

Functions

A *function* is a special type of keyword. It takes in zero or more inputs, does something to the inputs, and returns an output. In SQL, a function is usually followed by parentheses, but not always. The two functions in the code block are bolded here:

```sql
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;
```

There are four categories of functions: numeric, string, datetime, and other:

- `COUNT()` is a numeric function. It takes in a column and returns the number of non-null rows (rows that have a value).
- `YEAR()` is a date function. It takes in a column of a date or datetime data type, extracts the years, and returns the values as a new column.

A list of common functions can be found in Table 7-2.

Identifiers and Aliases

Identifiers and aliases are terms that the user defines.

Identifiers

An *identifier* is the name of a database object, such as a table or a column. All identifiers in the code block are bolded here:

```
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;
```

Identifiers should start with a letter (`a-z` or `A-Z`), followed by any combination of letters, numbers, and underscores (_). Some software will allow additional characters such as `@`, `#`, and `$`.

For readability's sake, identifiers are typically lowercase while keywords are uppercase, although the code will run regardless of case.

TIP

As a best practice, identifiers should not be given the same name as an existing keyword. For example, you wouldn't want to name a column `COUNT` because that is already a keyword in SQL.

If you still choose to do so, you can avoid confusion by enclosing the identifier in double quotes. So instead of naming a column `COUNT`, you can name it `"COUNT"`, but it is best to use a completely different name altogether like `num_sales`.

MySQL uses backticks (` `` `) to enclose identifiers instead of double quotes (`""`).

Aliases

An *alias* renames a column or a table temporarily, only for the duration of the query. In other words, the new alias names will be displayed in the results of the query, but the original column names will remain unchanged in the tables you are querying from. All the aliases in the code block are bolded here:

```
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;
```

The standard is to use `AS` when renaming columns (`AS num_sales`) and no additional text when renaming tables (`e`). Technically, though, either syntax works for both columns and tables.

In addition to columns and tables, aliases are also useful if you'd like to temporarily name a subquery.

Statements and Clauses

These are ways to refer to subsets of SQL code.

Statements

A *statement* starts with a keyword and ends with a semicolon. This entire code block is called a `SELECT` statement because it starts with the keyword `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;
```

TIP

Many database tools that provide a graphical user interface do not require a semicolon (`;`) at the end of a statement.

The `SELECT` statement is the most popular type of SQL statement, and is often called a query instead because it finds data in a database. Other types of statements are covered in "Sublanguages" on page 50.

Clauses

A *clause* is a way to refer to a particular section of a statement. Here is our original `SELECT` statement:

```
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;
```

This statement contains four main clauses:

- `SELECT` clause

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

- `FROM` clause

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

- `WHERE` clause

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

- `GROUP BY` clause

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

In conversation, you'll often hear people refer to a section of a statement like "take a look at the tables in the `FROM` clause." It's a helpful way to zoom in on a particular section of the code.

NOTE

This statement actually has more clauses than the four listed. In grammar, a clause is a part of a sentence that contains a subject and a verb. So you could refer to the following:

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

as the `LEFT JOIN` clause if you want to get even more specific about the section of the code that you are referring to.

The six most popular clauses start with `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` and are covered in detail in Chapter 4.

Expressions and Predicates

These are combinations of functions, identifiers, and more.

Expressions

An *expression* can be thought of as a formula that results in a value. An expression in the code block was:

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

This expression includes a function (`COUNT`) and an identifier (`s.sale_id`). Together, they make an expression that says to count the number of sales.

Other examples of expressions are:

- `s.sale_id + 10` is a numeric expression that incorporates basic math operations.
- `CURRENT_DATE` is a datetime expression, simply a single function, that returns the current date.

Predicates

A *predicate* is a logical comparison that results in one of three values: `TRUE`/`FALSE`/`UNKNOWN`. They are sometimes called *conditional statements*. The three predicates in the code block are bolded here:

```
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;
```

Some things you'll notice from these examples are:

- The equal sign (=) is the most popular operator to compare values.

- The `NULL` stands for no value. When checking to see if a field has no value, instead of writing `= NULL`, you would write `IS NULL`.

Comments, Quotes, and Whitespace

These are punctuation marks with meaning in SQL.

Comments

A *comment* is text that is ignored when the code is run, like the following.

```
-- Sales closed in 2021
```

It is useful to include comments in your code so that other reviewers of your code (including your future self!) can quickly understand the intent of the code without reading all of it.

To comment out:

- A single line of text:

```
-- These are my comments
```

- Multiple lines of text:

```
/* These are
my comments */
```

Quotes

There are two types of quotes you can use in SQL, the single quote and the double quote.

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

Single Quotes: Strings

Take a look at `'Bob'`. Single quotes are used when referring to a string value. You will see far more single quotes in practice compared to double quotes.

Double Quotes: Identifiers

Take a look at `"This column"`. Double quotes are used when referring to an identifier. In this case, because there is a space in between `This` and `column`, the double quotes are necessary for `This column` to be interpreted as a column name. Without the double quotes, SQL would throw an error due to the space. That said, it is best practice to use `_` instead of spaces when naming columns to avoid using the double quotes.

NOTE

MySQL uses backticks (``` `` ```) to enclose identifiers instead of double quotes (`""`).

Whitespace

SQL does not care about the number of spaces between terms. Whether it's one space, a tab, or a new line, SQL will execute the query from the first keyword all the way to the semicolon at the end of the statement. The following two queries are equivalent.

```
SELECT * FROM my_table;

SELECT *
  FROM my_table;
```

NOTE

For simple SQL queries, you may see code all written on one line. For longer queries that have dozens or even hundreds of lines, you'll see new lines for new clauses, tabs when listing many columns or tables, etc.

The end goal is to have readable code, so you'll need to decide how you want to space out your code (or follow your company's guidelines) so that it looks clean and can be quickly skimmed.

Sublanguages

There are many types of statements that can be written within SQL. They all fall under one of five sublanguages, which are detailed in Table 3-1.

Table 3-1. SQL sublanguages

Sublanguage	Description	Common Commands	Reference Sections
Data Query Language (DQL)	This is the language that most people are familiar with. These statements are used to retrieve information from a database object, such as a table, and are often referred to as SQL queries.	`SELECT`	The majority of this book is dedicated to DQL
Data Definition Language (DDL)	This is the language used to define or create a database object, such as a table or an index.	`CREATE` `ALTER` `DROP`	Creating, Updating, and Deleting
Data Manipulation Language (DML)	This is the language used to manipulate or modify data in a database.	`INSERT` `UPDATE` `DELETE`	Creating, Updating, and Deleting

Sublanguage	Description	Common Commands	Reference Sections
Data Control Language (DCL)	This is the language used to control access to data in a database, which are sometimes referred to as permissions or privileges.	`GRANT` `REVOKE`	Not covered
Transaction Control Language (TCL)	This is the language used to manage transactions in a database, or apply permanent changes to a database.	`COMMIT` `ROLLBACK`	Transaction Management

While most data analysts and data scientists will write DQL `SELECT` statements to query tables, it is important to know that database administrators and data engineers will also write code in these other sublanguages to maintain a database.

The SQL Language Summary

- ANSI SQL is standardized SQL code that works across all database software. Many RDBMSs have extensions that don't meet the standards but add functionality to their software.
- Keywords are terms that are reserved in SQL and have a special meaning.
- Clauses refer to particular sections of a statement. Common clauses are `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`.
- Capitalization and whitespace do not matter in SQL for execution, but there are best practices for readability.
- In addition to `SELECT` statements, there are commands for defining objects, manipulating data, and more.

CHAPTER 4

Querying Basics

A *query* is a nickname for a `SELECT` statement, which consists of six main clauses. Each section of this chapter covers a clause in detail:

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

The last section of this chapter covers the LIMIT clause, which is supported by *MySQL*, *PostgreSQL*, and *SQLite*.

The code examples in this chapter reference four tables:

`waterfall`
: waterfalls in Michigan's Upper Peninsula

`owner`
: owners of the waterfalls

`county`
: counties where the waterfalls are located

`tour`
: tours that consist of multiple waterfall stops

Here is a sample query that uses the six main clauses. It is followed by the query results, which are also known as the *result set*.

```
-- Tours with 2 or more public waterfalls
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
```

To *query* a database means to retrieve data from a database, typically from a table or multiple tables.

NOTE

It is also possible to query a *view* instead of a table. Views look like tables and are derived from tables, but they themselves do not hold any data. More on views can be found in "Views" on page 133 in Chapter 5.

The SELECT Clause

The SELECT clause specifies the columns that you want a statement to return.

In the SELECT clause, the SELECT keyword is followed by a list of column names and/or expressions that are separated by commas. Each column name and/or expression then becomes a column in the results.

Selecting Columns

The simplest SELECT clause lists one or more column names from the tables in the FROM clause:

```
SELECT id, name
FROM owner;

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

Selecting All Columns

To return all columns from a table, you can use a single asterisk rather than write out each column name:

```
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
```

WARNING

The asterisk is a helpful shortcut when testing out queries because it can save you quite a bit of typing. However, it is risky to use the asterisk in production code because the columns in a table may change over time, causing your code to fail when there are fewer or more columns than expected.

Selecting Expressions

In addition to simply listing columns, you can also list more complex expressions within the SELECT clause to return as columns in the results.

The following statement includes an expression to calculate a 10% drop in population, rounded to zero decimal places:

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

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

Selecting Functions

Expressions in the SELECT list typically refer to columns in the tables that you are pulling from, but there are exceptions. For example, a common function that doesn't refer to any tables is the one to return the current date:

```
SELECT CURRENT_DATE;

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

The preceding code works in *MySQL*, *PostgreSQL*, and *SQLite*. Equivalent code that works in other RDBMSs can be found in "Datetime Functions" on page 218 in Chapter 7.

NOTE

The majority of queries include both a SELECT and a FROM clause, but only the SELECT clause is required when using particular database functions, such as CURRENT_DATE.

It is also possible to include expressions within the SELECT clause that are *subqueries* (a query nested inside another query). More details can be found in "Selecting Subqueries" on page 61.

Aliasing Columns

The purpose of a *column alias* is to give a temporary name to any column or expression listed in the SELECT clause. That temporary name, or column alias, is then displayed as a column name in the results.

Note that this is not a permanent name change because the column names in the original tables remain unchanged. The alias only exists within the query.

This code displays three columns.

```
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
...
```

Let's say we want to rename the column names in the results. `id` is too ambigious and we'd like to give it a more descriptive name. `ROUND(population * 0.9, 0)` is too long and we'd like to give it a simpler name.

To create a column alias, you follow a column name or expression with either (1) an alias name or (2) the `AS` keyword and an alias name.

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

or:

```
-- 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
...
```

Both options are used in practice when creating aliases. Within the `SELECT` clause, the second option is more popular because the `AS` keyword makes it visually easier to differentiate column names and aliases among a long list of column names.

NOTE

Older versions of *PostgreSQL* require the use of `AS` when creating a column alias.

Although column aliases are not required, they are highly recommended when working with expressions to give sensible names to the columns in the results.

Aliases with case sensitivity and punctuation

As can be seen with the column aliases `county_id` and `estimated_pop`, the convention is to use lowercase letters with underscores in place of spaces when naming column aliases.

You can also create aliases containing uppercase letters, spaces, and punctuation using the double quote syntax, as shown in this example:

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

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

Qualifying Columns

Let's say you write a query that pulls data from two tables and they both contain a column called `name`. If you were to just include `name` in the `SELECT` clause, the code wouldn't know which table you were referring to.

To solve this problem, you can *qualify* a column name by its table name. In other words, you can give a column a prefix to specify which table it belongs to using *dot notation*, as in `table_name.column_name`.

The following example queries a single table, so while it isn't necessary to qualify the columns here, this is shown for demonstration's sake. This is how you would qualify a column by its table name:

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

TIP

If you get an error in SQL referencing an *ambiguous column name*, it means that multiple tables in your query have a column of the same name and you haven't specified which table/column combination you are referring to. You can resolve the error by qualifying the column name.

Qualifying tables

If you qualify a column name by its table name, you can also qualify that table name by its database or schema name. The following query retrieves data specifically from the `owner` table within the `sqlbook` schema:

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

The preceding code is lengthy since `sqlbook.owner` is repeated multiple times. To save on typing, you can provide a *table alias*. The following example gives the alias `o` to the table `owner`:

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

or:

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

Column Aliases Versus Table Aliases

Column aliases are defined within the `SELECT` clause to rename a column in the results. It is common to include `AS`, although not required.

```
-- Column alias
SELECT num AS new_col
FROM my_table;
```

Table aliases are defined within the FROM clause to create a temporary nickname for a table. It is common to exclude AS, although including AS also works.

```
-- Table alias
SELECT *
FROM my_table mt;
```

Selecting Subqueries

A *subquery* is a query that is nested inside another query. Subqueries can be located within various clauses, including the SELECT clause.

In the following example, in addition to the id, name, and population, let's say we also want to see the average population of all the counties. By including a subquery, we are creating a new column in the results for the average population.

```
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
...
```

A few things to note here:

- A subquery must be surrounded by parentheses.
- When writing a subquery within the SELECT clause, it is highly recommended that you specify a column alias,

which in this case was average_pop. That way, the column has a simple name in the results.

- There is only one value in the average_pop column that is repeated across all rows. When including a subquery within the SELECT clause, the result of the subquery must return a single column and either zero or one row, as shown in the following subquery to calculate the average population.

```
SELECT AVG(population) FROM county;

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

- If the subquery returned zero rows, then the new column would be filled with NULL values.

Noncorrelated Versus Correlated Subqueries

The preceding example is a *noncorrelated subquery*, meaning that the subquery does not refer to the outer query. The subquery can be run on its own independent of the outer query.

The other type of subquery is called a *correlated subquery*, which is one that does refer to values in the outer query. This often significantly slows down processing time, so it's best to rewrite the query using a JOIN instead. What follows is an example of a correlated subquery along with more efficient code.

Performance issues with correlated subqueries

The following query returns the number of waterfalls for each owner. Note the o.id = w.owner_id step in the subquery references the owner table in the outer query, making it a correlated subquery.

```
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
```

A better approach would be to rewrite the query with a JOIN. That way, the tables are first joined together and then the rest of the query is run, which is much faster than rerunning a subquery for each row of data. More on joins can be found in "Joining Tables" on page 270 in Chapter 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
```

DISTINCT

When a column is listed in the SELECT clause, by default, all of the rows are returned. To be more explicit, you can include the ALL keyword, but it is purely optional. The following queries return each type/open_to_public combination.

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

or:

```
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
```

If you want to remove duplicate rows from the results, you can use the `DISTINCT` keyword. The following query returns a list of unique `type`/`open_to_public` combinations.

```
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 and DISTINCT

To count the number of unique values within a *single column*, you can combine the `COUNT` and `DISTINCT` keywords within the `SELECT` clause. The following query returns the number of unique `type` values.

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

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

To count the number of unique combinations of *multiple columns*, you can wrap a DISTINCT query up as a subquery, and then do a COUNT on the subquery. The following query returns the number of unique type/open_to_public combinations.

```
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 and *PostgreSQL* support the use of the COUNT(DISTINCT) syntax on multiple columns. The following two queries are equivalent to the preceding query, without needing a subquery:

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

-- PostgreSQL equivalent
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
```

The FROM Clause

The FROM clause is used to specify the source of the data you want to retrieve. The simplest case is to name a single table or view in the FROM clause of query.

```
SELECT name
FROM waterfall;
```

You can qualify a table or view with either a database or schema name using the dot notation. The following query retrieves data specifically from the waterfall table within the sqlbook schema:

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

From Multiple Tables

Instead of retrieving data from one table, you'll often want to pull together data from multiple tables. The most common way to do this is using a JOIN clause within the FROM clause. The following query retrieves data from both the waterfall and tour tables and displays a single results table.

```
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
...
```

Let's break down each part of the code block.

Table aliases

```
waterfall w JOIN tour t
```

The `waterfall` and `tour` tables are given table aliases `w` and `t`, which are temporary names for the tables within the query. Table aliases are not required in a `JOIN` clause, but they are very helpful for shortening table names that need to be referenced within the `ON` and `SELECT` clauses.

JOIN … ON …

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

These two tables are pulled together with the `JOIN` keyword. A `JOIN` clause is always followed by an `ON` clause, which specifies how the tables should be linked together. In this case, the `id` of the waterfall in the `waterfall` table must match the `stop` with the waterfall in the `tour` table.

NOTE

You may see the `FROM`, `JOIN`, and `ON` clauses on different lines or indented. This is not required, but helpful for readability's sake, especially when you are joining many tables together.

Results table

A query always results in a single table. The `waterfall` table has 12 columns and the `tour` table has 3 columns. After joining these tables together, the results table has 15 columns.

```
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
...
```

You'll notice that there are two columns called `name` in the results table. The first is from the `waterfall` table, and the second is from the `tour` table. To refer to them in the `SELECT` clause, you would need to qualify the column names.

```
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
...
```

To differentiate the two columns, you would also want to alias the column names.

```
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
...
```

JOIN variations

In the preceding example, if a waterfall isn't listed in any tour, then it would not appear in the results table. If you wanted to see all waterfalls in the results, you would need to use a different type of join.

JOIN Defaults to INNER JOIN

This example uses a simple `JOIN` keyword to pull together data from two tables, although it is best practice to explicity state the type of join you are using. `JOIN` on its own defaults to an `INNER JOIN`, meaning that only records that are in both tables are returned in the results.

There are a variety of join types used in SQL, which are covered in more detail in "Joining Tables" on page 270 in Chapter 9.

From Subqueries

A subquery is a query that is nested inside another query. Subqueries within the `FROM` clause should be standalone `SELECT` statements, meaning that they do not reference the outer query at all and can be run on their own.

NOTE

A subquery within the `FROM` clause is also known as a *derived table* because the subquery ends up essentially acting like a table for the duration of the query.

The following query lists all publicly owned waterfalls, with the subquery portion bolded.

```
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
```

It is important to understand the order in which the query is executed.

Step 1: Execute the subquery

The contents of the subquery are first executed. You can see that this results in a table of only public owners:

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

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

Going back to the original query, you'll notice that the subquery is immediately followed by the letter `o`. This is the temporary name, or alias, that we are assigning to the results of the subquery.

NOTE

Aliases are required for subqueries within the `FROM` clause in *MySQL*, *PostgreSQL*, and *SQL Server*, but not in Oracle and SQLite.

Step 2: Execute the entire query

Next, you can think of the letter `o` taking the place of the subquery. The query is now executed as usual.

```
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
```

Subqueries Versus the WITH Clause

An alternative to writing a subquery is to write a common table expression (CTE) using a `WITH` clause instead. The advantage of the `WITH` clause is that the subquery is named up front, which makes for cleaner code and also the ability to reference the subquery multiple times.

```
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;
```

The `WITH` clause is supported by *MySQL 8.0+* (2018 and later), *PostgreSQL*, *Oracle*, *SQL Server*, and *SQLite*. "Common Table Expressions" on page 291 in Chapter 9 includes more examples of this technique.

Why Use a Subquery in the FROM Clause?

The main advantage of using subqueries is that you can turn a large problem into smaller ones. Here are two examples:

Example 1: Multiple steps to get to results

Let's say you want to find the average number of stops on a tour. First, you'd have to find the number of stops on each tour, and then average the results.

The following query finds the number of stops on each tour:

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

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

You could then turn the query into a subquery and write another query around it to find the average:

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

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

Example 2: Table in FROM clause is too large

The original goal was to list all publicly owned waterfalls. This can actually be done without a subquery and with a JOIN instead:

```
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
```

Let's say that the query takes a really long time to run. This can happen when you join massive tables together (think tens of millions of rows). There are multiple ways you could rewrite the query to speed it up, and one of them is to use a subquery.

Since we are only interested in public owners, we can first write a subquery that filters out all of the private owners. The smaller `owner` table would then be joined with the `waterfall` table, which would take less time and produce the same results.

```
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
```

These are just two of the many examples of how subqueries can be used to break down a larger query into smaller steps.

The WHERE Clause

The `WHERE` clause is used to restrict query results to only rows of interest, or simply put, it is the place to filter data. Rarely will you want to display all rows from a table, but rather rows that match specific criteria.

TIP

When exploring a table with millions of rows, you never want to do a `SELECT * FROM my_table;` because it will take an unnecessarily long time to run.

Instead, it's a good idea to filter down the data. Two common ways to do this are:

Filter by a column within the `WHERE` *clause*
: Better yet, filter by a column that is already indexed to make the retrieval even faster.

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

Show the top few rows of data with the `LIMIT` *clause (or* `WHERE ROWNUM <= 10` *in Oracle or* `SELECT TOP 10 *` *in SQL Server)*

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

The following query finds all waterfalls that do not contain *Falls* in the name. More on the `LIKE` keyword can be found in Chapter 7.

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

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

The bolded section is often referred to as a conditional statement or a predicate. The predicate makes a logical comparison for each row of data that results in `TRUE`/`FALSE`/`UNKNOWN`.

The `waterfall` table has 16 rows. For each row, it checks if the waterfall name contains *Falls* or not. If it doesn't contain *Falls*, then the `name NOT LIKE '%Falls%'` predicate is `TRUE`, and the row is returned in the results, which was the case for the two preceding rows.

Multiple Predicates

It is also possible to combine multiple predicates with *operators* like `AND` or `OR`. The following example shows waterfalls without *Falls* in its name and that also don't have an owner:

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

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

More details on operators can be found in Operators in Chapter 7.

Filtering on Subqueries

A subquery is a query nested inside another query, and the `WHERE` clause is a common place to find one. The following example retrieves publicly accessible waterfalls located in Alger County:

```
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
...
```

NOTE

Unlike subqueries within the `SELECT` clause or the `FROM` clause, subqueries in the `WHERE` clause do not require an alias. In fact, you will get an error if you include an alias.

Why use a subquery in the WHERE clause?

The original goal was to retrieve publicly accessible waterfalls located in Alger County. If you were to write this query from scratch, you would likely start with the following:

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

At this point, you have all waterfalls that are publicly accessible. The final touch is to find ones that are specifically in Alger County. You know that the `waterfall` table doesn't have a county name column, but the `county` table does.

You have two options to pull the county name into the results. You can either (1) write a subquery within the `WHERE` clause that specifically pulls the Alger County information or (2) join together the `waterfall` and `county` tables:

```
-- Subquery in WHERE clause
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');
```

or:

```
-- JOIN clause
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
...
```

The two queries produce the same results. The advantage of the first approach is that subqueries are often easier to understand than joins. The advantage of the second approach is that joins typically execute faster than subqueries.

Working > Optimizing

When writing SQL code, there are often multiple ways to do the same thing.

Your top priority should be to write *working* code. If it takes a long time to run or it's ugly, it doesn't matter…it works!

The next step, if you have time, is to *optimize* the code by improving the performance by perhaps rewriting it with a `JOIN`, making it more readable with indentations and capitalizations, etc.

Don't stress about writing the most optimized code up front, but rather writing code that works. Writing elegant code comes with experience.

Other Ways to Filter Data

The WHERE clause is not the only place within a SELECT statement to filter rows of data.

- FROM clause: When joining together tables, the ON clause specifies how they should be linked together. This is where you can include conditions to restrict rows of data returned by the query. See Joining Tables in Chapter 9 for more details.
- HAVING clause: If there are aggregations within the SELECT statement, the HAVING clause is where you specify how the aggregations should be filtered. See "The HAVING Clause" on page 83 for more details.
- LIMIT clause: To display a specific number of rows, you can use the LIMIT clause. In *Oracle*, this is done with WHERE ROWNUM and in *SQL Server*, this is done with SELECT TOP. See "The LIMIT Clause" on page 88 in this chapter for more details.

The GROUP BY Clause

The purpose of the GROUP BY clause is to collect rows into groups and summarize the rows within the groups in some way, ultimately returning just one row per group. This is sometimes referred to as "slicing" the rows into groups and "rolling up" the rows in each group.

The following query counts the number of waterfalls along each of the tours:

```
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
```

There are two parts to focus on here:

- *The collecting of rows*, which is specified within the GROUP BY clause
- *The summarizing of rows* within groups, which is specified within the SELECT clause

Step 1: The collecting of rows

In the GROUP BY clause:

```
GROUP BY t.name
```

we state that we would like to look at all of the rows of data and put the M-28 tour waterfalls into a group, all of the Munising tour waterfalls into a group, and so on. Behind the scenes, the data is being grouped like this:

```
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
```

Step 2: The summarizing of rows

In the `SELECT` clause,

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

we state that for each group, or each tour, we want to count the number of rows of data in the group. Because each row represents a waterfall, this would result in the total number of waterfalls along each tour.

The `COUNT()` function here is more formally known as an *aggregate function*, or a function that summarizes many rows of data into a single value. More aggregate functions can be found in "Aggregate Functions" on page 191 in Chapter 7.

WARNING

In this example, `COUNT(*)` returns the number of waterfalls along each tour. However, this is only because each row of data in the `waterfall` and `tour` tables represent a single waterfall.

If a single waterfall was listed on multiple rows, `COUNT(*)` would result in a larger value than expected. In this case, you could potentially use `COUNT(DISTINCT waterfall_name)` instead to find the unique waterfalls. More details can be found in `COUNT and DISTINCT`.

The key takeaway is that it is important to manually double-check the results of the aggregate function to make sure it is summarizing the data in the way that you intended.

Now that the groups have been created with the GROUP BY clause, the aggregate function will be applied once to each group:

```
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
```

Any columns to which an aggregate function has not been applied, which in this case is the tour_name column, are now collapsed into one value:

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

NOTE

This collapsing of many detail rows into one aggregate row means that when using a GROUP BY clause, the SELECT clause should *only* contain:

- All columns listed in the GROUP BY clause: t.name
- Aggregations: COUNT(*)

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

Not doing so could either result in an error message or return inaccurate values.

GROUP BY In Practice

These are the steps you should take when using a GROUP BY:

1. Figure out what column(s) you want to use to separate out, or group, your data (i.e., tour name).
2. Figure out how you'd like to summarize the data within each group (i.e. count the waterfalls within each tour).

When you've decided on those:

1. In the SELECT clause, list the column(s) you want to group by (i.e., tour name) and the aggregation(s) you want to calculate within each group (i.e., count of waterfalls).
2. In the GROUP BY clause, list all columns that are not aggregations (i.e., tour name).

For more complex grouping situations including ROLLUP, CUBE, and GROUPING SETS, go to "Grouping and Summarizing" on page 242 in Chapter 8.

The HAVING Clause

The HAVING clause places restrictions on the rows returned from a GROUP BY query. In other words, it allows you to filter on the results after a GROUP BY has been applied.

NOTE

A HAVING clause always immediately follows a GROUP BY clause. Without a GROUP BY clause, there can be no HAVING clause.

This is a query that lists the number of waterfalls on each tour using a GROUP BY clause:

```
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
```

Let's say we only want to list the tours that have exactly six stops. To do so, you would add a HAVING clause after the GROUP BY clause:

```
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
```

WHERE Versus HAVING

The purpose of both clauses is to filter data. If you are trying to:

- Filter on particular columns, write your conditions within the `WHERE` clause
- Filter on aggregations, write your conditions within the `HAVING` clause

The contents of a `WHERE` and `HAVING` clause cannot be swapped:

- Never put a condition with an aggregation in the `WHERE` clause. You will get an error.
- Never put a condition in the `HAVING` clause that does not involve an aggregation. Those conditions are evaluated much more efficiently in the `WHERE` clause.

You'll notice that the `HAVING` clause refers to the aggregation `COUNT(*)`,

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

and not the alias,

```
# code will not run
SELECT COUNT(*) AS num_waterfalls
...
HAVING num_waterfalls = 6;
```

The reason for this is because of the order of execution of the clauses. The `SELECT` clause is written before the `HAVING` clause. However, the `SELECT` clause is actually executed *after* the `HAVING` clause.

That means that the alias `num_waterfalls` in the `SELECT` clause does not exist at the time the `HAVING` clause is being executed. The `HAVING` clause must refer to the raw aggregation `COUNT(*)` instead.

NOTE

MySQL and *SQLite* are exceptions, and allow aliases (`num_waterfalls`) in the `HAVING` clause.

The ORDER BY Clause

The `ORDER BY` clause is used to specify how you want the results of a query to be sorted.

The following query returns a list of owners and waterfalls, without any sorting:

```
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
...
```

The COALESCE Function

The COALESCE function replaces all NULL values in a column with a different value. In this case, it turned the NULL values in the o.name column into the text Unknown.

If the COALESCE function were not used here, all waterfalls without owners would have been left out of the results. Instead, they are now marked as having an Unknown owner, and can be sorted on and included in the results.

More details can be found in Chapter 7.

The following query returns the same list, but first sorted alphabetically by owner, and then by waterfall:

```
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
...
```

The default sort is in ascending order, meaning text will go from A to Z and numbers will go from lowest to highest. You can use the keywords ASCENDING and DESCENDING (which can be abbreviated as ASC and DESC) to control the sort on each column.

The following is a modification of the previous sort, but this time, it sorts owner names in reverse order:

```
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
...
```

You can sort by columns and expressions that are not in your `SELECT` list:

```
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
...
```

You can also sort by numeric column position:

```
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
...
```

Because the rows of a SQL table are unordered, if you don't include an ORDER BY clause in a query, each time you execute the query, the results could be displayed in a different order.

ORDER BY Cannot Be Used in a Subquery

Of the six main clauses, only the ORDER BY clause cannot be used in a subquery. Unfortunately, you can't force the rows of a subquery to be ordered.

To avoid this issue, you would need to rewrite your query with different logic to avoid using an ORDER BY clause within the subquery, and only include an ORDER BY clause in the outer query.

The LIMIT Clause

When quickly viewing a table, it is best practice to return a limited number of rows instead of the entire table.

MySQL, *PostgreSQL*, and *SQLite* support the LIMIT clause. *Oracle* and *SQL Server* use different syntax with the same functionality:

```
-- MySQL, PostgreSQL, and 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
```

Another way to limit the number of rows returned is to filter on a column within the `WHERE` clause. The filtering will execute even faster if the column is indexed.

CHAPTER 5

Creating, Updating, and Deleting

The majority of this book covers how to read data from a database with SQL queries. Reading is one of the four basic database operations out of create, read, update, and delete (CRUD).

This chapter focuses on the remaining three operations for Databases, Tables, Indexes, and Views. In addition, the Transaction Management section covers how to execute multiple commands as a single unit.

Databases

A *database* is a place to store data in an organized way.

Within a database, you can create *database objects*, which are things that store or reference data. Common database objects include tables, constraints, indexes, and views.

A *data model* or a *schema* describes how database objects are organized within a database.

Figure 5-1 shows a database that contains many tables. The specifics around how the tables are defined (i.e., the `Sales` table contains five columns) and how they connect with one another (i.e., the `customer_id` column in the `Sales` table matches the `customer_id` column in the `Customer` table) are all a part of the *schema* of the database.

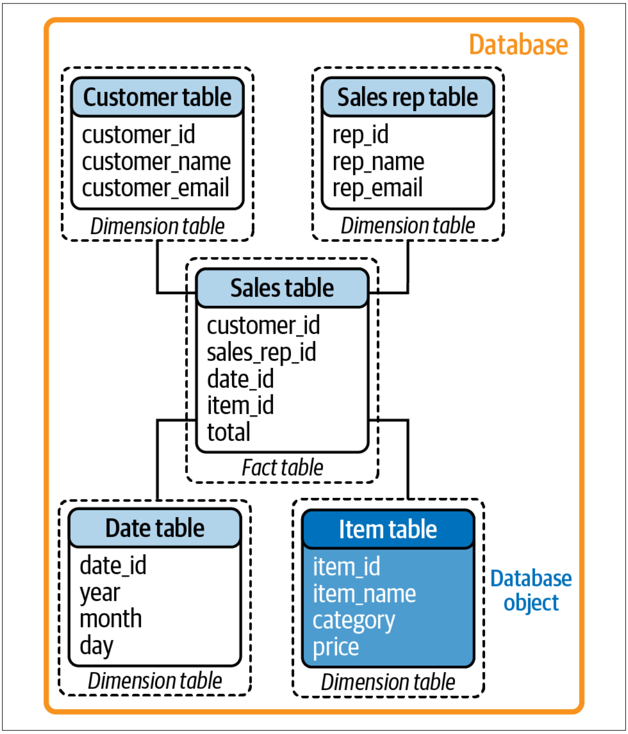


Figure 5-1. A database containing a star schema

The tables in Figure 5-1 are arranged in a *star schema*, which is a basic way of organizing tables in a database. The star schema includes a *fact table* in the center and is surrounded by *dimension tables* (also known as *lookup tables*). The fact table records transactions made (sales in this case) along with IDs of additional information, which are fully detailed out in the dimension tables.

Data Model Versus Schema

When designing a database, you would first come up with a *data model*, which is how you want your database organized at a high level. It could look like Figure 5-1 and include table names, how they are connected to one another, etc.

When you are ready to take action, you would create a *schema*, which is the implementation of the data model in a database. Within the software you are working in, you would specify the tables, constraints, primary and foreign keys, etc.

NOTE

The definition of a schema varies for some RDBMSs.

In *MySQL*, a schema is the same thing as a database and the two terms can be used interchangeably.

In *Oracle*, a schema consists of the database objects owned by a particular user, so the terms *schema* and *user* are used interchangeably.

Display Names of Existing Databases

All database objects reside in databases, so a good first step is to see what databases currently exist. Table 5-1 shows the code to display the names of all existing databases in each RDBMS.

Table 5-1. Code to display names of existing databases

RDBMS	Code
MySQL	`SHOW databases;`
Oracle	`SELECT * FROM global_name;`
PostgreSQL	`\l`
SQL Server	`SELECT name FROM master.sys.databases;`
SQLite	`.database` (or look in the file browser for *.db* files)

NOTE

SQLite: For most RDBMS software, databases are located within the RDBMS. However, for SQLite, databases are stored outside of SQLite as *.db* files. To use a database, you would specify a *.db* file name when launching SQLite:

```
> sqlite3 existing_db.db
```

Display Name of Current Database

You may want to confirm the database you are currently in before writing any queries. Table 5-2 shows the code to display the name of the database you are currently in for each RDBMS.

Table 5-2. Code to display name of current database

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

NOTE

You may have noticed that the current database code is the same as the existing database code for Oracle and SQLite.

An instance of *Oracle* can only connect to a single database at a time, and you typically don't switch databases.

With *SQLite*, you can only open up and work with a single database file at a time.

Switch to Another Database

You may want to use data in another database or switch to a newly created database. Table 5-3 shows the code to switch to another database in each RDBMS.

Table 5-3. Code to switch to another database

RDBMS	Code
MySQL, SQL Server	`USE another_db;`
Oracle	You typically don't switch databases (see earlier note), but to switch users, you would type: **`connect another_user`**
PostgreSQL	`\c another_db`
SQLite	`.open another_db`

Create a Database

If you have CREATE privileges, you can create a new database. If not, you may only be able to work within an existing database. Table 5-4 shows the code to create a database in each RDBMS.

Table 5-4. Code to create a database

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

NOTE

Oracle: There are some additional steps (regarding instances, environment variables, etc.) surrounding the `CREATE DATABASE` statement in Oracle, which can be found in the Oracle documentation.

SQLite: The `>` symbol is not a character that you actually type. It just signifies that this is command line code, not SQL code.

Delete a Database

If you have DELETE privileges, you can delete a database. Table 5-5 shows the code to delete a database in each RDBMS.

WARNING

If you delete a database, you will lose all of the data in the database. *There is no undo*, unless a backup has been created. I recommend not running this command unless you are 100% sure you don't need the database.

Table 5-5. Table 5-5. Code to delete a database

RDBMS	Code
MySQL, Oracle, PostgreSQL, SQL Server	`DROP DATABASE my_new_db;`
SQLite	Delete the *.db* file in the file browser

NOTE

Oracle: There are some additional steps (regarding mounting, etc.) surrounding the `DROP DATABASE` statement in Oracle, which can be found in the Oracle documentation.

In some RDBMSs, you can't drop a database you are currently in. You would have to first switch to another database, such as the default one, before dropping the database:

- In *PostgreSQL*, the default database is `postgres`:

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

- In *SQL Server*, the default database is `master`:

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

Creating Tables

Tables consist of rows and columns, and store all of the data in a database. In SQL, there are a few additional requirements for tables:

- Each row of a table should be unique
- All data in a column must be of the same data type (integer, text, etc.)

NOTE

In *SQLite*, the data in a column *does not* have to all be of the same data type. SQLite is more flexible in that each value has a data type associated with it, rather than an entire column.

To be compatible with other RDBMSs, SQLite does support columns having data type assignments. These *type affinities* are recommended data types for the columns, and are not required.

Create a Simple Table

It takes two steps to create a table in SQL. You must first define the structure of a table before loading data into it:

1. Create a table.

   The following code creates an empty table called `my_simple_table` with three columns: `id`, `country`, and `name`. All values in the first column (`id`) must be integers, and the other two columns (`country`, `name`) can contain up to 2 and up to 15 characters:

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

   More data types in addition to `INTEGER` and `VARCHAR` are listed in Chapter 6.

2. Insert rows.
   a. Insert a single row of data.

      The following code inserts one row of data into columns `id`, `country`, and `name`:

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

   b. Insert multiple rows of data.

Table 5-6 shows how to insert multiple rows of data into a table in each RDBMS, instead of one row at a time.

Table 5-6. Code to insert multiple rows of data

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

After inserting the data, the table would look like this:

```
SELECT * FROM my_simple_table;

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

When inserting rows of data, the order of the values must match the order of the column names exactly.

Values in any columns omitted from the column list will take on their default value of `NULL`, unless another default value is specified.

NOTE

You need CREATE privileges to create a table. If you get an error when running the preceding code, you do not have the permission to do so and need to talk to your database administrator.

Display Names of Existing Tables

Before creating a table, you may want to see if the table name already exists. Table 5-7 shows the code to display the names of existing tables in the database for each RDBMS.

Table 5-7. Code to display names of existing tables

RDBMS	Code
MySQL	`SHOW tables;`

Oracle	`-- All tables, including system tables`
	`SELECT table_name FROM all_tables;`
	
	`-- All user created tables`
	`SELECT table_name FROM user_tables;`

PostgreSQL	`\dt`

SQL Server	`SELECT table_name`
	`FROM information_schema.tables;`

SQLite	`.tables`

Create a Table That Does Not Already Exist

In *MySQL*, *PostgreSQL*, and *SQLite*, you can check for existing tables using the `IF NOT EXISTS` keywords when creating a table:

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

If the table name does not exist, a new table will get created. If the table name already exists, without the IF NOT EXISTS, you would get an error message. With the IF NOT EXISTS, no new table gets created and you would avoid an error message.

If you want to replace an existing table, there are two approaches to doing so:

- You could use DROP TABLE to completely delete the existing table, and then create a new one.
- You could *truncate* the existing table, meaning you keep the schema (aka structure) of the table, but clear out the data inside of it. This can be done by using DELETE FROM to delete data from the table.

Create a Table with Constraints

A *constraint* is a rule that specifies what data can be inserted into a table. The following code creates two tables and multiple constraints (bolded):

```
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))
   );
```

The CONSTRAINT keyword names the constraint for future reference and is optional. You should avoid using the same name for both a column and a constraint.

For quick access to the constraint sections: NOT NULL, DEFAULT, CHECK, UNIQUE, PRIMARY KEY, FOREIGN KEY.

Constraint: Not allowing NULL values in a column with NOT NULL

In a SQL table, cells without a value are replaced with the term NULL. For each column, you can specify whether NULL values are allowed or not:

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

The NOT NULL constraint on the id column means that the column will not allow NULL values. In other words, there can be no missing values inserted into the column, or else you will get an error message.

The NULL constraint on the country column means that the column will allow NULL values. If you are inserting data into the table and exclude the country column, then no value will be inserted, and the cell will be replaced with a NULL value.

By not specifying NULL or NOT NULL, the name column defaults to NULL, meaning it will allow NULL values.

Constraint: Setting default values in a column with DEFAULT

When inserting data into a table, missing values get replaced with the term `NULL`. To replace missing values with another value, you can use the `DEFAULT` constraint. The following code turns any missing `country` value into `CA`:

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

Constraint: Restricting values in a column with CHECK

You can restrict the values allowed in a column by using the `CHECK` constraint. The following code only allows values of `CA` and `US` in the `country` column.

You can place the `CHECK` keyword immediately after the column name and data type:

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

Or you can place the `CHECK` keyword after all the column names and data types:

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

You can also include logic that checks multiple columns:

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

Constraint: Requiring unique values in a column with UNIQUE

You can require the values of a column to be unique by using the `UNIQUE` constraint.

You can place the `UNIQUE` keyword immediately after the column name and data type:

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

Or you can place the `UNIQUE` keyword after all the column names and data types:

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

You can also include logic that forces the combination of multiple columns to be unique. The following code requires unique `country`/`name` combinations, meaning that one row can include `CA`/`Emma` and another can include `US`/`Emma`:

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

Create a Table with Primary and Foreign Keys

Primary keys and foreign keys are special types of constraints that uniquely identify rows of data.

Specify a primary key

A *primary key* uniquely identifies each row of data in a table. A primary key can be made up of one or more columns in a table. Every table should have a primary key.

You can place the PRIMARY KEY keywords immediately after the column name and data type:

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

Or you can place the PRIMARY KEY keywords after all the column names and data types:

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

To specify a primary key consisting of multiple columns (also known as a *composite key*):

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

By creating a `PRIMARY KEY`, the constraints that you are putting on the column(s) are that they cannot include `NULL` values (`NOT NULL`) and the values must be unique (`UNIQUE`).

Primary Key Best Practices

Every table should have a primary key. This ensures that every row can be uniquely identified.

It is recommended that primary keys consist of ID columns, like `(country_id, name_id)` instead of `(country, name)`. Technically, multiple rows could have the same `country` and `name` combination. By adding columns that contain unique IDs (`101`, `102`, etc.), the combination of `country_id` and `name_id` is guaranteed to be unique.

Primary keys should be immutable, meaning that they can't be changed. This allows for a particular row in a table to always be identified by the same primary key.

Specify a foreign key

A *foreign key* in a table refers to a primary key in another table. The two tables can be linked together by the common column. A table can have zero or more foreign keys.

Figure 5-2 shows a data model of two tables: the `customers` table, which has a primary key of `id`, and the `orders` table, which has a primary key of `o_id`. From the viewpoint of `customers`, its `order_id` column matches with values of the `o_id` column, making `order_id` a foreign key because it refers to a primary key in another table.

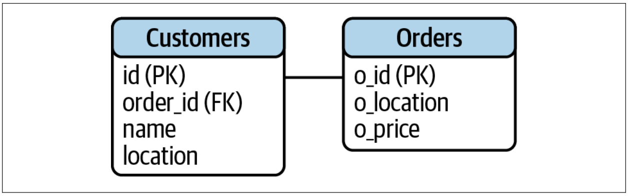


Figure 5-2. Two tables with primary keys and a foreign key

To specify a foreign key, use the following steps:

1. Locate the table you plan to reference and identify the primary key.

   In this case, we will be referencing `orders`, specifically the `o_id` column:

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

2. Create a table with a foreign key that references the primary key in the other table.

   In this case, we are creating the `customers` table where the `order_id` column references the `o_id` primary key in the `orders` table:

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

To specify a foreign key consisting of multiple columns, the primary key must consist of multiple columns as well:

```
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)
);
```

NOTE

The foreign key (`order_id`) and primary key it references (`o_id`) must both be the same data type.

Create a Table with an Automatically Generated Field

If you plan to load a dataset without a unique ID column, you may want to create a column that automatically generates a unique ID. The code in Table 5-8 automatically generates sequential numbers (1, 2, 3, etc.) in the `u_id` column, in each RDBMS.

Table 5-8. Code to automatically generate a unique ID

RDBMS	Code

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)
	);

SQL Server	-- u_id to begin at 1 and increment by 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)
	);

NOTE

In *Oracle*, VARCHAR2 is typically used instead of VARCHAR. They are identical in terms of functionality, but VARCHAR may one day be modified, so it's safer to use VARCHAR2.

SQLite recommends against using AUTOINCREMENT unless absolutely necessary because it uses additional computing resources. The code will still run without error.

Insert the Results of a Query into a Table

Instead of manually typing values to insert into a new table, you may want to load a new table with data from existing table(s).

Here is a table:

```
SELECT * FROM my_simple_table;

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

Create a new table with two columns:

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

Insert the results from a query into the new table:

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

The new table would then look like:

```
SELECT * FROM new_table_two_columns;

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

You can also insert values from an existing table and either add or modify other values along the way.

Create a new table with four columns:

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

Insert the results from a query into the new table and fill in values for the new columns:

```
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;
```

Insert the results from a query into the new table and change a value in the row (`id` in this case):

```
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;
```

The new table would then look like:

```
SELECT * FROM new_table_four_columns;

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

Insert Data from a Text File into a Table

You may want to load data from a *text file* (data stored in plain text without special formatting) into a table. A common type of text file is a *.csv* file (comma separated values). Text files can be opened up in applications outside of an RDBMS including Excel, Notepad, TextEdit, etc.

The following code shows how to load the *my_data.csv* file into a table.

Contents of the *my_data.csv* file:

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

Create a table:

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

The code in Table 5-9 loads the *my_data.csv* file into the `new_table` table for each RDBMS. When loading data, you can specify additional details about the data, such as:

- The data is separated by commas (`,`)

- Text values are enclosed in double quotes (`""`)
- Each new row is on a new line (`\n`)
- The first row of the text file (which contains the header) should be ignored

Table 5-9. Code to insert data from a .csv file

RDBMS	Code
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	While this can be done at the command line using `sqlldr`, the better approach is to load data through a graphical user interface like SQL*Loader or SQL Developer instead.
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`

After inserting the data, the table would look like this:

```
SELECT * FROM new_table;

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

Example Filepath to Desktop

If *my_data.csv* is on your Desktop, this is what the file path would look like for each operating system:

- 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*

NOTE

If *MySQL* gives you an error that says that loading local data is disabled, you can enable it by updating the global variable `local_infile`, quitting and restarting MySQL:

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

Missing Data and NULL Values

Each RDBMS interprets missing data from a *.csv* file in a different way. When the following line in a *.csv* file:

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

is inserted into a SQL table, the missing value between `8` and `Olivia` would get replaced with:

- A `NULL` value in *PostgreSQL* and *SQL Server*
- An empty string (`''`) in *MySQL* and *SQLite*

In *MySQL* and *SQLite*, you can use `\N` in a *.csv* file to represent a `NULL` value in a SQL table. When the following line in a *.csv* file,

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

is inserted into a *MySQL* table, the `\N` would get replaced with a `NULL` value in the table.

When it is inserted into a *SQLite* table, the `\N` would be hardcoded into the table. You could then run the code,

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

to replace the `\N` placeholders with `NULL` values in the table.

Modifying Tables

This section covers how to change the table name, columns, constraints, and data in a table.

NOTE

You need ALTER privileges to modify a table. If you get an error when running the code in this section, you do not have the permission to do so and need to talk to your database administrator.

Rename a Table or Column

After you've created a table, you can still rename the table and the columns of the table.

WARNING

If you modify a table, the table will be permanently changed. *There is no undo*, unless there has been a backup created. Double-check your statements before executing them.

Rename a table

The code in Table 5-10 shows how to rename a table in each RDBMS.

Table 5-10. Code to rename a table

RDBMS	Code
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';`

Rename a column

The code in Table 5-11 shows how to rename a column in each RDBMS.

Table 5-11. Code to rename a column

RDBMS	Code
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';`

Display, Add, and Delete Columns

After you've created a table, you can view, add, and delete columns from the table.

Display the columns of a table

The code in Table 5-12 shows how to display the columns of a table in each RDBMS.

Table 5-12. Code to display the columns of a table

RDBMS	Code
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);`

Add a column to a table

The code in Table 5-13 shows how to add a column to a table in each RDBMS.

Table 5-13. Code to add a column to a table

RDBMS	Code

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);`

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

Delete a column from a table

The code in Table 5-14 shows how to delete a column from a table in each RDBMS.

NOTE

If a column has any constraints, you must first delete the constraints before deleting the column.

Table 5-14. Code to delete a column from a table

RDBMS	Code
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	Refer to the manual modifications steps for SQLite

Manual Modifications in SQLite

SQLite does not support some table modifications, such as deleting columns or adding/modifying/deleting constraints.

As a workaround, you can either use a graphical user interface to generate code to modify a table, or you can manually create a new table and copy over data (see following steps).

1. Create a new table with the columns and constraints you want.

    ```
    CREATE TABLE my_table_2 (
       id INTEGER NOT NULL,
       country VARCHAR(2),
       name VARCHAR(30)
    );
    ```

2. Copy data from the old table to the new table.

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

3. Confirm that the data is in the new table.

    ```
    SELECT * FROM my_table_2;
    ```

4. Delete the old table.

    ```
    DROP TABLE my_table;
    ```

5. Rename the new table.

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

Display, Add, and Delete Rows

After you've created a table, you can view, add, and delete rows from the table.

Display rows of a table

To display the rows of a table, simply write a SELECT statement:

```
SELECT * FROM my_table;
```

Add rows to a table

Use INSERT INTO to add rows of data to a table:

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

Delete rows from a table

Use DELETE FROM to delete rows of data from a table:

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

Omit the WHERE clause to remove all rows from a table:

```
DELETE FROM my_table;
```

Deleting rows from a table is also known as *truncating*, which removes all of the data in a table without changing the table definition. So while the column names and constraints of the table still exist, it is now empty.

To get rid of a table completely, you can drop the table.

Display, Add, Modify, and Delete Constraints

A *constraint* is a rule that specifies what data can be inserted into a table. More on the various types of constraints can be found earlier in this chapter in the Create a Table with Constraints section.

Display the constraints of a table

The code in Table 5-15 shows how to display the constraints of a table in each RDBMS.

Table 5-15. Code to display the constraints of a table

RDBMS	Code
MySQL	`SHOW CREATE TABLE my_table;`

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

PostgreSQL	`\d my_table`

SQL Server	`-- List constraints (except default ones)`
	`SELECT table_name,`
	`       constraint_name,`
	`       constraint_type`
	`FROM information_schema.table_constraints`
	`WHERE table_name = 'my_table';`
	
	`-- List all default constraints`
	`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`

NOTE

Oracle stores table names and column names in all caps, unless you surround the column name with double quotes. When referring to a table name or a column name in a SQL statement, you must write the name in all caps (`MY_TABLE`).

Add a constraint

Let's start with the following `CREATE TABLE` statement:

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

The code in Table 5-16 adds a constraint that makes sure that the `lower_name` column is a lowercase version of the `name` column in each RDBMS.

Table 5-16. Code to add a constraint

RDBMS	Code
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	Refer to the manual modifications steps for SQLite

Modify a constraint

Let's start with the following `CREATE TABLE` statement:

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

The code in Table 5-17 modifies the following constraints:

- Changes the `country` column from defaulting to `CA` to defaulting to `NULL`
- Changes the `name` column from allowing 15 characters to allowing 30 characters

Table 5-17. Code to modify constraints in a table

RDBMS	Code

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

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 country`
	`      VARCHAR(2) NULL;`
	
	`ALTER TABLE my_table`
	`      ALTER COLUMN name`
	`      VARCHAR(30) NULL;`

SQLite	Refer to the manual modifications steps for SQLite

Delete a constraint

The code in Table 5-18 shows how to delete a constraint from a table in each RDBMS.

Table 5-18. Code to delete a constraint from a table

RDBMS	Code
MySQL	`ALTER TABLE my_table DROP CHECK chk_lower_name;`
Oracle, PostgreSQL, SQL Server	`ALTER TABLE my_table DROP CONSTRAINT chk_lower_name;`
SQLite	Refer to the manual modifications steps for SQLite

NOTE

In *MySQL*, `CHECK` can be replaced with `DEFAULT`, `INDEX` (for `UNIQUE` constraints), `PRIMARY KEY`, and `FOREIGN KEY`. To delete a `NOT NULL` constraint, you would `MODIFY` the constraint instead.

Update a Column of Data

Use `UPDATE .. SET ..` to update the values in a column of data.

Here is a table:

```
SELECT *
FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       Celine         5
  3 CA       Michael        4
  4 US       Stefani        9
```

Preview the change you'd like to make:

```
SELECT LOWER(name)
FROM my_table;

LOWER(name)
```

```
------------
celine
michael
stefani
```

Update the values in a column of data:

```
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
```

Update Rows of Data

Use `UPDATE .. SET .. WHERE ..` to update values in a row or multiple rows of data.

Here is a table:

```
SELECT *
FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       Celine         5
  3 CA       Michael        4
  4 US       Stefani        9
```

Preview the change you'd like to make:

```
SELECT awards + 1
FROM my_table
WHERE country = 'CA';

awards + 1
```

```
-----------
          6
          5
```

Update the values in multiple rows of data:

```
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
```

WARNING

It is very important to include a WHERE clause along with the SET clause when you are updating specific rows of data. Without the WHERE clause, the entire table would be updated.

Update Rows of Data with the Results of a Query

Instead of manually typing values to update a table, you can set a new value based on the results of a query.

Here is a table:

```
SELECT * FROM my_table;

id  country  name     awards
--- -------- -------- -------
  2 CA       Celine         5
  3 CA       Michael        4
  4 US       Stefani        9
```

Preview the change you'd like to make:

```
SELECT MIN(awards) FROM my_table;

MIN(awards)
-----------
          4
```

Update values based on a query:

```
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
```

NOTE

MySQL does not allow you to update a table with a query on the same table. In the preceding example, you cannot have `UPDATE my_table` and `FROM my_table`. The statement will run if you query `FROM another_table`.

The results of the query must always return one column and either zero or one row. If zero rows are returned, then the value is set to `NULL`.

Delete a Table

When you no longer need a table, you can delete it using a `DROP TABLE` statement:

```
DROP TABLE my_table;
```

In *MySQL*, *PostgreSQL*, *SQL Server*, and *SQLite*, you can also add `IF EXISTS` to avoid an error message if the table doesn't exist:

```
DROP TABLE IF EXISTS my_table;
```

WARNING

If you drop a table, you will lose all of the data in the table. *There is no undo*, unless there has been a backup created. I recommend not running this command unless you are 100% sure you don't need the table.

Delete a table with foreign key references

If other tables have foreign keys that reference the table you are dropping, you will need to delete the foreign key constraints in the other tables along with the table you are dropping.

The code in Table 5-19 shows how to delete a table with foreign key references in each RDBMS.

Table 5-19. Code to delete a table with foreign key references

RDBMS	Code
Oracle	`DROP TABLE my_table CASCADE CONSTRAINTS;`
PostgreSQL	`DROP TABLE my_table CASCADE;`
MySQL, SQL Server, SQLite	There is no `CASCADE` keyword, so you must manually delete any foreign key constraints that reference the table before dropping the table.

WARNING

It is dangerous to use `CASCADE` without knowing exactly what you are deleting. Please proceed with caution. I recommend not running this command unless you are 100% sure you don't need the constraints."

Indexes

Imagine you have a table with 10 million rows. You write a query on the table to return values that were logged on `2021-01-01`:

```
SELECT *
FROM my_table
WHERE log_date = '2021-01-01';
```

This query would take a long time to run. The reason is because behind the scenes, every single row is checked to see if the `log_date` matches `2021-01-01` or not. That is 10 million checks.

To speed this up, you could create an *index* on the `log_date` column. This is something you would do one time, and all future queries could benefit from it.

Book Index Versus SQL Index Comparison

To better understand how a SQL index works, it's helpful to use an analogy. Table 5-20 compares the index at the end of this book with an index in a SQL table.

Table 5-20. Book index versus SQL index comparison

	Book	SQL Table
Terms	A book has many *pages*. Each page has *attributes* including the word count, concepts covered, etc.	A table has many *rows*. Each row has *columns*, including `customer_id`, `log_date`, etc.
Scenario	You are reading this book and want to find all pages about the concept *subqueries*.	You are querying a table and want to find all rows where the `log_date` is `2021-01-01`.
The slow approach	You could start from page 1 and flip through every page of this book to see if *subqueries* are mentioned or not. This would take a long time.	You could start from row 1 and scan through every row to see whether the `log_date` is `2021-01-01` or not. This would take a long time.
Create an index	An index was created for all concepts in this book. Each concept is listed in the index along with the page numbers that talk about the concept.	An index was created on the `log_date` column in the table. Every `log_date` is listed in the index along with the row numbers that contain the `log_date`.
The fast approach	To find pages about *subqueries*, you can go to the index to quickly find the page numbers that reference *subqueries* and go to those pages.	To find rows with a `log_date` of `2021-01-01`, your query uses the index to quickly find the row numbers that contain the date and return those rows.

When the same query is run on `my_table` (that now has the `log_date` column indexed):

```
SELECT *
FROM my_table
WHERE log_date = '2021-01-01';
```

the query will run much faster because instead of checking each row in the table, it sees the `log_date` of `2021-01-01`, goes to the index, and quickly pulls all rows that have that `log_date`.

TIP

It’s a good idea to create an index on a few columns that you filter on often. For example, the primary key column, the date column, etc.

You wouldn’t want to create an index for too many columns, though, because it does take up space. Also, any time rows are added or removed, the index would need to be rebuilt, which is time consuming.

Create an Index to Speed Up Queries

The following code creates a new index called `my_index` on the `log_date` column in the `my_table` table:

```
CREATE INDEX my_index ON my_table (log_date);
```

NOTE

When creating an index in *Oracle*, you must uppercase the column name and surround it in quotes:

```
CREATE INDEX my_index ON my_table
  ('LOG_DATE');
```

Oracle automatically creates an index for `PRIMARY KEY` and `UNIQUE` columns when a table is created.

Indexes can take a long time to create. However, it’s a one-time task that’s worth it in the long run for many faster queries in the future.

You can also create a multicolumn index or a *composite index*. The following code creates an index on two columns: `log_date` and `team`:

```
CREATE INDEX my_index ON my_table (log_date, team);
```

The order of the columns matters here. If you write a query that filters on:

- Both columns: the index will make the query fast
- The first column (`log_date`): the index will make the query fast
- The second column (`team`): the index will not help because it first organizes data by the `log_date` and then the `team` column

---

NOTE

You need CREATE privileges to create an index. If you get an error when running the preceding code, you do not have the permission to do so and need to talk to your database administrator.

Delete an index

The code in Table 5-21 shows how to delete an index in each RDBMS.

Table 5-21. Code to delete an index

RDBMS	Code
MySQL, SQL Server	`DROP INDEX my_index ON my_table;`
Oracle, PostgreSQL, SQLite	`DROP INDEX my_index;`

WARNING

Dropping an index cannot be undone. Be 100% sure you want to delete an index before dropping it.

On the bright side, there is no data loss. The data in the table is untouched, and you can always recreate the index.

Views

Imagine you have a long and complex SQL query that includes many joins, filters, aggregations, etc. The results of the query are useful to you and something that you want to reference again at a later point.

This is a great situation to create a *view*, or give a name to the output of a query. Remember that the output of a query is a single table, so a view looks just like a table. The difference is that the view doesn't actually hold any data like a table, but just references the data instead.

NOTE

Sometimes database adminstrators (DBAs) will create views to restrict access to tables. Imagine there is a `customer` table. Most people should only be able to read the data in the table, and not make changes to it.

The DBA can create a `customer` view that includes data identical to the `customer` table. Now, everyone can query the `customer` *view*, and only the DBA would be able to edit the data within the `customer` *table*.

The following code is a complex query that we don't want to write over and over again:

```
-- Number of waterfalls owned by each owner
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
```

Let's say that we want to find the average number of waterfalls that an owner owns. We could do this using either a subquery or a view:

```
-- Subquery Approach
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


-- View Approach
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
```

NOTE

You need CREATE privileges to create a view. If you get an error when running the preceding code, you do not have the permission to do so and need to talk to your database administrator.

Subqueries Versus Views

Both subqueries and views represent the results of a query, which can then go on to be queried themselves.

- A *subquery* is temporary. It only exists for the duration of the query and is great for one-time use.
- A *view* is saved. Once a view is created, you can continue to write queries that reference the view.

Create a View to Save the Results of a Query

Use `CREATE VIEW` to save the results of a query as a view. The view can then be queried, just like a table.

Using this query:

```
SELECT *
FROM my_table
WHERE country = 'US';

id  country  name
--- -------- ------
  1 US       Anna
  2 US       Emily
  3 US       Molly
```

Create a view:

```
CREATE VIEW my_view AS
SELECT *
FROM my_table
WHERE country = 'US';
```

Query the view:

```
SELECT * FROM my_view;

id  country  name
--- -------- ------
  1 US       Anna
  2 US       Emily
  3 US       Molly
```

Display existing views

The code in Table 5-22 shows how to display all existing views in each RDBMS.

Table 5-22. Code to display existing views

RDBMS	Code

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';`

Update a view

To update a view is another way of saying to overwrite a view. The code in Table 5-23 shows how to update a view in each RDBMS.

Table 5-23. Code to update a view

RDBMS	Code

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';`

Delete a view

When you no longer need a view, you can delete it using a `DROP VIEW` statement:

```
DROP VIEW my_view;
```

WARNING

Dropping a view cannot be undone. Be 100% sure you want to delete a view before dropping it.

On the bright side, there is no data loss. The data is still in the original table, and you can always recreate the view.

Transaction Management

A *transaction* allows you to more safely update a database. It consists of a sequence of operations that are executed as a single unit. Either all of the operations are executed or none of them are, which is also known as *atomicity*.

The following code kicks off a transaction before making any changes to the tables. After the statements are run, no updates are permanently made to the database until the changes are committed:

```
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;
```

Why is it safer to use a transaction?

After starting a transaction:

All four statements are treated as one unit.
Imagine you run the first three statements, and while you're doing that, someone else edits the database in a way that your fourth statement doesn't run. This is problematic because for you to update the database properly, all four statements need to run together. The transaction does just that—it requires all four statements to act as one unit, so either all of them run or none of them run.

You can undo your changes if needed.
After starting the transaction, you can run each of the statements and see how they would affect the tables. If everything looks right, you can end the transaction and

lock in your changes with a `COMMIT`. If something looks wrong and you want to return things back to the way they were before the transaction, you can do so with a `ROLL BACK`.

In general, if you are updating a database, it is good practice to use a transaction.

The following sections cover two scenarios in which using a transaction is helpful—one ending in a `COMMIT` to confirm changes and the other ending in a `ROLLBACK` to undo changes.

Double-Check Changes Before a COMMIT

Imagine you want to delete some rows of data, but you want to double-check that the correct rows are going to get deleted before you permanently remove them from the table.

The following code shows the step-by-step process for how you would use a transaction in SQL to do so.

1. Start a transaction.

   ```
   -- MySQL and PostgreSQL
   START TRANSACTION;
   or
   BEGIN;

   -- SQL Server and SQLite
   BEGIN TRANSACTION;
   ```

   In *Oracle*, you are essentially always in a transaction. A transaction begins when you execute your first SQL statement. After a transaction has ended (with a `COMMIT` or `ROLL BACK`), another one begins when the next SQL statement is executed.

2. View the table you plan to change.

   You are in transaction mode at this point, meaning no changes will be made to the database.

```
SELECT * FROM books;

+------+-------------+
| id   | title       |
+------+-------------+
|    1 | Becoming    |
|    2 | Born a Crime |
|    3 | Bossypants  |
+------+-------------+
```

3. Test the change and see how it affects the table.

   You want to delete all multiword book titles. The following `SELECT` statement lets you view all the multiword book titles in the table.

   ```
   SELECT * FROM books WHERE title LIKE '% %';

   +------+-------------+
   | id   | title       |
   +------+-------------+
   |    2 | Born a Crime |
   +------+-------------+
   ```

   The following `DELETE` statement uses the same `WHERE` clause to now delete the multiword book titles in the table.

   ```
   DELETE FROM books WHERE title LIKE '% %';

   SELECT * FROM books;

   +------+-------------+
   | id   | title       |
   +------+-------------+
   |    1 | Becoming    |
   |    3 | Bossypants  |
   +------+-------------+
   ```

   You are still in transaction mode at this point, so the change has not been made permanent.

4. Confirm the change with `COMMIT`.

   Use `COMMIT` to lock in the changes. After this step, you are no longer in transaction mode.

   ```
   COMMIT;
   ```

WARNING

You cannot undo (aka rollback) a transaction once it has been committed.

Undo Changes with a ROLLBACK

Transactions are especially useful to test out changes and undo them if necessary.

1. Start a transaction.

   ```
   -- MySQL and PostgreSQL
   START TRANSACTION;
   or
   BEGIN;

   -- SQL Server and SQLite
   BEGIN TRANSACTION;
   ```

   In *Oracle*, you are essentially always in a transaction. A transaction begins when you execute your first SQL statement. After a transaction has ended (with a `COMMIT` or `ROLLBACK`), another one begins when the next SQL statement is executed.

2. View the table you plan to change.

   You are in transaction mode at this point, meaning no changes will be made to the database.

   ```
   SELECT * FROM books;

   +------+-------------+
   | id   | title       |
   +------+-------------+
   |    1 | Becoming    |
   |    2 | Born a Crime |
   |    3 | Bossypants  |
   +------+-------------+
   ```

3. Test the change and see how it affects the table.

You want to delete all multiword book titles. The following DELETE statement accidentally deletes the entire table (you've forgotten a space in '%%'). You didn't want this to happen!

```
DELETE FROM books WHERE title LIKE '%%';

SELECT * FROM books;

+------+-------------+
| id   | title       |
+------+-------------+
```

It's a good thing you're still in transaction mode at this point, so the change has not been made permanent.

4. Undo the change with ROLLBACK.

   Instead of COMMIT, ROLLBACK the changes. The table will not be deleted. After this step, you are no longer in transaction mode and can continue on with your other statements.

   ```
   ROLLBACK;
   ```

CHAPTER 6

Data Types

In a SQL table, each column can only include values of a single data type. This chapter covers commonly used data types, as well as how and when to use them.

The following statement specifies three columns along with the data type for each column: `id` holds integer values, `name` holds values containing up to 30 characters, and `dt` holds date values:

```
CREATE TABLE my_table (
   id INT,
   name VARCHAR(30),
   dt DATE
);
```

`INT`, `VARCHAR`, and `DATE` are just three of the many data types in SQL. Table 6-1 lists four categories of data types, along with common subcategories. Data type syntax varies widely by RDBMS, and the differences are detailed out in each section of this chapter.

Table 6-1. Data types in SQL

Numeric	String	Datetime	Other
Integer (`123`) Decimal (`1.23`) Floating Point (`1.23e10`)	Character (`'hello'`) Unicode (`'西瓜'`)	Date (`'2021-12-01'`) Time (`'2:21:00'`) Datetime (`'2021-12-01 2:21:00'`)	Boolean (`TRUE`) Binary (images, documents, etc.)

Table 6-2 lists example values of each data type to show how they are represented in SQL. These values are often referred to as *literals* or *constants*.

Table 6-2. Literals in SQL

Category	Subcategory	Example Values

Numeric	Integer	`123`
		`+123`
		`-123`

	Decimal	`123.45`
		`+123.45`
		`-123.45`

	Floating Point	`123.45E+23`
		`123.45e-23`

String	Character	`'Thank you!'`
		`'The combo is 39-6-27.'`

	Unicode	`N'Amélie'` `N'♥♥♥'`
Datetime	Date	`'2022-10-15'` `'15-OCT-2022'` *(Oracle)*

	Time	`'10:30:00'`
		`'10:30:00.123456'`
		`'10:30:00 -6:00'`

	Datetime	`'2022-10-15 10:30:00'` `'15-OCT-2022 10:30:00'` *(Oracle)*

Category	Subcategory	Example Values
Other	Boolean	`TRUE` `FALSE`
	Binary (example values are displayed as hexadecimal)	`X'AB12'` *(MySQL, PostgreSQL)* `x'AB12'` *(MySQL, PostgreSQL)* `0xAB12` *(MySQL, SQL Server, SQLite)*

The NULL Literal

Cells with no value are represented by the `NULL` keyword (aka the `NULL` literal), which is case insensitive (`NULL` = `Null` = `null`).

You will often see null values in a table, but null itself is not a data type. Any numeric, string, datetime, or other column can include null values within the column.

How to Choose a Data Type

When deciding on a data type for a column, it is important to balance storage size and flexibility.

Table 6-3 shows a few examples of integer data types. Note that each data type allows for a different range of values and requires a different amount of storage space.

Table 6-3. A sample of integer data types

Data Type	Range of Values Allowed	Storage Size
`INT`	-2,147,483,648 to 2,147,483,647	4 bytes
`SMALLINT`	-32,768 to 32,767	2 bytes
`TINYINT`	0 to 255	1 byte

Imagine you have a column of data that contains the number of students in a classroom:

```
15
25
50
70
100
```

This column contains numeric data—more specifically, integers. You could choose any of the three integer data types in Table 6-3 to assign to this column.

The case for `INT`
: If storage space isn't an issue, then `INT` is a simple and solid choice that works across all RDBMSs.

The case for `TINYINT`
: Since all values are between 0 and 255, choosing `TINYINT` would save on storage space.

The case for `SMALLINT`
: If there may be higher student counts inserted into the column at a later point, `SMALLINT` allows for more flexibility while still using less space than `INT`.

There is no single right answer here. The best data type for a column depends on both the storage space and flexibility required.

TIP

If you've already created a table but want to change the data type for a column, you can do so by modifying the column's constraint with an `ALTER TABLE` statement. More details can be found under Modifying a Constraint in Chapter 5.

Numeric Data

This section introduces numeric values to give you an idea of how they are represented in SQL, and then goes into detail on integer, decimal, and floating point data types.

Columns with numeric data can be input into numeric functions such as `SUM()` and `ROUND()`, which are covered in the Numeric Functions section in Chapter 7.

Numeric Values

Numeric values include integers, decimal numbers, and floating point numbers.

Integer values

Numbers without a decimal are treated as integers. The + is optional.

```
123    +123    -123
```

Decimal values

Decimals include a decimal point and are stored as exact values. The + is optional.

```
123.45    +123.45    -123.45
```

Floating point values

Floating point values use scientific notation.

```
123.45E+23    123.45e−23
```

These values are interpreted as 123.45×10^{23} and 123.45×10^{-23}, respectively.

NOTE

Oracle allows for a trailing F, f, D, or d to indicate FLOAT or DOUBLE (more precise FLOAT value):

```
123F   +123f   -123.45D   123.45d
```

Integer Data Types

The following code creates an integer column:

```
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 |
+-------------------+
```

Table 6-4 lists the integer data type options for each RDBMS.

Table 6-4. Integer data types

RDBMS	Data Type	Range of Values Allowed	Storage Size
MySQL	`TINYINT`	–128 to 127 0 to 255 (unsigned)	1 byte
	`SMALLINT`	–32,768 to 32,767 0 to 65,535 (unsigned)	2 bytes
	`MEDIUMINT`	–8,388,608 to 8,388,607 0 to 16,777,215 (unsigned)	3 bytes
	`INT` or `INTEGER`	–2,147,483,648 to 2,147,483,647 0 to 4,294,967,295 (unsigned)	4 bytes
	`BIGINT`	-2^{63} to $2^{63} - 1$ 0 to $2^{64} - 1$ (unsigned)	8 bytes
Oracle	`NUMBER`	-10^{125} to $10^{125} - 1$	1 to 22 bytes
PostgreSQL	`SMALLINT`	–32,768 to 32,767	2 bytes
	`INT` or `INTEGER`	–2,147,483,648 to 2,147,483,647	4 bytes
	`BIGINT`	-2^{63} to $2^{63} - 1$	8 bytes
SQL Server	`TINYINT`	0 to 255	1 byte
	`SMALLINT`	–32,768 to 32,767	2 bytes
	`INT` or `INTEGER`	–2,147,483,648 to 2,147,483,647	4 bytes
	`BIGINT`	-2^{63} to $2^{63} - 1$	8 bytes
SQLite	`INTEGER`	-2^{63} to $2^{63} - 1$ (if larger, will switch to a `REAL` data type)	1, 2, 3, 4, 6, or 8 bytes

NOTE

MySQL allows for both signed ranges (positive and negative integers) and unsigned ranges (positive integers only). The default is the signed range. To specify an unsigned range:

```
CREATE TABLE my_table (
   my_integer_column INT UNSIGNED
);
```

PostgreSQL has a `SERIAL` data type that creates an auto-incrementing integer (1, 2, 3, etc.) in a column. Table 6-5 lists the `SERIAL` options, each with a different range.

Table 6-5. Serial options in PostgreSQL

Data Type	Range of Values Generated	Storage Size
`SMALLSERIAL`	1 to 32,767	2 bytes
`SERIAL`	1 to 2,147,483,647	4 bytes
`BIGSERIAL`	1 to 9,223,372,036,854,775,807	8 bytes

Decimal Data Types

Decimal numbers are also known as *fixed point* numbers. They include a decimal point and are stored as an exact value. Monetary data (like `799.95`) is often stored as a decimal number.

The following code creates a decimal column:

```
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 |
+-------------------+
```

When defining the data type `DECIMAL(5,2)`:

- *5* is the maximum number of *total digits* that are stored. This is called the *precision*.
- *2* is the number of digits to the *right of the decimal point*. This is called the *scale*.

Table 6-6 lists the decimal data type options for each RDBMS.

Table 6-6. Decimal data types

RDBMS	Data Type	Max Digits Allowed	Default
MySQL	`DECIMAL` or `NUMERIC`	Total: 65 After decimal point: 30	`DECIMAL(10,0)`
Oracle	`NUMBER`	Total: 38 After decimal point: –84 to 127 (negative means before the decimal point)	0 digits after decimal point
PostgreSQL	`DECIMAL` or `NUMERIC`	Before decimal point: 131,072 After decimal point: 16,383	`DECIMAL(30,6)`
SQL Server	`DECIMAL` or `NUMERIC`	Total: 38 After decimal point: 38	`DECIMAL(18,0)`
SQLite	`NUMERIC`	No inputs	No default

Floating Point Data Types

Floating point numbers are a computer science concept. When a number has many digits, either before or after a decimal point, instead of storing all the digits, floating point numbers only store a limited number of them to save on space.

- Number: `1234.56789`
- Floating point notation: `1.23 x 10^3`

You'll notice that the decimal point "floated" over a few spaces to the left and that an *approximate* value (`1.23`) was stored, instead of the full original value (`1234.56789`).

There are two floating point data types:

- *Single precision*: number is represented by at least 6 digits, with a full range of around 1E–38 to 1E+38
- *Double precision*: number is represented by at least 15 digits, with a full range of around 1E–308 to 1E+308 The following code creates both a single precision (`FLOAT`) and a double precision (`DOUBLE`) floating point column:

```
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 |
+-----------------+-------------------+
```

WARNING

Because floating point data stores approximate values, comparisons and calculations may be slightly off from what you would expect.

If your data will always have the same number of decimal digits, it is better to use a fixed point data type like `DECIMAL` to store exact values instead of a floating point data type.

Table 6-7 lists the floating point data type options for each RDBMS.

Table 6-7. Floating point data types

RDBMS	Data Type	Input Range	Storage Size
MySQL	`FLOAT`	0 to 23 bits	4 bytes
	`FLOAT`	24 to 53 bits	8 bytes
	`DOUBLE`	0 to 53 bits	8 bytes
Oracle	`BINARY_FLOAT`	No inputs	4 bytes
	`BINARY_DOUBLE`	No inputs	8 bytes
PostgreSQL	`REAL`	No inputs	4 bytes
	`DOUBLE PRECISION`	No inputs	8 bytes
SQL Server	`REAL`	No inputs	4 bytes
	`FLOAT`	1 to 24 bits	4 bytes
	`FLOAT`	25 to 53 bits	8 bytes
SQLite	`REAL`	No inputs	8 bytes

NOTE

Oracle's `FLOAT` data type is NOT a floating point number. Instead, `FLOAT` is equivalent to `NUMERIC`, which is a decimal number. For a floating point data type, you should use `BINARY_FLOAT` or `BINARY_DOUBLE` instead.

Bits versus Bytes versus Digits

1 *bit* is the smallest unit of storage. It can have a value of `0` or `1`.

1 *byte* consists of 8 bits. Example byte: `10101010`.

Each character is represented by a byte. The *digit* `7` = `00000111` in byte form.

String Data

This section introduces string values to give you an idea of how they are represented in SQL, and then goes into detail on character and unicode data types.

Columns with string data can be input into string functions such as `LENGTH()` and `REGEXP()` (regular expression), which are covered in the String Functions section in Chapter 7.

String Values

String values are sequences of characters including letters, numbers, and special characters.

String basics

The standard is to enclose string values in single quotes:

```
'This is a string.'
```

Use two adjacent single quotes when you need to embed a single quote in a string:

```
'You''re welcome.'
```

SQL will treat the two adjacent single quotes as a single quote within the string and return:

```
'You're welcome.'
```

TIP

As a best practice, single quotes (`''`) should be used to enclose string values, while double quotes (`""`) should be used for identifiers (names of tables, columns, etc.).

Alternatives to single quotes

If your text contains many single quotes and you want to use a different character to denote a string, Oracle and PostgreSQL allow you to do so.

Oracle allows you to preface a string with a `Q` or `q`, followed by any character, then the string and finally the character again:

```
Q'[This is a string.]'
q'[This is a string.]'
Q'|This is a string.|'
```

PostgreSQL allows you to surround text with two dollar signs and an optional tag name:

```
$$This is a string.$$
$mytag$This is a string.$mytag$
```

Escape sequences

MySQL and PostgreSQL support *escape sequences*, or a special sequence of text that has meaning. Table 6-8 lists common escape sequences.

Table 6-8. Common escape sequences

Escape Sequence	Description
`\'`	Single quote
`\t`	Tab
`\n`	New line
`\r`	Carriage return
`\b`	Backspace
`\\`	Backslash

MySQL allows you to include escape sequences within a string using the \ character:

```
SELECT 'hello', 'he\'llo', '\thello';

+-------+--------+-----------+
| hello | he'llo |      hello |
+-------+--------+-----------+
```

PostgreSQL allows you to include escape sequences in strings if the overall string is prefaced with an `E` or `e`:

```
SELECT 'hello', E'he\'llo', e'\thello';

---------+---------+--------------
 hello   | he'llo   |         hello
```

Escape sequences only apply to strings enclosed by single quotes and not strings enclosed by dollar signs.

Character Data Types

The most common way to hold string values is to use character data types. The following code creates a variable character column allowing for up to 50 characters:

```
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! :)     |
+------------------------------+
```

There are three main character data types:

`VARCHAR` *(variable character)*
: This is the most popular string data type. If the data type is `VARCHAR(50)`, then the column will allow up to 50 characters. In other words, the string length is variable.

`CHAR` *(character)*
: If the data type is `CHAR(5)`, then each value in the column will have exactly 5 characters. In other words, the string length is fixed. Data will be right-padded with spaces to be exactly the length specified. For example, `'hi'` would be stored as `'hi   '`.

`TEXT`
: Unlike `VARCHAR` and `CHAR`, `TEXT` requires no inputs, meaning you do not have to specify a length for the text. It is useful for storing long strings, like a paragraph or more of text.

Table 6-9 lists the character data type options for each RDBMS.

Table 6-9. Character data types

RDBMS	Data Type	Input Range	Default	Storage Size
MySQL	`CHAR`	0 to 255 characters	`CHAR(1)`	Varies
	`VARCHAR`	0 to 65,535 characters	Input required	Varies
	`TINY TEXT`	No inputs	No inputs	255 bytes
	`TEXT`	No inputs	No inputs	65,535 bytes
	`MEDIUM TEXT`	No inputs	No inputs	16,777,215 bytes
	`LARGE TEXT`	No inputs	No inputs	4,294,967,295 bytes
Oracle	`CHAR`	1 to 2,000 characters	`CHAR(1)`	Varies
	`VAR CHAR2`	1 to 4,000 characters	Input required	Varies
	`LONG`	No inputs	No inputs	2 GB
PostgreSQL	`CHAR`	1 to 10,485,760 characters	`CHAR(1)`	Varies
	`VARCHAR`	1 to 10,485,760 characters	Input required	Varies
	`TEXT`	No inputs	No inputs	Varies
SQL Server	`CHAR`	1 to 8,000 bytes	Input required	Varies
	`VARCHAR`	1 to 8,000 bytes, or `max`	Input required	Varies, or up to 2 GB
	`TEXT`	No inputs	No inputs	2,147,483,647 bytes
SQLite	`TEXT`	No inputs	No inputs	Varies

NOTE

Oracle's `VARCHAR2` is typically used instead of `VARCHAR`. They are identical in terms of functionality, but `VARCHAR` may one day be modified, so it's safer to use `VARCHAR2`.

Unicode Data Types

Character data types are typically stored as *ASCII* data, but can also be stored as *Unicode* data if a larger library of characters is needed.

ASCII Versus Unicode Encoding

There are many ways to *encode* data, or in other words, turn data into 0's and 1's for a computer to understand. The default encoding that SQL uses is called *ASCII (American Standard Code for Information Interchange)*.

With ASCII, there are 2^8 = 128 characters that are turned into a series of eight 0's and 1's. For example, the `!` character maps to `00100001`. These eight 0's and 1's are known as a *byte* of data.

There are other encoding types beyond ASCII, such as *UTF (Unicode Transformation Format)*. With Unicode, there are 2^{21} characters:

- The first 2^8 characters are the same as ASCII (`!` = `100001`).
- Other characters include Asian characters, math symbols, emojis, etc.
- Not all characters have been assigned values yet.

The following code shows the difference between the VARCHAR and NVARCHAR (Unicode) data types:

```
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       |
+------------+-------------+
```

NOTE

When inserting Unicode data from a text file into an NVARCHAR column, the Unicode values in the text file do not need the N prefix.

Table 6-10 lists the Unicode data type options for each RDBMS.

Table 6-10. Unicode data types

RDBMS	Data Type	Description
MySQL	NCHAR	Like CHAR, but for Unicode data
	NVARCHAR	Like VARCHAR, but for Unicode data
Oracle	NCHAR	Like CHAR, but for Unicode data
	NVARCHAR2	Like VARCHAR2, but for Unicode data

RDBMS	Data Type	Description
PostgreSQL	`CHAR`	`CHAR` supports Unicode data
	`VARCHAR`	`VARCHAR` supports Unicode data
SQL Server	`NCHAR`	Like `CHAR`, but for Unicode data
	`NVARCHAR`	Like `VARCHAR`, but for Unicode data
SQLite	`TEXT`	`TEXT` supports Unicode data

Datetime Data

This section introduces datetime values to give you an idea of how they are represented in SQL, and then goes into detail on the datetime data types in each RDBMS.

Columns with datetime data can be input into datetime functions such as `DATEDIFF()` and `EXTRACT()`, which are covered in the Datetime Functions section in Chapter 7.

Datetime Values

Datetime values can come in the form of dates, times or datetimes.

Date values

A date column should have date values in the format *YYYY-MM-DD*. In *Oracle*, the default format is *DD-MON-YYYY*.

October 15th, 2022 is written as:

```
'2022-10-15'
```

In *Oracle*, October 15th, 2022 is written as:

```
'15-OCT-2022'
```

When referencing a date value in a query, you must preface the string with either a `DATE` or `CAST` keyword to tell SQL it is a date, as shown in Table 6-11.

Table 6-11. Referencing a date in a query

RDBMS	Code

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');`

NOTE

In *Oracle*, the date format after the `DATE` keyword is different than the date format within the `CAST` function.

Also, in *Oracle*, when doing a calculation or looking up a system variable that only contains a `SELECT` clause, you need to add `FROM dual` to the end of the query. `dual` is a dummy table that holds a single value.

```
SELECT DATE '2021-02-25' FROM dual;
SELECT CURRENT_DATE FROM dual;
```

If a column contains dates of a different format, such as *MM/DD/YY*, you can apply a string to date function for SQL to recognize it as a date.

Time values

A time column should have time values in the format *hh:mm:ss*. 10:30 a.m. is written as:

```
'10:30:00'
```

You can also include more granular seconds, up to six decimal places:

```
'10:30:12.345678'
```

You can also add a time zone. Central Standard Time is also known as `UTC-06:00`:

```
'10:30:12.345678 -06:00'
```

When referencing a time value in a query, you must preface the string with either a `TIME` or `CAST` keyword to tell SQL it is a time, as shown in Table 6-12.

Table 6-12. Referencing a time in a query

RDBMS	Code

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');`

NOTE

In *Oracle*, the time format after the `TIME` keyword must include seconds as well.

If a column contains times of a different format, such as *mmss*, you can apply a string to time function for SQL to recognize it as a time.

Date and time values

A datetime column should have datetime values in the format *YYYY-MM-DD hh:mm:ss*. In *Oracle*, the default format is *DD-MON-YYYY hh:mm:ss*.

October 15, 2022 at 10:30 a.m. is written as:

```
'2022-10-15 10:30'
```

In *Oracle*, October 15, 2022 at 10:30 a.m. is written as:

```
'15-OCT-2022 10:30'
```

When referencing a datetime value in a query, you must preface the string with either a `DATETIME`, `TIMESTAMP`, or `CAST` keyword to tell SQL it is a datetime, as shown in Table 6-13.

Table 6-13. Referencing a datetime in a query

RDBMS	Code

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');`

NOTE

In *MySQL*, the keyword is `TIMESTAMP`, but the data type is `DATETIME` within the `CAST` function.

In *Oracle*, the date format after the `TIMESTAMP` keyword is different than the date format within the `CAST` function. Also, the time format after the `TIMESTAMP` keyword must include seconds, but it is not required within the `CAST` function.

If a column contains datetimes of a different format, such as *MM/DD/YY mm:ss*, you can apply a string to date or string to time function for SQL to recognize it as a datetime.

Datetime Data Types

There are many ways to store datetime values. Because the data types vary so widely, in this section, there is a separate subsection for each RDBMS.

MySQL datetime data types

The following code creates five different datetime columns:

```
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 |
+---------------------+------+
```

Table 6-14 lists common datetime data type options in MySQL.

Table 6-14. MySQL datetime data types

Data Type	Format	Range
DATE	YYYY-MM-DD	1000-01-01 to 9999-12-31
TIME	hh:mm:ss	–838:59:59 to 838:59:59
DATETIME	YYYY-MM-DD hh:mm:ss	1000-01-01 00:00:00 to 9999-12-31 23:59:59
TIMESTAMP	YYYY-MM-DD hh:mm:ss	1970-01-01 00:00:01 UTC to 2038-01-19 03:14:07 UTC
YEAR	YYYY	0000 to 9999

NOTE

Both `DATETIME` and `TIMESTAMP` store dates and times. The difference is that `DATETIME` doesn't have a time zone attached to it, whereas `TIMESTAMP` stores Unix values (a specific point in time) and is often used to note when a record is created or updated.

Oracle datetime data types

The following code creates four different datetime columns:

```
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
```

Table 6-15 lists common datetime data type options in Oracle.

Table 6-15. Oracle datetime data types

Data Type	Description
DATE	Can store either just the date or the date and time if the `NLS_DATE_FORMAT` is updated
TIMESTAMP	Like `DATE`, but adds fractional seconds (the default is six digits, but can go up to nine digits after the decimal point)
TIMESTAMP WITH TIME ZONE	Like `TIMESTAMP`, but adds the time zone
TIMESTAMP WITH LOCAL TIME ZONE	Like `TIMESTAMP WITH TIME ZONE`, but adjusts based on the user's local time zone

Check the datetime formats in Oracle

The following code checks the current date and timestamp formats:

```
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 AM
```

To change the date or timestamp format, you can alter the `NLS_DATE_FORMAT` or `NLS_TIMESTAMP_FORMAT` parameter.

The following code changes the current `NLS_DATE_FORMAT = DD-MON-RR` to include time as well:

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH:MI:SS';
```

Other common symbols for date and time, such as `YYYY` for year and `HH` for hour can be found in Table 7-27: Datetime format specifiers.

PostgreSQL datetime data types

The following code creates five different datetime columns:

```
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
```

Table 6-16 lists common datetime data type options in PostgreSQL.

Table 6-16. PostgreSQL datetime data types

Data Type	Format	Range
DATE	YYYY-MM-DD	4713 BC to 5874897 AD
TIME	hh:mm:ss	00:00:00 to 24:00:00
TIME WITH TIME ZONE	hh:mm:ss+tz	00:00:00+1459 to 24:00:00–1459
TIMESTAMP	YYYY-MM-DD hh:mm:ss	4713 BC to 294276 AD
TIMESTAMP WITH TIME ZONE	YYYY-MM-DD hh:mm:ss+tz	4713 BC to 294276 AD

SQL Server datetime data types

The following code creates six different datetime columns:

```
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
```

Table 6-17 lists common datetime data type options in SQL Server.

Table 6-17. SQL Server datetime data types

Data Type	Format	Range
DATE	YYYY-MM-DD	0001-01-01 to 9999-12-31
TIME	hh:mm:ss	00:00:00.0000000 to 23:59:59.9999999
SMALLDATETIME	YYYY-MM-DD hh:mm:ss	*Date*: 1900-01-01 to 2079-06-06 *Time*: 0:00:00 through 23:59:59
DATETIME	YYYY-MM-DD hh:mm:ss	*Date*: 1753-01-01 to 9999-12-31 *Time*: 00:00:00 to 23:59:59.999
DATETIME2	YYYY-MM-DD hh:mm:ss	*Date*: 0001-01-01 to 9999-12-31 *Time*: 00:00:00 to 23:59:59.9999999

Data Type	Format	Range
DATETIMEOFFSET	YYYY-MM-DD hh:mm:ss +hh:mm	Time zone offset ranges from –12:00 to +14:00

SQLite datetime data types

SQLite doesn't have a datetime data type. Instead, `TEXT`, `REAL`, or `INTEGER` can be used to store datetime values.

> **NOTE**
>
> Even though there aren't specific datetime data types in SQLite, datetime functions including `DATE()`, `TIME()`, and `DATETIME()` allow you to work with dates and times in SQLite.
>
> More details can be found in the Datetime Functions section in Chapter 7.

The following code shows three ways to store datetime values in 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
```

Table 6-18 lists the datetime data type options in SQLite.

Table 6-18. Table 6-18. SQLite datetime data types

Data Type	Description
`TEXT`	Stored as a string in the format *YYYY-MM-DD HH:MM:SS.SSS*
`REAL`	Stored as a Julian day number, which is the number of days since noon in Greenwich on November 24, 4714 BC
`INTEGER`	Stored as Unix time, which is the number of seconds since 1970-01-01 00:00:00 UTC

Other Data

There are many other data types in SQL, including ones that are specific to each RDBMS.

Some of them fall into one of the existing categories of data types, but capture more detailed data, like the numeric type `MONEY` or the datetime type `INTERVAL`.

Others capture more complex data, like geospatial data that notes a particular location on earth or web data stored in JSON/XML formats.

This section covers two additional data types: Boolean data and data from external files.

Boolean Data

The two Boolean values are `TRUE` and `FALSE`. They are case insensitive and should be written without quotes:

```
SELECT TRUE, True, FALSE, False;

+------+------+-------+-------+
|    1 |    1 |     0 |     0 |
+------+------+-------+-------+
```

Boolean data types

MySQL, *PostgreSQL*, and *SQLite* support Boolean data types. The following code creates a Boolean column:

```
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 |
+-------------------+
```

Oracle and *SQL Server* don't have Boolean data types, but there are workarounds:

- In *Oracle*, use the `CHAR(1)` data type to hold values `'T'` and `'F'` or the `NUMBER(1)` data type to hold values `1` and `0`.
- In *SQL Server*, use the `BIT` data type, which holds `1`, `0`, and `NULL` values.

External Files (Images, Documents, etc.)

If you plan to include images (.jpg, .png, etc.) or documents (.doc, .pdf, etc.) in a column of data, there are two approaches to do so: store links to the files (more common) or store the files as binary values.

Approach 1: Store links to the files

This is typically the recommended approach if your files are over 1 MB each. For reference, the average iPhone photo is a few MB.

The files would be stored outside of the database, putting less load on the database, and often resulting in better performance.

Steps to store links to files:

1. Note the path names of the files on the file system (`/Users/images/img_001.jpg`).
2. Create a column that stores strings, like `VARCHAR(100)`.
3. Insert the path names into the column.

Approach 2: Store the files as binary values

This is typically the recommended approach if your files are smaller in size.

The files would be stored inside of the database, which makes things like backing up the data more straightforward.

Steps to store binary values:

1. Convert the files to binary (if you open up a binary file, it will look like a random sequence of characters, such as `Z™/≈jhJcE Ät, ÷mfPfõrà`).
2. Create a column that stores binary values, like `BLOB`.
3. Insert the binary values into the column.

Binary and hexadecimal values

Binary data represents the raw values that a computer interprets. It is often displayed in a more compact, human-readable form called *hexidecimal*.

- Character: a
- Equivalent binary value: 01100001
- Equivalent hexidecimal value: 61

Hexadecimals convert 1's and 0's into a number system of 16 symbols (0-9 and A-F). Hexadecimals are proceeded by X, x, or 0x:

```
SELECT X'AF12', x'AF12', 0xAF12;

+---------+---------+--------+
| 0xAF12  | 0xAF12  | 0xAF12 |
+---------+---------+--------+
```

MySQL supports all three formats. *PostgreSQL* supports the first two formats. *SQL Server* and *SQLite* support the third format.

In *Oracle*, while you can't easily display a hexidecimal value, you can use the TO_NUMBER function to display a hexidecimal value as a number instead: SELECT TO_NUMBER('AF12', 'XXXX') FROM dual; with the X standing for hexidecimal notation.

Binary data types

The following code creates a binary data column:

```
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                   |
+------------------------------------+
```

In *MySQL*, *Oracle*, and *SQLite*, the most common binary data type is `BLOB`.

In *PostgreSQL*, use `bytea` instead.

In *SQL Server*, use `VARBINARY` (such as `VARBINARY(100)`) instead.

NOTE

In *Oracle* and *SQL Server*, the string `ae$ iou` isn't automatically recognized as a binary value and needs to first be converted into one before getting inserted into a table.

```
-- Oracle
SELECT RAWTOHEX('ae$ iou') FROM dual;

-- SQL Server
SELECT CONVERT(VARBINARY, 'ae$ iou');
```

Table 6-19 lists the binary data type options for each RDBMS.

Table 6-19. Binary data types

RDBMS	Data Type	Description	Input Range	Storage Size
MySQL	`BINARY`	Fixed length binary string where values are right-padded with 0's to get to the exact size	0 to 255 bytes	Varies
	`VARBINARY`	Variable length binary string	0 to 65,535 bytes	Varies
	`TINY BLOB`	Tiny Binary Large OBject	No inputs	255 bytes
	`BLOB`	Binary Large OBject	No inputs	65,535 bytes
	`MEDIUM BLOB`	Medium Binary Large OBject	No inputs	16,777,215 bytes
	`LARGE BLOB`	Large Binary Large OBject	No inputs	4,294,967,295 bytes
Oracle	`RAW`	Variable length binary string	1 to 32,767 bytes	Varies
	`LONG RAW`	Larger `RAW`	No inputs	2 GB
	`BLOB`	Larger `LONG RAW`	No inputs	4 GB
PostgreSQL	`BYTEA`	Variable length binary string	No inputs	1 or 4 bytes plus the actual binary string

RDBMS	Data Type	Description	Input Range	Storage Size
SQL Server	`BINARY`	Fixed length binary string where values are left padded with 0's to get to the exact size	1 to 8,000 bytes	Varies
	`VARBINARY`	Variable length binary string	1 to 8,000 bytes, or `max`	Varies, or up to 2 GB
SQLite	`BLOB`	Binary Large OBject	No inputs	Stored exactly as it was input

CHAPTER 7

Operators and Functions

Operators and *functions* are used to perform calculations, comparisons, and transformations within a SQL statement. This chapter provides code examples for commonly used operators and functions.

The following query highlights five operators (+, =, OR, BETWEEN, AND) and two functions (UPPER, YEAR):

```
-- Pay increases for employees
SELECT name, pay_rate + 5 AS new_pay_rate
FROM employees
WHERE UPPER(title) = 'ANALYST'
      OR YEAR(start_date) BETWEEN 2016 AND 2018;
```

Operators Versus Functions

Operators are symbols or keywords that perform a calculation or comparison. Operators can be found within the SELECT, ON, WHERE, and HAVING clauses of a query.

Functions take in zero or more inputs, apply a calculation or transformation, and output a value. Functions can be found within the SELECT, WHERE and HAVING clauses of a query.

In addition to SELECT statements, operators and functions can also be used in INSERT, UPDATE, and DELETE statements.

This chapter includes one section on Operators and five sections on functions: Aggregate Functions, Numeric Functions, String Functions, Date Time Functions, and Null Functions.

Table 7-1 lists common operators and Table 7-2 lists common functions.

Table 7-1. Common operators

Logical Operators	Comparison Operators (Symbols)	Comparison Operators (Keywords)	Math Operators
AND	=	BETWEEN	+
OR	!=, <>	EXISTS	-
NOT	<	IN	*
	<=	IS NULL	/
	>	LIKE	%
	v=		

Table 7-2. Common functions

Aggregate Functions	Numeric Functions	String Functions	Datetime Functions	Null Functions
COUNT()	ABS()	LENGTH()	CURRENT_DATE	COALESCE()
SUM()	SQRT()	TRIM()	CURRENT_TIME	
AVG()	LOG()	CONCAT()	DATEDIFF()	
MIN()	ROUND()	SUBSTR()	EXTRACT()	
MAX()	CAST()	REGEXP()	CONVERT()	

Operators

Operators can be symbols or keywords. They can perform calculations (+) or comparisons (BETWEEN). This section describes the available operators in SQL.

Logical Operators

Logical operators are used to modify conditions, which result in TRUE, FALSE, or NULL. The logical operators in the code block (NOT, AND, OR) are bolded:

```
SELECT *
FROM employees
WHERE start_date IS NOT NULL
      AND (title = 'analyst' OR pay_rate < 25);
```

TIP

When using AND and OR to combine multiple conditional statements, it's a good idea to clearly state the order of operations with parentheses: ().

Table 7-3 lists the logical operators in SQL.

Table 7-3. Logical operators

Operator	Description
AND	Returns TRUE if both conditions are TRUE. Returns FALSE if either is FALSE. Returns NULL otherwise.
OR	Returns TRUE if either condition is TRUE. Returns FALSE if both are FALSE. Returns NULL otherwise.
NOT	Returns TRUE if the condition is FALSE. Returns FALSE if it is TRUE. Returns NULL otherwise.

Imagine there is a column called name. Table 7-4 shows how values in the column would be evaluated in a conditional statement without a NOT and with a NOT.

Table 7-4. NOT example

name	name IN ('Henry', 'Harper')	name NOT IN ('Henry', 'Harper')
Henry	TRUE	FALSE
Lily	FALSE	TRUE
NULL	NULL	NULL

Imagine there are two columns called `name` and `age`. Table 7-5 shows how values in the columns would be evaluated in a conditional statement with an `AND` and with an `OR`.

Table 7-5. AND and OR example

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

Comparison Operators

Comparison operators are used in predicates.

Operators Versus Predicates

Predicates are comparisons that include an *operator*:

- The predicate `age = 35` includes the = operator.
- The predicate `COUNT(id) < 20` includes the < operator.

Predicates are also known as conditional statements. These comparisons are evaluated on each row in a table, and result in a value of TRUE, FALSE, or NULL.

The comparison operators in the code block (IS NULL, =, BETWEEN) are bolded:

```
SELECT *
FROM employees
WHERE start_date IS NOT NULL
      AND (title = 'analyst'
      OR pay_rate BETWEEN 15 AND 25);
```

Table 7-6 lists comparison operators that are symbols and Table 7-7 lists comparison operators that are keywords.

Table 7-6. Comparison operators (symbols)

Operator	Description
=	Tests for equality
!=, <>	Tests for inequality
<	Tests for less than
<=	Tests for less than or equal to
>	Tests for greater than
>=	Tests for greater than or equal to

NOTE

MySQL also allows for <=>, which is a null-safe test for equality.

When using =, if two values are compared and one of them is NULL, the resulting value is NULL.

When using <=>, if two values are compared and one of them is NULL, the resulting value is 0. If they are both NULL, the resulting value is 1.

Table 7-7. Comparison operators (keywords)

Operator	Description
`BETWEEN`	Tests whether a value lies within a given range
`EXISTS`	Tests whether rows exist in a subquery
`IN`	Tests whether a value is contained in a list of values
`IS NULL`	Tests whether a value is null or not
`LIKE`	Tests whether a value matches a simple pattern

NOTE

The `LIKE` operator is used to match simple patterns, like finding text that starts with the letter `A`. More details can be found in the `LIKE` section.

Regular expressions are used to match more complex patterns, like extracting any text located between two punctuation marks. More details can be found in the regular expressions section.

Each keyword comparison operator is explained in detail in the following sections.

BETWEEN

Use `BETWEEN` to test if a value falls within a range. `BETWEEN` is a combination of `>=` and `<=`. The smaller of the two values should always be written first, with the `AND` operator separating the two.

To find all rows where the ages are greater than or equal to 35 and less than or equal to 44:

```
SELECT *
FROM my_table
WHERE age BETWEEN 35 AND 44;
```

To find all rows where the ages are less than 35 or greater than 44:

```
SELECT *
FROM my_table
WHERE age NOT BETWEEN 35 AND 44;
```

EXISTS

Use EXISTS to test if a subquery returns results or not. Typically, the subquery references another table.

The following query returns employees who also happen to be customers:

```
SELECT e.id, e.name
FROM employees e
WHERE EXISTS (SELECT *
              FROM customers c
              WHERE c.email = e.email);
```

EXISTS Versus JOIN

The EXISTS query could also be written with a JOIN:

```
SELECT *
FROM employees e INNER JOIN customers c
     ON e.email = c.email;
```

A JOIN is preferred when you want values from both tables to be returned (SELECT *).

An EXISTS is preferred when you want values from a single table to be returned (SELECT e.id, e.name). This type of query is sometimes referred to as a *semi-join*. EXISTS is also useful when the second table has duplicate rows and you're only interested in whether a row exists or not.

The following query returns customers who have never made a purchase:

```
SELECT c.id, c.name
FROM customers c
WHERE NOT EXISTS (SELECT *
                  FROM orders o
                  WHERE o.email = c.email);
```

IN

Use IN to test whether a value falls within a list of values.

The following query returns values for a few employees:

```
SELECT *
FROM employees
WHERE e.id IN (10001, 10032, 10057);
```

The following query returns employees who have not taken a vacation day:

```
SELECT e.id
FROM employees e
WHERE e.id NOT IN (SELECT v.emp_id
                   FROM vacations v);
```

WARNING

When using NOT IN, if there is even a single NULL value in the column in the subquery (v.emp_id in this case), the subquery will never be TRUE, meaning no rows will be returned.

If there are potentially NULL values in the column in the subquery, it is better to use NOT EXISTS:

```
SELECT e.id
FROM employees e
WHERE NOT EXISTS (SELECT *
                  FROM vacations v
                  WHERE v.emp_id = e.id);
```

IS NULL

Use IS NULL or IS NOT NULL to test whether a value is null or not.

The following query returns employees who don't have a manager:

```
SELECT *
FROM employees
WHERE manager IS NULL;
```

The following query returns employees who have a manager:

```
SELECT *
FROM employees
WHERE manager IS NOT NULL;
```

LIKE

Use LIKE to match a simple pattern. The percent sign (%) is a wildcard that means one or more characters.

Here is a sample table:

```
SELECT * FROM my_table;

+------+-------------------+
| id   | txt               |
+------+-------------------+
|    1 | You are great.    |
|    2 | Thank you!        |
|    3 | Thinking of you.  |
|    4 | I'm 100% positive. |
+------+-------------------+
```

Find all rows that *contain* the term you:

```
SELECT *
FROM my_table
WHERE txt LIKE '%you%';
```

```
-- MySQL, SQL Server and SQLite results
+------+-----------------+
| id   | txt             |
+------+-----------------+
|    1 | You are great.  |
|    2 | Thank you!      |
|    3 | Thinking of you. |
+------+-----------------+

-- Oracle and PostgreSQL results
+------+-----------------+
| id   | txt             |
+------+-----------------+
|    2 | Thank you!      |
|    3 | Thinking of you. |
+------+-----------------+
```

In *MySQL*, *SQL Server*, and *SQLite*, the pattern is case insensitive. Both `You` and `you` are captured by `'%you%'`.

In *Oracle* and *PostgreSQL*, the pattern is case sensitive. Only `you` is captured by `'%you%'`.

Find all rows that *start with* the term `You`:

```
SELECT *
FROM my_table
WHERE txt LIKE 'You%';

+------+----------------+
| id   | txt            |
+------+----------------+
|    1 | You are great. |
+------+----------------+
```

Use `NOT LIKE` to return rows that don't contain the characters.

Instead of the percent sign (`%`) to match one or more characters, you can use the underscore (`_`) to match exactly one character.

WARNING

Because `%` and `_` have special meaning when used with `LIKE`, if you want to search for those actual characters, you'll need to add the `ESCAPE` keyword.

The following code finds all rows that contain the `%` symbol:

```
SELECT *
FROM my_table
WHERE txt LIKE '%!%%' ESCAPE '!';

+------+-------------------+
| id   | txt               |
+------+-------------------+
|    4 | I'm 100% positive. |
+------+-------------------+
```

After the `ESCAPE` keyword, we've declared `!` as the escape character, so when the `!` is put in front of the middle `%` in `%!%%`, `!%` is interpreted as `%`.

`LIKE` is useful when searching for a particular string of characters. For more advanced pattern searches, you can use regular expressions, which are covered in the regular expressions section later in this chapter.

Math Operators

Math operators are math symbols that can be used in SQL. The math operator in the code block (`/`) is bolded:

```
SELECT salary / 52 AS weekly_pay
FROM my_table;
```

Table 7-8 lists the math operators in SQL.

Table 7-8. Math operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
% (MOD in *Oracle*)	Modulo (remainder)

NOTE

In *PostgreSQL*, *SQL Server*, and *SQLite*, dividing an integer by an integer results in an integer:

```
SELECT 15/2;
7
```

If you want the result to include decimals, you can either divide by a decimal or use the CAST function:

```
SELECT 15/2.0;
7.5

 -- PostgreSQL and 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
```

Other math operators include:

- *Bitwise operators* such as `&` (AND), `|` (OR), and `^` (XOR) for working with bits (0 and 1 values).
- *Assignment operators* such as `+=` (add equals) and `-=` (subtract equals) for updating values in a table.

Aggregate Functions

An *aggregate function* performs a calculation on many rows of data and results in a single value. Table 7-9 lists the five basic aggregate functions in SQL.

Table 7-9. Basic aggregate functions

Function	Description
`COUNT()`	Counts the number of values
`SUM()`	Calculates the sum of a column
`AVG()`	Calculates the average of a column
`MIN()`	Finds the minimum of a column
`MAX()`	Finds the maximum of a column

Aggregate functions apply calculations to non-null values in a column. The only exception is `COUNT(*)`, which counts *all* rows, including null values.

You can also aggregate multiple rows into a single list using functions like `ARRAY_AGG`, `GROUP_CONCAT`, `LISTAGG`, and `STRING_AGG`. More details can be found in the Aggregate rows into a single value or list section in Chapter 8.

NOTE

Oracle supports additional aggregate functions like median (MEDIAN), mode (STATS_MODE), and standard deviation (STDDEV).

Aggregate functions (bolded in the example) are located in the SELECT and HAVING clauses of a query:

```
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;
```

WARNING

If you choose to have both aggregate and nonaggregate columns in the SELECT statement, you *must* include all nonaggregate columns in the GROUP BY clause (region in the preceding example).

Some RDBMSs will throw an error if you do not do this. Other RDBMSs (such as *SQLite*), will not throw an error and allow the statement to run, even though the results returned will be *inaccurate*. It is good practice to double-check your results to make sure they make sense.

MIN/MAX Versus LEAST/GREATEST

The `MIN` and `MAX` functions find the smallest and largest values within a column.

The `LEAST` and `GREATEST` functions find the smallest and largest values within a row. Inputs can be numeric, string, or datetime values. If one value is `NULL`, the function returns `NULL`.

The following table shows the total miles run each quarter, and the query finds the miles run in the best quarter:

```
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 |
+--------+------------+
```

Numeric Functions

Numeric functions can be applied to columns with numeric data types. This section covers common numeric functions in SQL.

Apply Math Functions

There are multiple types of math calculations in SQL:

Math Operators
: Calculations using symbols such as `+`, `-`, `*`, `/`, and `%`

Aggregate Functions
: Calculations that summarize an entire column of data into a single value such as `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`

Math Functions
: Calculations using keywords that apply to each row of data such as `SQRT`, `LOG`, and more that are listed in Table 7-10

NOTE

SQLite only supports the `ABS` function. Other math functions need to be manually enabled. More details can be found on the math functions page on the SQLite website.

Table 7-10. Math functions

Category	Function	Description	Code	Result
Positive and Negative Values	`ABS`	Absolute value	`SELECT ABS(-5);`	5
	`SIGN`	Returns –1, 0, or 1 depending on if a number is negative, zero, or positive	`SELECT SIGN(-5);`	–1

Category	Function	Description	Code	Result
Exponents and Logarithms	`POWER`	*x* raised to the power of *y*	`SELECT POWER(5,2);`	25
	`SQRT`	Square root	`SELECT SQRT(25);`	5
	`EXP`	*e* (=2.71828) raised to the power of *x*	`SELECT EXP(2);`	7.389
	`LOG` (`LOG(y,x)` in *SQL Server*)	Log of *y* base *x*	`SELECT LOG(2,10);` `SELECT LOG(10,2);`	3.322
	`LN` (`LOG` in *SQL Server*)	Natural log (base *e*)	`SELECT LN(10);` `SELECT LOG(10);`	2.303
	`LOG10` (`LOG(10,x)` in *Oracle*)	Log base 10	`SELECT LOG10(100);` `SELECT LOG(10,100) FROM dual;`	2
Other	`MOD` (`x%y` in *SQL Server*)	Remainder of *x* / *y*	`SELECT MOD(12,5);` `SELECT 12%5;`	2
	`PI` (not available in *Oracle*)	Value of pi	`SELECT PI();`	3.14159
	`COS`, `SIN`, etc.	Cosine, sine, and other trig functions (input is in radians)	`SELECT COS(.78);`	0.711

Generate Random Numbers

Table 7-11 shows how to generate a random number in each RDBMS. In some cases, you can input a *seed* so that the random number generated is the same each time.

Table 7-11. Random number generator

RDBMS	Code	Range of Results

MySQL, SQL Server	`SELECT RAND();`
	
	`-- Optional seed`
	`SELECT RAND(22);`
		0 to 1

Oracle	`SELECT DBMS_RANDOM.VALUE`
	`FROM dual;`
		0 to 1

	`SELECT DBMS_RANDOM.RANDOM`
	`FROM dual;`
		–2E31 to +2E31

PostgreSQL	`SELECT RANDOM();`	0 to 1
SQLite	`SELECT RANDOM();`	–9E18 to +9E18

The random number function is sometimes used to return a few random rows of a table. While not the most efficient query (since the table has to be sorted), it is a quick hack:

```
-- Return 5 random rows
SELECT *
FROM my_table
ORDER BY RANDOM()
LIMIT 5;
```

Oracle and *SQL Server* allow you to randomly sample a table:

```
-- Return random 20% of rows in Oracle
SELECT *
FROM my_table
SAMPLE(20);

-- Return random 100 rows in SQL Server
SELECT *
```

```
FROM my_table
TABLESAMPLE(100 ROWS);
```

Round and Truncate Numbers

Table 7-12 shows the various ways to round numbers in each RDBMS.

Table 7-12. Rounding options

Function	Description	Code	Output
`CEIL` (`CEILING` in *SQL Server*)	Rounds up to the nearest integer	`SELECT CEIL(98.7654);` `SELECT CEILING(98.7654);`	99
`FLOOR`	Rounds down to the nearest integer	`SELECT FLOOR(98.7654);`	98
`ROUND`	Rounds to a specific number of decimal places, defaults to 0 decimals	`SELECT ROUND(98.7654,2);`	98.77
`TRUNC` (`TRUNCATE` in *MySQL*; `ROUND(x,y,1)` in *SQL Server*)	Cuts off number at a specific number of decimal places, default to 0 decimals	`SELECT TRUNC(98.7654,2);` `SELECT TRUNCATE(98.7654,2);` `SELECT ROUND(98.7654,2,1);`	98.76

NOTE

SQLite only supports the `ROUND` function. Other rounding options need to be manually enabled. More details can be found on the math functions page on the SQLite website.

Convert Data to a Numeric Data Type

The `CAST` function is used to convert between various data types, and is often used for numeric data.

In the following example, we want to compare a string column with a numeric column

Here is a table with a string column:

```
+------+---------+
| id   | str_col |
+------+---------+
|    1 | 1.33    |
|    2 | 5.5     |
|    3 | 7.8     |
+------+---------+
```

Try to compare the string column with numeric value:

```
SELECT *
FROM my_table
WHERE str_col > 3;

-- MySQL, Oracle, and SQLite results
+------+---------+
| id   | str_col |
+------+---------+
|    2 | 5.5     |
|    3 | 7.8     |
+------+---------+

-- PostgreSQL and SQL Server results
Error
```

NOTE

In *MySQL*, *Oracle*, and *SQLite*, the query returns the correct results because the string column is recognized as a numeric column when the > operator is introduced.

In *PostgreSQL* and *SQL Server*, you must explicitly CAST the string column into a numeric column.

Cast the string column to a decimal column to compare it with a number:

```
SELECT *
FROM my_table
WHERE CAST(str_col AS DECIMAL) > 3;

 id | str_col
----+---------
  2 | 5.5
  3 | 7.8
```

NOTE

Using CAST does not permanently change the data type of the column; it is only for the duration of the query. To permanently change the data type of a column, you can alter the table.

String Functions

String functions can be applied to columns with string data types. This section covers common string operations in SQL.

Find the Length of a String

Use the LENGTH function.

In the SELECT clause:

```
SELECT LENGTH(name)
FROM my_table;
```

In the WHERE clause:

```
SELECT *
FROM my_table
WHERE LENGTH(name) < 10;
```

In *SQL Server*, use LEN instead of LENGTH.

NOTE

Most RDBMSs exclude trailing spaces when calculating the length of a string, while *Oracle* includes them.

Example string: `'Al   '`

Length: 2

Length in Oracle: 5

To exclude trailing spaces in Oracle, use the TRIM function:

```
SELECT LENGTH(TRIM(name))
FROM my_table;
```

Change the Case of a String

Use the UPPER or LOWER function.

UPPER:

```
SELECT UPPER(type)
FROM my_table;
```

LOWER:

```
SELECT *
FROM my_table
WHERE LOWER(type) = 'public';
```

Oracle and *PostgreSQL* also have `INITCAP(string)` to uppercase the first letter of each word in a string and lowercase the other letters.

Trim Unwanted Characters Around a String

Use the `TRIM` function to remove both leading and trailing characters around a string value. The following table has several characters that we'd like to remove:

```
SELECT * FROM my_table;

+----------------+
| color          |
+----------------+
| !!red          |
|  .orange!      |
|    ..yellow..  |
+----------------+
```

Remove spaces around a string

By default, `TRIM` removes spaces from both the left and right sides of a string:

```
SELECT TRIM(color) AS color_clean
FROM my_table;

+-------------+
| color_clean |
+-------------+
| !!red       |
| .orange!    |
| ..yellow..  |
+-------------+
```

Remove other characters around a string

You can specify other characters to remove besides a single space. The following code removes exclamation marks around a string:

```
SELECT TRIM('!' FROM color) AS color_clean
FROM my_table;

+----------------+
| color_clean    |
+----------------+
| red            |
|  .orange       |
|    ..yellow..  |
+----------------+
```

In *SQLite*, use `TRIM(color, '!')` instead.

Remove characters from the left or right side of a string

There are two options for removing characters on either side of a string.

Option 1: `TRIM(LEADING .. )` *and* `TRIM(TRAILING .. )`

In *MySQL*, *Oracle* and *PostgreSQL*, you can remove characters from either the left or right side of a string with `TRIM(LEADING ..)` and `TRIM(TRAILING ..)`, respectively. The following code removes exclamation marks from the beginning of a string:

```
SELECT TRIM(LEADING '!' FROM color) AS color_clean
FROM my_table;

+----------------+
| color_clean    |
+----------------+
| red            |
|  .orange!      |
|    ..yellow..  |
+----------------+
```

Option 2: `LTRIM` *and* `RTRIM`

Use the keywords `LTRIM` and `RTRIM` to remove characters from either the left or right side of a string, respectively.

In *Oracle*, *PostgreSQL*, and *SQLite*, all unwanted characters can be listed within a single string. The following code removes periods, exclamation marks, and spaces from the beginning of a string:

```
SELECT LTRIM(color, '.! ') AS color_clean
FROM my_table;

+-------------+
| color_clean |
+-------------+
| red         |
| orange!     |
| yellow..    |
+-------------+
```

In *MySQL* and *SQL Server*, only whitespace characters can be removed using `LTRIM(color)` or `RTRIM(color)`.

Concatenate Strings

Use the `CONCAT` function or the concatenation operator (`||`).

```
-- MySQL, PostgreSQL, and SQL Server
SELECT CONCAT(id, '_', name) AS id_name
FROM my_table;

-- Oracle, PostgreSQL, and SQLite
SELECT id || '_' || name AS id_name
FROM my_table;

+-----------+
| id_name   |
+-----------+
| 1_Boots   |
| 2_Pumpkin |
| 3_Tiger   |
+-----------+
```

Search for Text in a String

There are two approaches to search for text in a string.

Approach 1: Does the text appear in the string or not?
: Use the LIKE operator to determine whether text appears in a string or not. With the following query, only rows that contain the text some will be returned:

```
SELECT *
FROM my_table
WHERE my_text LIKE '%some%';
```

More details can be found in the LIKE section earlier in this chapter.

Approach 2: Where does the text appear in the string?
: Use the INSTR/POSITION/CHARINDEX function to determine the location of text in a string.

Table 7-13 lists the parameters required by the location functions in each RDBMS.

Table 7-13. Functions to find the location of text in a string

RDBMS	Code Format

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*)

The inputs are:

- *string* (*required*): the string you are searching in (i.e., the name of a VARCHAR column)
- *substring* (*required*): the string you are searching for (i.e., a character, a word, etc.)

- *position* (*optional*): the starting position for the search. The default is to start at the first character (1). If *position* is negative, the search begins at the end of the string.
- *occurrence* (*optional*): the first/second/third, etc. time the substring appears in the string. The default is the first occurrence (1).

Here is a sample table:

```
+-------------------------------+
| my_text                       |
+-------------------------------+
| Here is some text.            |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)      |
+-------------------------------+
```

Find the location of the substring `some` within the string `my_text`:

```
SELECT INSTR(my_text, 'some') AS some_location
FROM my_table;

+---------------+
| some_location |
+---------------+
|             9 |
|             5 |
|             5 |
+---------------+
```

Counting in SQL Starts at 1

Unlike other programming languages that are zero-indexed (the count starts from 0), the count starts from 1 in SQL.

The 9 in the preceding output means the ninth character.

NOTE

In *Oracle*, regular expressions can also be used to search for a substring using `REGEXP_INSTR`. More details are in the regular expressions in Oracle section.

Extract a Portion of a String

Use the `SUBSTR` or `SUBSTRING` function. The function name and inputs differ for each RDBMS:

```
-- MySQL, Oracle, PostgreSQL, and SQLite
SUBSTR(string, start, length)

-- MySQL
SUBSTR(string FROM start FOR length)

-- MySQL, PostgreSQL, and SQL Server
SUBSTRING(string, start, length)

-- MySQL and PostgreSQL
SUBSTRING(string FROM start FOR length)
```

The inputs are:

- *string* (*required*): the string you are searching in (i.e., the name of a `VARCHAR` column)
- *start* (*required*): the starting location of the search. If *start* is set to `1`, the search will start from the first character, `2` is the second character, and so on. If *start* is set to `0`, it will be treated like a 1. If *start* is negative, the search will start from the last character.
- *length* (*optional*): the length of the string returned. If *length* is omitted, then all characters from the *start* to the end of the string will be returned. In *SQL Server*, *length* is required. Here is a sample table:

```
+-------------------------------+
| my_text                       |
+-------------------------------+
| Here is some text.            |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)     |
+-------------------------------+
```

Extract a substring:

```
SELECT SUBSTR(my_text, 14, 8) AS sub_str
FROM my_table;

+----------+
| sub_str  |
+----------+
| text.    |
| ers - 1  |
| tuation! |
+----------+
```

NOTE

In *Oracle*, regular expressions can also be used to extract a substring using REGEXP_SUBSTR. More details are in the regular expressions in Oracle section.

Replace Text in a String

Use the REPLACE function. Note the order of the inputs for the function:

```
REPLACE(string, old_string, new_string)
```

Here is a sample table:

```
+-------------------------------+
| my_text                       |
+-------------------------------+
| Here is some text.            |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :)     |
+------------------------------+
```

Replace the word some with the word 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! :)     |
+-----------------------------+
```

NOTE

In *Oracle* and *PostgreSQL*, regular expressions can also be used to replace a string using REGEXP_REPLACE. More details are in the regular expressions in Oracle and regular expressions in PostgreSQL sections.

Delete Text from a String

You can use the REPLACE function, but specify an empty string as the replace value.

Replace the word some with an empty string:

```
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! :)     |
+-------------------------+
```

Use Regular Expressions

Regular expressions allow you to match complex patterns. For example, finding all words that have exactly five letters or finding all words that start with a capital letter.

Imagine you have the following recipe for taco seasoning:

```
- 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
```

You want to exclude the amounts and just have a list of ingredients. To do so, you can write a regular expression to extract all of the text that follows the term `spoon`.

The regular expression would look like:

```
(?<=spoon ).*$
```

and the results would look like:

```
chili powder
ground cumin
paprika
garlic powder
onion powder
crushed red pepper flakes
dried oregano
```

The regular expression went through all of the text and extracted any text that was sandwiched between the term `spoon` and the end of the line.

A couple things to note about regular expressions:

- Regular expression syntax is not intuitive. It is helpful to break down the meaning of each part of a regular expression using an online tool, such as Regex101.
- Regular expressions are not SQL-specific. They can be used within many programming languages and text editors.
- RegexOne provides a quick introductory tutorial. You can also reference Thomas Nield's O'Reilly post, "An Introduction to Regular Expressions."

TIP

Instead of memorizing regular expression syntax, I recommend finding existing regular expressions and modifying them to fit your needs.

For the previous regular expression, I searched for "regular expression text after string."

The second Google search result got me to `(?<=WORD).*$`. I used Regex101 to understand each part of the regular expression, and finally replaced `WORD` with `spoon`.

Regular expression functions vary widely by RDBMS, so there is a separate section for each one. SQLite does not support regular expressions by default, but they can be implemented. More details can be found in the SQLite documentation.

Regular Expressions in MySQL

Use `REGEXP` to look for a regular expression pattern anywhere in a string.

Here is a sample table:

```
+----------------------------+------------+
| title                      | city       |
+----------------------------+------------+
| 10 Things I Hate About You | Seattle     |
| 22 Jump Street             | New Orleans |
| The Blues Brothers         | Chicago     |
| Ferris Bueller's Day Off   | Chi         |
+----------------------------+-------------+
```

Find all variant spellings of Chicago:

```
SELECT *
FROM movies
WHERE city REGEXP '(Chicago|CHI|Chitown)';

+--------------------------+---------+
| title                    | city    |
+--------------------------+---------+
| The Blues Brothers       | Chicago |
| Ferris Bueller's Day Off | Chi     |
+--------------------------+---------+
```

MySQL's regular expressions are case-insensitive for character strings; `CHI` and `Chi` are seen as equivalent.

Find all movies with numbers in the title:

```
SELECT *
FROM movies
WHERE title REGEXP '\\d';

+----------------------------+-------------+
| title                      | city        |
+----------------------------+-------------+
| 10 Things I Hate About You | Seattle     |
| 22 Jump Street             | New Orleans |
+----------------------------+-------------+
```

In MySQL, any single backslash in a regular expression (`\d` = any digit) needs to be changed to a double backslash.

Regular expressions in Oracle

Oracle supports many regular expression functions including:

- REGEXP_LIKE matches a regular expression pattern within the text.
- REGEXP_COUNT counts the number of times a pattern appears in the text.
- REGEXP_INSTR locates the positions that a pattern appears in the text.
- REGEXP_SUBSTR returns the substrings in the text that match a pattern.
- REGEXP_REPLACE replaces substrings that match a pattern with other text.

Here is a sample table:

```
 TITLE                          CITY
---------------------------- -------------
 10 Things I Hate About You   Seattle
 22 Jump Street               New Orleans
 The Blues Brothers           Chicago
 Ferris Bueller's Day Off     Chi
```

Find all movies with numbers in the title:

```
SELECT *
FROM movies
WHERE REGEXP_LIKE(title, '\d');

 TITLE                          CITY
---------------------------- -------------
 10 Things I Hate About You   Seattle
 22 Jump Street               New Orleans
```

NOTE

The following expressions are equivalent:

```
REGEXP_LIKE(title, \d)
REGEXP_LIKE(title, [0-9])
REGEXP_LIKE(title, [[:digit:]])
```

The third option uses POSIX regular expression syntax, which is supported by Oracle.

Count the number of capital letters in the title:

```
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
```

Find the location of the first vowel in the title:

```
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
```

Return all numbers in the title:

```
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
```

Replace all numbers in the title with the number `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
```

NOTE

More details and examples on regular expressions in Oracle can be found in the *Oracle Regular Expressions Pocket Reference* by Jonathan Gennick and Peter Linsley (O'Reilly).

Regular expressions in PostgreSQL

Use `SIMILAR TO` or `~` to look for a regular expression pattern anywhere in a string.

Here is a sample table:

```
           title             |    city
-----------------------------+-------------
 10 Things I Hate About You | Seattle
 22 Jump Street             | New Orleans
 The Blues Brothers         | Chicago
 Ferris Bueller's Day Off   | Chi
```

Find all variant spellings of Chicago:

```
SELECT *
FROM movies
WHERE city SIMILAR TO '(Chicago|CHI|Chi|Chitown)';

          title           |  city
--------------------------+---------
 The Blues Brothers       | Chicago
 Ferris Bueller's Day Off | Chi
```

PostgreSQL's regular expressions are case-sensitive for character strings; CHI and Chi are seen as different values.

SIMILAR TO Versus ~

SIMILAR TO offers limited regular expression capabilities, and is most often used to offer multiple alternatives (Chicago|CHI|Chi). Other common regex symbols to use with SIMILAR TO are * (0 or more), + (1 or more), and {} (exact number of times).

The tilde (~) should be used for more advanced regular expressions along with POSIX syntax, which is another flavor of regular expression that PostgreSQL supports.

The full list of supported symbols can be found in the PostgreSQL documentation.

The following example uses ~ instead of SIMILAR TO.

Find all movies with numbers in the title:

```
SELECT *
FROM movies
WHERE title ~ '\d';

+-----------------------------+-------------+
| title                       | city        |
+-----------------------------+-------------+
| 10 Things I Hate About You  | Seattle     |
| 22 Jump Street              | New Orleans |
+-----------------------------+-------------+
```

PostgreSQL also supports `REGEXP_REPLACE`, which allows you to replace characters in a string that match a particular pattern.

Replace all numbers in the title with the number `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
```

The regular expression `\d` is equivalent to `[0-9]` and `[[:digit::]]`.

Regular expressions in SQL Server

SQL Server supports a very limited amount of regular expressions through its `LIKE` keyword.

Here is a sample table:

```
 title                         city
---------------------------- -------------
 10 Things I Hate About You    Seattle
 22 Jump Street                New Orleans
 The Blues Brothers            Chicago
 Ferris Bueller's Day Off      Chi
```

SQL Server uses a slightly different flavor of regular expression syntax, which is detailed in the Microsoft documentation.

Find all movies with numbers in the title:

```
SELECT *
FROM movies
WHERE title LIKE '%[0-9]%';
```

```
 title                         city
---------------------------- ------------
 10 Things I Hate About You   Seattle
 22 Jump Street               New Orleans
```

Convert Data to a String Data Type

When string functions are applied to nonstring data types, it is typically not an issue even though there is a data type mismatch.

The following table has a numeric column called `numbers`:

```
+---------+
| numbers |
+---------+
| 1.33    |
| 2.5     |
| 3.777   |
+---------+
```

When the string function `LENGTH` (or `LEN` in *SQL Server*) is applied on the numeric column, the statement executes without error in most RDBMSs:

```
SELECT LENGTH(numbers) AS len_num
FROM my_table;

-- MySQL, Oracle, SQL Server, and SQLite results
+---------+
| len_num |
+---------+
|       4 |
|       3 |
|       5 |
+---------+

-- PostgreSQL results
Error
```

In *PostgreSQL*, you must explicitly `CAST` the numeric column into a string column:

```
SELECT LENGTH(CAST(numbers AS CHAR(5))) AS len_num
FROM my_table;

 len_num
---------
       4
       3
       5
```

NOTE

Using CAST does not permanently change the data type of the column—it is only for the duration of the query. To permanently change the data type of a column, you can alter the table.

Datetime Functions

Datetime functions can be applied to columns with datetime data types. This section covers common datetime functions in SQL.

Return the Current Date or Time

The following statements return the current date, current time, and current date and time:

```
-- MySQL, PostgreSQL, and 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;
```

There are many other functions equivalent to these including `CURDATE()` in *MySQL*, `GETDATE()` in *SQL Server*, etc.

The following three situations show how these functions are used in practice.

Display the current time:

```
SELECT CURRENT_TIME;

+--------------+
| current_time |
+--------------+
| 20:53:35     |
+--------------+
```

Create a table that marks the date and time of creation:

```
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 |
 +------+---------------------+
```

Find all rows of data before a certain date:

```
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 |
+------+---------------------+
```

Add or Subtract a Date or Time Interval

You can add or subtract various time intervals (years, months, days, hours, minutes, seconds, etc.) from date and time values.

Table 7-14 lists the ways to subtract a day.

Table 7-14. Return yesterday's date

RDBMS	Code

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');`

Table 7-15 lists the ways to add three hours.

Table 7-15. Return the date and time three hours from now

RDBMS	Code

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');`

Find the Difference Between Two Dates or Times

You can find the difference between two dates, times, or datetimes in terms of various time intervals (years, months, days, hours, minutes, seconds, etc.).

Finding a date difference

Given a start and end date, Table 7-16 lists the ways to find the days between the two dates.

Here is a sample table:

```
+------------+------------+
| start_date | end_date   |
+------------+------------+
| 2016-10-10 | 2020-11-11 |
| 2019-03-03 | 2021-04-04 |
+------------+------------+
```

Table 7-16. Days between two dates

RDBMS	Code
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;`

After running the code in the table, these are the results:

```
-- MySQL, Oracle, SQL Server, and SQLite
+----------+
| day_diff |
+----------+
|     1493 |
|      763 |
+----------+

-- PostgreSQL

       day_diff
---------------------
 4 years 1 mon 1 day
 2 years 1 mon 1 day
```

Finding a time difference

Given a start and end time, Table 7-17 lists the ways to find the seconds between the two times.

Here is a sample table:

```
+------------+----------+
| start_time | end_time |
+------------+----------+
| 10:30:00   | 11:30:00 |
| 14:50:32   | 15:22:45 |
+------------+----------+
```

Table 7-17. Seconds between two times

RDBMS	Code

MySQL	`SELECT TIMEDIFF(end_time, start_time)`
	`       AS time_diff`
	`FROM   my_table;`

Oracle	No time data type

PostgreSQL	`SELECT EXTRACT(epoch from end_time -`
	`       start_time) AS time_diff`
	`FROM   my_table;`

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;`

After running the code in the table, these are the results:

```
-- MySQL
+-----------+
| time_diff |
+-----------+
| 01:00:00  |
| 00:32:13  |
+-----------+
-- PostgreSQL, SQL Server, and SQLite

 time_diff
-----------
 3600
 1933
```

Finding a datetime difference

Given a start and end datetime, Table 7-18 lists the ways to find the number of hours between the two datetimes.

Here is a sample table:

```
+---------------------+---------------------+
| 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 |
+---------------------+---------------------+
```

Table 7-18. Hours between two datetimes

RDBMS	Code
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;`

After running the code in the table, these are the results:

```
-- MySQL, SQL Server, and 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
```

NOTE

The *PostgreSQL* result is lengthy:

```
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
```

Use the EXTRACT function to pull out only the year field.

```
SELECT EXTRACT(year FROM
               AGE(end_dt, start_dt))
FROM my_table;

 date_part
-----------
         4
         2
```

Extract a Part of a Date or Time

There are multiple ways to extract a time unit (month, hour, etc.) from a date or time value. Table 7-19 shows how to do so, specifically for the month time unit.

Table 7-19. Extract the month from a date

RDBMS	Code

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);`
	`SELECT DATE_PART('month', CURRENT_DATE);`

RDBMS	Code

SQL Server	`SELECT DATEPART(month, CURRENT_TIMESTAMP);`
	`SELECT MONTH(CURRENT_TIMESTAMP);`

SQLite	`SELECT strftime('%m', CURRENT_DATE);`

Both *MySQL* and *SQL Server* support time unit specific functions like `MONTH()`, as seen in Table 7-19.

- *MySQL* supports `YEAR()`, `QUARTER()`, `MONTH()`, `WEEK()`, `DAY()`, `HOUR()`, `MINUTE()`, and `SECOND()`.
- *SQL Server* supports `YEAR()`, `MONTH()`, and `DAY()`.

You can replace the `month` or `%m` values in Table 7-19 with other time units. Table 7-20 lists the time units accepted by each RDBMS.

Table 7-20. Time unit options

MySQL	Oracle	PostgreSQL	SQL Server	SQLite
microsecond second minute hour day week month quarter year	second minute hour day month year	microsecond millisecond second minute hour day dow week month quarter year decade century	nanosecond microsecond millisecond second minute hour week weekday day dayofyear month quarter year	%f (fractional second) %S (second) %s (seconds since 1970-01-01) %M (minute) %H (hour) %J (Julian day number) %w (day of week) %d (day of month) %j (day of year) %W (week of year) %m (month) %Y (year)

NOTE

You can also extract a time unit from a string value. The code can be found in Table 7-28: Extract year from a string.

Determine the Day of the Week of a Date

Given a date, determine the day of the week:

- Date: `2020-03-16`
- Numeric day of the week: `2` (Sunday is the first day)
- Day of the week: `Monday`

Table 7-21 returns the numeric day of the week of a given date. Sunday is the first day, Monday the second day, and so on.

Table 7-21. Return the numeric day of the week

RDBMS	Code	Range of Values
MySQL	`SELECT DAYOFWEEK('2020-03-16');`	1 to 7

Oracle	`SELECT TO_CHAR(`
	`       date '2020-03-16', 'd')`
	`FROM   dual;`
		1 to 7

PostgreSQL	`SELECT DATE_PART('dow',`
	`       date '2020-03-16');`
		0 to 6

SQL Server	`SELECT DATEPART(weekday,`
	`       '2020-03-16');`
		1 to 7

SQLite	`SELECT strftime('%w',`
	`       '2020-03-16');`
		0 to 6

Table 7-22 returns the day of the week of a given date.

Table 7-22. Return the day of the week

RDBMS	Code
MySQL	`SELECT DAYNAME('2020-03-16');`

Oracle	`SELECT TO_CHAR(date '2020-03-16', 'day')`
	`FROM   dual;`

PostgreSQL	`SELECT TO_CHAR(date '2020-03-16', 'day');`
SQL Server	`SELECT DATENAME(weekday, '2020-03-16');`
SQLite	Not available

Round a Date to the Nearest Time Unit

Oracle and *PostgreSQL* support rounding and truncating (also known as rounding down).

Rounding in Oracle

Oracle supports rounding and truncating a date to the nearest year, month, or day (first day of the week).

To round down to the first of the month:

```
SELECT TRUNC(date '2020-02-25', 'month')
FROM   dual;

01-FEB-20
```

To round to the nearest month:

```
SELECT ROUND(date '2020-02-25', 'month')
FROM   dual;

01-MAR-20
```

Rounding in PostgreSQL

PostgreSQL supports truncating a date to the nearest year, quarter, month, week (first day of the week), day, hour, minute, or second. Additional time units can be found in the PostgreSQL documentation.

To round down to the first of the month:

```
SELECT DATE_TRUNC('month', DATE '2020-02-25');

2020-02-01 00:00:00-06
```

To round down to the minute:

```
SELECT DATE_TRUNC('minute', TIME '10:30:59.12345');

10:30:00
```

Convert a String to a Datetime Data Type

There are two ways to convert a string to a datetime data type:

- Use the `CAST` function for a simple case.
- Use `STR_TO_DATE/TO_DATE/CONVERT` for a custom case.

The CAST function

If a string column contains dates in a standard format, you can use the `CAST` function to turn it into a date data type.

Table 7-23 shows the code for converting to a date data type.

Table 7-23. Convert a string to a date

RDBMS	Required Date Format	Code
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');`

Table 7-24 shows the code for converting to a time data type.

Table 7-24. Convert a string to a time

RDBMS	Required Time Format	Code
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');`

Table 7-25 shows the code for converting to a datetime data type.

Table 7-25. Convert a string to a datetime

RDBMS	Required Datetime Format	Code
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');`

The `CAST` function can also be used to convert dates to numeric and string data types.

The STR_TO_DATE, TO_DATE, and CONVERT functions

For dates and times not in the standard `YYYY-MM-DD`/`DD-MON-YYYY`/`hh:mm:ss` formats, use a string to date or a string to time function instead.

Table 7-26 lists the string to date and string to time functions for each RDBMS. The example strings in the code are in nonstandard formats `MM-DD-YY` and `hhmm`.

Table 7-26. String to date and string to time functions

RDBMS	String to date	String to 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	No nonstardard date function	No non-standard time function

NOTE

SQL Server uses the `CONVERT` function to change a string to a datetime data type. `VARCHAR` is the original data type, `10-15-22` is the date, and `105` stands for the format `MM-DD-YYYY`.

Other date formats are `MM/DD/YYYY` (101), `YYYY.MM.DD` (102), `DD/MM/YYYY` (103), and `DD.MM.YYYY` (104). More formats are listed in the Microsoft documentation.

The time formats are `hh:mi:ss` (108) and `hh:mi:ss:mmm` (114), neither which match the format in Table 7-26, which is why the time can't be read in by SQL Server using `CONVERT`.

You can replace the `%H%i` or `HH24MI` values in Table 7-26 with other time units. Table 7-27 lists common format specifiers for *MySQL*, *Oracle*, and *PostgreSQL*.

Table 7-27. Datetime format specifiers

MySQL	Oracle and PostgreSQL	Description
%Y	YYYY	4-digit year
%y	YY	2-digit year
%m	MM	Numeric month (1–12)
%b	MON	Abbreviated month (Jan–Dec)
%M	MONTH	Name of month (January–December)
%d	DD	Day (1–31)
%h	HH or HH12	12 hours (1–12)
%H	HH24	24 hours (0–23)
%i	MI	Minutes (0–59)
%s	SS	Seconds (0–59)

Apply a date function to a string column

Imagine you have the following string column:

```
str_column
10/15/2022
10/16/2023
10/17/2024
```

You want to extract the year from each date:

```
year_column
2022
2023
2024
```

Problem

You cannot use a datetime function (`EXTRACT`) on a string column (`str_column`).

Solution

First convert the string column into a date column. Then apply the datetime function. Table 7-28 lists how to do so in each RDBMS.

Table 7-28. Extract year from a string

RDBMS	Code
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;`

NOTE

SQLite does not have datetime functions, but a workaround is to use the `SUBSTR` (substring) function to extract the last four digits.

Null Functions

Null functions can be applied to any type of column and are triggered when a null value is encountered.

Return an Alternative Value if There Is a Null Value

Use the COALESCE function.

Here is a sample table:

```
+------+----------+
| id   | greeting |
+------+----------+
|    1 | hi there |
|    2 | hello!   |
|    3 | NULL     |
+------+----------+
```

When there is no greeting, return hi:

```
SELECT COALESCE(greeting, 'hi') AS greeting
FROM my_table;

+----------+
| greeting |
+----------+
| hi there |
| hello!   |
| hi       |
+----------+
```

MySQL and *SQLite* also accept IFNULL(greeting, 'hi').

Oracle also accepts NVL(greeting, 'hi').

SQL Server also accepts ISNULL(greeting, 'hi').

CHAPTER 8

Advanced Querying Concepts

This chapter covers a few advanced ways of wrangling data using SQL queries, beyond the six main clauses covered in Chapter 4, *Querying Basics*, and the common keywords covered in Chapter 7, *Operators and Functions*.

Table 8-1 includes descriptions and code examples of the four concepts covered in this chapter.

Table 8-1. Advanced querying concepts

Concept	Description	Code Example

Case Statements	If a condition is met, return a particular value. Otherwise, return another value.	`SELECT house_id,`
		`  CASE WHEN flg = 1`
		`  THEN 'for sale'`
		`  ELSE 'sold' END`
		`FROM houses;`

Grouping and Summarizing	Split data into groups, aggregate the data within each group, and return a value for each *group*.	`SELECT zip, AVG(ft)`
		`FROM houses`
		`GROUP BY zip;`

Concept	Description	Code Example

Window Functions	Split data into groups, aggregate or order the data within each group, and return a value for each *row*.	`SELECT zip,`
		`  ROW_NUMBER() OVER`
		`  (PARTITION BY zip`
		`  ORDER BY price)`
		`FROM houses;`

Pivoting and Unpivoting	Turn values in a column into multiple columns or consolidate multiple columns into a single column. Supported by *Oracle* and *SQL Server*.	`-- Oracle syntax`
		`SELECT *`
		`FROM listing_info`
		`PIVOT`
		`  (COUNT(*) FOR`
		`  room IN ('bd','br'));`

This chapter describes each of the concepts in Table 8-1 in detail, along with common use cases.

Case Statements

A `CASE` statement is used to apply if-else logic within a query. For example, you could use a `CASE` statement to spell out values. If a `1` is seen, display `vip`. Otherwise, display `general admission`.

```
+--------+       +-------------------+
| ticket |       | ticket            |
+--------+       +-------------------+
|      1 |       | vip               |
|      0 |  -->  | general admission |
|      1 |       | vip               |
+--------+       +-------------------+
```

In *Oracle*, you may also see the `DECODE` function, which is an older function that operates similarly to the `CASE` statement.

NOTE

Using a `CASE` statement temporarily updates values for the duration of a query. To save the updated values, you can do so with an `UPDATE` statement.

The following two sections go over two types of `CASE` statements:

- *Simple* `CASE` statement for a *single column* of data
- *Searched* `CASE` statement for *multiple columns* of data

Display Values Based on If-Then Logic for a Single Column

To check for equality within a single column of data, use the *simple* `CASE` statement syntax.

Our goal:

Instead of displaying the values `1`/`0`/`NULL`, display the values `vip`/`reserved seating`/`general admission`:

- If `flag = 1`, then `ticket = vip`
- If `flag = 0`, then `ticket = reserved seating`
- Else, `ticket = general admission`

Here is a sample table:

```
SELECT * FROM concert;

+-------+------+
| name  | flag |
+-------+------+
| anton |    1 |
| julia |    0 |
| maren |    1 |
| sarah | NULL |
+-------+------+
```

Implement the if-else logic with a simple `CASE` statement:

```
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 |
+-------+------+-------------------+
```

If no `WHEN` clause is a match and no `ELSE` value is specified, a `NULL` will be returned.

Display Values Based on If-Then Logic for Multiple Columns

To check for any condition (=, <, `IN`, `IS NULL`, etc.) within potentially multiple columns of data, use the *searched* `CASE` statement syntax.

Our goal:

Instead of displaying the values `1/0/NULL`, display the values `vip/reserved seating/general admission`:

- If `name = anton`, then `ticket = vip`
- If `flag = 0` or `flag = 1`, then `ticket = reserved seating`
- Else, `ticket = general admission`

Here is a sample table:

```
SELECT * FROM concert;

+-------+------+
| name  | flag |
+-------+------+
| anton |    1 |
| julia |    0 |
| maren |    1 |
| sarah | NULL |
+-------+------+
```

Implement the if-else logic with a searched `CASE` statement:

```
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 |
+-------+------+-------------------+
```

If multiple conditions are met, the first listed condition takes precedence.

NOTE

To replace all `NULL` values in a column with another value, you could use a `CASE` statement, but it is more common to use the `NULL` function `COALESCE` instead.

Grouping and Summarizing

SQL allows you to separate rows into groups and summarize the rows within each group in some way, ultimately returning just one row per group.

Table 8-2 lists the concepts associated with grouping and summarizing data.

Table 8-2. Grouping and summarizing concepts

Category	Keyword	Description
The main concept	`GROUP BY`	Use the `GROUP BY` clause to separate rows of data into groups.

Ways to summarize rows within each group	`COUNT`
	`SUM`
	`MIN`
	`MAX`
	`AVG`
		These aggregate functions summarize multiple rows of data into *a single value*.

	`ARRAY_AGG`
	`GROUP_CONCAT`
	`LISTAGG`
	`STRING_AGG`
		These functions combine multiple rows of data into *a single list*.

Extensions of the `GROUP BY` clause	`ROLLUP`	Includes rows for subtotals and the grand total as well.
	`CUBE`	Includes aggregations for all possible combinations of the grouped by columns.
	`GROUPING SETS`	Allows you to specify particular groupings to display.

GROUP BY Basics

The following table shows the number of calories burned by two people:

```
SELECT * FROM workouts;
```

```
+------+----------+
| name | calories |
+------+----------+
| ally |       80 |
| ally |       75 |
| ally |       90 |
| jess |      100 |
| jess |       92 |
+------+----------+
```

To create a summary table, you need to decide how to:

1. Group the data: separate all the `name` values into two groups—`ally` and `jess`.
2. Aggregate the data within the groups: find the total `calories` within each group.

Use the `GROUP BY` clause to create a summary table:

```
SELECT name,
       SUM(calories) AS total_calories
FROM workouts
GROUP BY name;

+------+----------------+
| name | total_calories |
+------+----------------+
| ally |            245 |
| jess |            192 |
+------+----------------+
```

More details on how `GROUP BY` works behind the scenes can be found in The GROUP BY Clause section in Chapter 4.

Grouping by multiple columns

The following table shows the number of calories burned by two people during their daily workouts:

```
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 |
+------+------+------------+----------+
```

When writing a query with a `GROUP BY` clause that groups by multiple columns and/or includes multiple aggregations:

- The `SELECT` clause should include all *column names* and *aggregations* that you want to appear in the output.
- The `GROUP BY` clause should include the same *column names* that are in the `SELECT` clause.

Use the `GROUP BY` clause to summarize the stats for each person, returning both the `id` and `name` along with two aggregations:

```
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 |
+------+------+----------+----------+
```

Reduce the GROUP BY List for Efficiency

If you know that each `id` is linked to a single `name`, you can exclude the `name` column from the `GROUP BY` clause and get the same results as the previous query:

```
SELECT id,
       MAX(name) AS name,
       COUNT(date) AS workouts,
       SUM(calories) AS calories
FROM daily_workouts
GROUP BY id;
```

This runs more efficiently behind the scenes since the GROUP BY only has to occur on one column.

To compensate for dropping the name from the GROUP BY clause, you'll notice that an arbitrary aggregate function (MAX) was applied to the name column within the SELECT clause. Because there is only one name value within each id group, MAX(name) will simply return the name associated with each id.

Aggregate Rows into a Single Value or List

With the GROUP BY clause, you must specify how the rows of data within each group should be summarized using either:

- *An aggregate function to summarize rows into a single value*: COUNT, SUM, MIN, MAX, and AVG
- *A function to summarize rows into a list* (shown in the sample table): GROUP_CONCAT and others listed in Table 8-3

Here is a sample table:

```
SELECT * FROM workouts;

+------+----------+
| name | calories |
+------+----------+
| ally |       80 |
| ally |       75 |
| ally |       90 |
| jess |      100 |
| jess |       92 |
+------+----------+
```

Use GROUP_CONCAT in *MySQL* to create a list of calories:

```
SELECT name,
       GROUP_CONCAT(calories) AS calories_list
FROM workouts
GROUP BY name;

+------+---------------+
| name | calories_list |
+------+---------------+
| ally | 80,75,90      |
| jess | 100,92        |
+------+---------------+
```

The GROUP_CONCAT function differs for each RDBMS. Table 8-3 shows the syntax supported by each RDBMS:

Table 8-3. Aggregate rows into a list in each RDBMS

RDBMS	Code	Default Separator

MySQL	`GROUP_CONCAT(calories)`
	`GROUP_CONCAT(calories SEPARATOR ',')`
		Comma

Oracle	`LISTAGG(calories)`
	`LISTAGG(calories, ',')`
		No value

PostgreSQL	`ARRAY_AGG(calories)`	Comma
SQL Server	`STRING_AGG(calories, ',')`	Separator required

SQLite	`GROUP_CONCAT(calories)`
	`GROUP_CONCAT(calories, ',')`
		Comma

In *MySQL*, *Oracle*, and *SQLite*, the separator portion (',') is optional. *PostgreSQL* doesn't accept a separator, and *SQL Server* requires one.

You can also return a sorted list or a unique list of values. Table 8-4 shows the syntax supported by each RDBMS.

Table 8-4. Return a sorted or unique list of values in each RDBMS

RDBMS	Sorted List	Unique List
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)`**	Not supported
SQLite	Not supported	`GROUP_CONCAT(` **`DISTINCT`** `calories)`

ROLLUP, CUBE, and GROUPING SETS

In addition to `GROUP BY`, you can also add on the `ROLLUP`, `CUBE`, or `GROUPING SETS` keywords to include additional summary information.

The following table lists five purchases over the course of three months:

```
SELECT * FROM spendings;

 YEAR  MONTH  AMOUNT
----- ------ -------
 2019      1      20
 2019      1      30
 2020      1      42
 2020      2      37
 2020      2     100
```

The examples in this section build on the following `GROUP BY` example, which returns the total monthly spendings:

```
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*, and *SQL Server* support ROLLUP, which extends the GROUP BY by including additional rows for subtotals and the grand total.

Use ROLLUP to display the yearly and total spendings as well. The 2019, 2020, and total spending rows are added with the addition of 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
```

The preceding syntax works in *Oracle*, *PostgreSQL*, and *SQL Server*. The *MySQL* syntax is GROUP BY year, month WITH ROLLUP, which also works in *SQL Server*.

CUBE

Oracle, *PostgreSQL*, and *SQL Server* support CUBE, which extends the ROLLUP by including additional rows for all possible combinations of the columns that you are grouping by, as well as the grand total.

Use CUBE to display monthly spendings (single month across multiple years) as well. The January and Feburary spending rows are added with the addition of 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
```

The preceding syntax works in *Oracle*, *PostgreSQL*, and *SQL Server*. *SQL Server* also supports the syntax GROUP BY year, month WITH CUBE.

GROUPING SETS

Oracle, *PostgreSQL*, and *SQL Server* support GROUPING SETS, which lets you specify particular groupings that you want to display.

This data is a subset of the results generated by CUBE, only including groupings of one column at a time. In this case, only the total yearly and total monthly spendings are returned:

```
SELECT year, month,
       SUM(amount) AS total
FROM spendings
GROUP BY GROUPING SETS(year, month)
ORDER BY year, month;

 YEAR  MONTH  TOTAL
----- ------ ------
 2019            50
 2020           179
           1     92
           2    137
```

Window Functions

A *window function* (or *analytic function* in *Oracle*) is similar to an aggregate function in that they both perform a calculation on rows of data. The difference is that an aggregate function returns a single value while a window function returns a value for each row of data.

The following table lists employees along with their monthly sales. The following queries use this table to show the difference between an aggregate function and a window function.

```
SELECT * FROM sales;

+-------+-------+-------+
| name  | month | sales |
+-------+-------+-------+
| David |     3 |     2 |
| David |     4 |    11 |
| Laura |     3 |     3 |
| Laura |     4 |    14 |
| Laura |     5 |     7 |
| Laura |     6 |     1 |
+-------+-------+-------+
```

Aggregate Function

`SUM()` is an aggregate function. The following query sums up the sales for each person and returns each `name` along with its `total_sales` value.

```
SELECT name,
       SUM(sales) AS total_sales
FROM sales
GROUP BY name;

+-------+-------------+
| name  | total_sales |
+-------+-------------+
| David |          13 |
| Laura |          25 |
+-------+-------------+
```

Window Function

`ROW_NUMBER() OVER (PARTITION BY name ORDER BY month)` is a window function. In the bolded portion of the following query, for each person, a row number is generated that represents the first month, second month, etc. that they sold something. The query returns each row along with its `sale_month` value.

```
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 |
+-------+------------+
```

Breaking Down the Window Function

```
ROW_NUMBER() OVER (PARTITION BY name ORDER BY month)
```

A *window* is a group of rows. In the preceding example, there were two windows. The name `David` had a window of two rows and the name `Laura` had a window of four rows:

`ROW_NUMBER()`
: The function you want to apply to each window. Other common functions include `RANK()`, `FIRST_VALUE()`, `LAG()`, etc. This is required.

`OVER`
: This states that you are specifying a window function. This is required.

`PARTITION BY name`
: This states how you want to split your data into windows. It can be split according to one or more columns. This is optional. If excluded, the window is the entire table.

`ORDER BY month`
: This states how each window should be sorted before the function is applied. This is optional in *MySQL*, *PostgreSQL*, and *SQLite*. It is required in *Oracle* and *SQL Server*.

The following sections include examples of how window functions are used in practice.

Rank the Rows in a Table

Use the `ROW_NUMBER()`, `RANK()`, or `DENSE_RANK()` function to add a row number to each row of a table.

The following table shows the number of babies given popular names:

```
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 |
+--------+--------+--------+
```

The two following queries:

- Rank the names by popularity
- Rank the names by popularity for each gender

Rank the names by popularity:

```
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 |
+--------+--------+------------+
```

Rank the names by popularity for each gender:

```
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 Versus RANK Versus DENSE_RANK

There are three approaches to adding row numbers. Each one has a different way of handling ties.

`ROW_NUMBER` breaks the tie:

```
NAME     BABIES  POPULARITY
------- ------- -----------
Olivia       99           1
Emma         80           2
Sophia       80           3
Mia          75           4
```

`RANK` keeps the tie:

```
NAME     BABIES  POPULARITY
------- ------- -----------
Olivia       99           1
Emma         80           2
Sophia       80           2
Mia          75           4
```

`DENSE_RANK` keeps the tie and doesn't skip numbers:

```
NAME     BABIES  POPULARITY
------- ------- -----------
Olivia       99           1
Emma         80           2
Sophia       80           2
Mia          75           3
```

Return the First Value in Each Group

Use `FIRST_VALUE` and `LAST_VALUE` to return the first and last rows of a window, respectively.

The following queries break down the two-step process to return the most popular name for each gender.

Step 1: Display the most popular name for each gender.

```
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     |
+--------+--------+--------+----------+
```

Use the output as a subquery for the next step, which filters on the subquery.

Step 2: Return only the two rows containing the most popular names.

```
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      |
+--------+--------+--------+-----------+
```

In *Oracle*, exclude the `AS top_name_table` portion.

Return the Second Value in Each Group

Use `NTH_VALUE` to return a specific rank number within each window. *SQL Server* does not support `NTH_VALUE`. Instead, refer to the code in the next section, Return the first two values in each group, but only return the second value.

The following queries break down the two-step process to return the second most popular name for each gender.

Step 1: Display the second most popular name for each gender.

```
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        |
+--------+--------+--------+-------------+
```

The second parameter in `NTH_VALUE(name, 2)` is what specifies the second value in the window. This can be any positive integer.

Use the output as a subquery for the next step, which filters on the subquery.

Step 2: Return only the two rows containing the second most popular names.

```
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        |
+--------+--------+--------+-------------+
```

In *Oracle*, exclude the `AS second_name_table` portion.

Return the First Two Values in Each Group

Use `ROW_NUMBER` within a subquery to return multiple rank numbers within each group.

The following queries break down the two-step process to return the first and second most popular names for each gender.

Step 1: Display the popularity rank for each gender.

```
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 |
+--------+--------+--------+------------+
```

Use the output as a subquery for the next step, which filters on the subquery.

Step 2: Filter on the rows that contain ranks 1 and 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 |
+--------+--------+--------+------------+
```

In *Oracle*, exclude the `AS popularity_table` portion.

Return the Prior Row Value

Use `LAG` and `LEAD` to look a certain number of rows behind and ahead, respectively.

Use `LAG` to return the previous row:

```
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       |
+--------+--------+--------+------------+
```

Use `LAG(name, 2, 'No name')` to return the names from two rows prior and replace `NULL` values with `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         |
+--------+--------+--------+--------------+
```

The `LAG` and `LEAD` functions each take three arguments: `LAG(name, 2, 'None')`

- `name` is the value you want to return. It is required.
- `2` is the row offset. It is optional and defaults to 1.
- `'No name'` is the value that will be returned when there is no value. It is optional and defaults to `NULL`.

Calculate the Moving Average

Use a combination of the `AVG` function and the `ROWS BETWEEN` clause to calculate the moving average.

Here is a sample table:

```
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 |
+-------+-------+-------+
```

For each person, find the three-month moving average of sales, from two months prior to the current month:

```
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 |
+-------+-------+-------+----------------+
```

NOTE

The preceding example looks at the two rows prior through the current row:

```
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

You can also look at the next rows using the FOLLOWING keyword:

```
ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING
```

These ranges are sometimes referred to as *sliding windows*.

Calculate the Running Total

Use a combination of the SUM function and the ROWS BETWEEN UNBOUNDED clause to calculate the running total.

For each person, find the running total of sales, up to the current month:

```
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 |
+-------+-------+-------+---------------+
```

NOTE

Here, we calculated the running total for each person. To calculate the running total for the entire table, you can remove the `PARTITION BY name` portion of the code.

ROWS Versus RANGE

An alternative to `ROWS BETWEEN` is `RANGE BETWEEN`. The following query calculates the running total of sales made by all employees, using both the `ROWS` and `RANGE` keywords:

```
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 |
+-------+-------+---------+-----------+
```

The difference between the two is that `RANGE` will return the same running total value for each `month` (since the data was ordered by `month`), while `ROWS` will have a different running total value for each row.

Pivoting and Unpivoting

Oracle and *SQL Server* support the PIVOT and UNPIVOT operations. PIVOT takes a single column and splits it out into multiple columns. UNPIVOT takes multiple columns and consolidates them into a single column.

Break Up the Values of a Column into Multiple Columns

Imagine you have a table where each row is a person followed by a fruit that they ate that day. You want to take the fruit column and create a separate column for each fruit.

Here is a sample table:

```
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   |
+------+-------+-------------+
```

Expected output:

```
+-------+-------------+-----------+------------+
| name  | strawberries | grapefruit | watermelon |
+-------+-------------+-----------+------------+
| Henry |            1 |          1 |          1 |
| Lily  |            2 |          0 |          2 |
+-------+-------------+-----------+------------+
```

Use the PIVOT operation in *Oracle* and *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;
```

Within the PIVOT section, the id and fruit columns are referenced, but the name column is not. Therefore, the name column will stay as its own column in the final result and each fruit will be turned into a new column.

The values of the table are the count of the number of rows in the original table that contained each particular name/fruit combination.

PIVOT Alternative: CASE

A more manual way of doing a PIVOT is to use a CASE statement instead in *MySQL*, *PostgreSQL*, and *SQLite* since they do not support 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;
```

List the Values of Multiple Columns in a Single Column

Imagine you have a table where each row is a person followed by multiple columns that contain their favorite fruits. You want to rearrange the data so that all of the fruits are in one column.

Here is a sample table:

```
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      |
+----+-------+-----------+-----------+-----------+
```

Expected output:

```
+----+-------+-----------+------+
| 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 |
+----+-------+-----------+------+
```

Use the `UNPIVOT` operation in *Oracle* and *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 <> '';
```

The UNPIVOT section takes the columns fruit_one, fruit_two, and fruit_thr and consolidates them into a single column called fruit.

Once that's done, you can go ahead and use a typical SELECT statement to pull the original id and name columns along with the newly created fruit column.

UNPIVOT Alternative: UNION ALL

A more manual way of doing an UNPIVOT is to use UNION ALL instead in *MySQL*, *PostgreSQL*, and *SQLite* since they do not support 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 does not support inserting a constant into a column within a query (`1 AS rank`, `2 AS rank`, and `3 AS rank`). Remove those lines for the code to run.

CHAPTER 9

Working with Multiple Tables and Queries

This chapter covers how to bring together multiple tables by either joining them or using union operators, and also how to work with multiple queries using common table expressions.

Table 9-1 includes descriptions and code examples of the three concepts covered in this chapter.

Table 9-1. Working with multiple tables and queries

Concept	Description	Code Example
Joining Tables	Combine the columns of two tables based on matching rows.	`SELECT c.id, l.city FROM customers c INNER JOIN loc l ON c.lid = l.id;`
Union Operators	Combine the rows of two tables based on matching columns.	`SELECT name, city FROM employees; UNION SELECT name, city FROM customers;`

Concept	Description	Code Example

Common Table Expressions	Temporarily save the output of a query, for another query to reference it. Also includes recursive and hierarchical queries.	`WITH my_cte AS (`
		 `  SELECT name,`
		 `    SUM(order_id)`
		 `    AS num_orders`
		 `  FROM customers`
		 `  GROUP BY name)`
		
		 `SELECT MAX(num_orders)`
		 `FROM my_cte;`

Joining Tables

In SQL, *joining* means combining data from multiple tables together within a single query. The following two tables list the state a person lives in and the pets they own:

```
-- states               -- pets
+------+-------+      +------+------+
| name | state |      | name | pet  |
+------+-------+      +------+------+
| Ada  | AZ    |      | Deb  | dog  |
| Deb  | DE    |      | Deb  | duck |
+------+-------+      | Pat  | pig  |
                      +------+------+
```

Use the JOIN clause to join the two tables into one table:

```
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 |
+------+-------+------+------+
```

The resulting table only includes rows for Deb since she is present in both tables.

The left two columns are from the `states` table and the right two are from the `pets` table. The columns in the output can be referenced using the aliases `s.name`, `s.state`, `p.name`, and `p.pet`.

Breaking Down the JOIN Clause

```
states s INNER JOIN pets p ON s.name = p.name
```

Tables (`states`, `pets`)
: The tables we would like to combine.

Aliases (`s`, `p`)
: These are nicknames for the tables. This is optional, but recommended for simplicity. Without aliases, the `ON` clause could be written as `states.name = pets.name`.

Join Type (`INNER JOIN`)
: The `INNER` portion specifies that only matching rows should be returned. If only `JOIN` is written, then it defaults to an `INNER JOIN`. Other join types can be found in Table 9-2.

Join Condition (`ON s.name = p.name`)
: The condition that must be true in order for two rows to be considered matching. Equal (=) is the most common operator, but others can be used as well including not equal (`!=` or `<>`), `>`, `<`, `BETWEEN`, etc.

In addition to the `INNER JOIN`, Table 9-2 lists the various types of joins in SQL. The following query shows the general format for joining tables together:

```
SELECT *
FROM states s [JOIN_TYPE] pets p
     ON s.name = p.name;
```

Replace the bolded `[JOIN_TYPE]` portion with the keywords in the Keyword column to get the results shown in the Resulting Rows column. For the `CROSS JOIN` join type, exclude the `ON` clause to get the results shown in the table.

Table 9-2. Ways to join together tables

Keyword	Description	Resulting Rows
`JOIN`	Defaults to an `INNER JOIN`.	`nm  | st | nm  | pt`
		`-----+----+-----+------`
		`Deb | DE | Deb | dog`
		`Deb | DE | Deb | duck`
`INNER JOIN`	Returns the rows in common.	`nm  | st | nm  | pt`
		`-----+----+-----+------`
		`Deb | DE | Deb | dog`
		`Deb | DE | Deb | duck`
`LEFT JOIN`	Returns the rows in the left table and the matching rows in the other table.	`nm  | st | nm   | pt`
		`-----+----+------+------`
		`Ada | AZ | NULL | NULL`
		`Deb | DE | Deb  | dog`
		`Deb | DE | Deb  | duck`
`RIGHT JOIN`	Returns the rows in the right table and the matching rows in the other table.	`nm   | st   | nm  | pt`
		`------+------+-----+------`
		`Deb  | DE   | Deb | dog`
		`Deb  | DE   | Deb | duck`
		`NULL | NULL | Pat | pig`
`FULL OUTER JOIN`	Returns the rows in both tables.	`nm   | st   | nm   | pt`
		`------+------+------+------`
		`Ada  | AZ   | NULL | NULL`
		`Deb  | DE   | Deb  | dog`
		`Deb  | DE   | Deb  | duck`
		`NULL | NULL | Pat  | pig`
`CROSS JOIN`	Returns all combinations of rows in the two tables.	`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`

In addition to joining tables using the standard `JOIN ... ON ...` syntax, Table 9-3 lists others ways to join tables in SQL.

Table 9-3. Syntax to join together tables

Type	Description	Code
`JOIN ... ON ...` Syntax	Most common join syntax that works with `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL OUTER JOIN`, and `CROSS JOIN`.	`SELECT * FROM states s` **`INNER JOIN`** `pets p` **`ON`** `s.name = p.name;`
`USING` Shortcut	Use `USING` instead of the ON clause if the names of the columns that you are joining on match.	`SELECT * FROM states INNER JOIN pets` **`USING (name)`**`;`
`NATURAL JOIN` Shortcut	Use `NATURAL JOIN` instead of `INNER JOIN` if the names of all of the columns that you are joining on match.	`SELECT * FROM states` **`NATURAL JOIN`** `pets;`
Old Join Syntax	Return all the combinations of the rows in two tables. Equivalent to a `CROSS JOIN`.	`SELECT * FROM` **`states s, pets p`** `WHERE s.name = p.name;`

Type	Description	Code

Self Join	Use either the old join or new join syntax to return all the combinations of the rows in a table with itself.	`>SELECT * FROM states s1, states s2 WHERE s1.region = s2.region;`
		
		 `SELECT * FROM states s1 INNER JOIN states s2 WHERE s1.region = s2.region;`

The following sections describe the concepts in Tables 9-2 and 9-3 in detail.

Join Basics and INNER JOIN

This section walks through how a join works conceptually, as well as the basic join syntax using an `INNER JOIN`.

Join basics

You can think of joining tables in two steps:

1. Display all combinations of rows in the tables.
2. Filter on the rows that have matching values.

Here are two tables we'd like to join:

```
-- states                 -- pets
+------+-------+        +------+------+
| name | state |        | name | pet  |
+------+-------+        +------+------+
| Ada  | AZ    |        | Deb  | dog  |
| Deb  | DE    |        | Deb  | duck |
+------+-------+        | Pat  | pig  |
                        +------+------+
```

Step 1: Display all combinations of rows.

By listing the table names in the `FROM` clause, all possible combinations of rows from the two tables are returned.

```
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  |
+------+-------+------+------+
```

The `FROM states, pets` syntax is an older way of doing a join in SQL. A more modern way of doing the same thing is using a `CROSS JOIN`.

Step 2: Filter on the rows that have matching names.

You likely don't want to display all combinations of rows in the two tables, but rather only situations where the `name` column of both tables match.

```
SELECT *
FROM states s, pets p
WHERE s.name = p.name;

+------+-------+------+------+
| name | state | name | pet  |
+------+-------+------+------+
| Deb  | DE    | Deb  | dog  |
| Deb  | DE    | Deb  | duck |
+------+-------+------+------+
```

The row `Deb/DE` is listed twice because it matched two `Deb` values in the `pets` table.

The preceding code is equivalent to an `INNER JOIN`.

NOTE

The two-step process described previously is purely conceptual. Databases will rarely do a cross join when executing a join, but instead do it in a more optimized way.

However, thinking in these conceptual terms will help you correctly write join queries and understand their results.

INNER JOIN

The most common way to join together two tables is using an `INNER JOIN`, which returns rows that are in both tables.

Use `INNER JOIN` *to only return people in both tables*

```
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 |
+------+-------+------+------+
```

Join together more than two tables

This can be done by including additional sets of the `JOIN .. ON ..` keywords:

```
SELECT *
FROM states s
     INNER JOIN pets p
        ON s.name = p.name
     INNER JOIN lunch l
        ON s.name = l.name;
```

Join on more than one column

This can be done by including additional conditions within the `ON` clause. Imagine you want to join the following tables on both `name` and `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, and FULL OUTER JOIN

Use `LEFT JOIN`, `RIGHT JOIN`, and `FULL OUTER JOIN` to bring together rows from two tables, including ones that don't appear in both tables.

LEFT JOIN

Use `LEFT JOIN` to return all people in the `states` table. People in the `states` table that are not in the `pets` table get returned with `NULL` values.

```
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 |
+------+-------+------+------+
```

A `LEFT JOIN` is equivalent to a `LEFT OUTER JOIN`.

RIGHT JOIN

Use `RIGHT JOIN` to return all people in the `pets` table. People in the `pets` table that are not in the `states` table get returned with `NULL` values.

```
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  |
+------+-------+------+------+
```

A `RIGHT JOIN` is equivalent to a `RIGHT OUTER JOIN`.

SQLite does not support `RIGHT JOIN`.

TIP

The `LEFT JOIN` is much more common than the `RIGHT JOIN`. If a `RIGHT JOIN` is needed, swap the two tables within the `FROM` clause and do a `LEFT JOIN` instead.

FULL OUTER JOIN

Use `FULL OUTER JOIN` to return all people in both the `states` and `pets` tables. Missing values from both tables are returned with `NULL` values.

```
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  |
+------+-------+-----+------+
```

A `FULL OUTER JOIN` is equivalent to a `FULL JOIN`.

MySQL and *SQLite* do not support `FULL OUTER JOIN`.

USING and NATURAL JOIN

When joining tables together, to save on typing, you can use the `USING` or `NATURAL JOIN` shortcutsinstead of the standard `JOIN .. ON ..` syntax.

USING

MySQL, *Oracle*, *PostgreSQL*, and *SQLite* support the `USING` clause.

You can use the `USING` shortcut in place of the `ON` clause to join on two columns of the exact same name. The join must be an equi-join (= in the `ON` clause) to use `USING`.

```
-- ON clause
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 |
+------+------+-----+------+

-- Equivalent USING shortcut
SELECT *
FROM states INNER JOIN pets
     USING (name);

+------+------+------+
| name | state | pet  |
+------+------+------+
| Deb  | DE    | dog  |
| Deb  | DE    | duck |
+------+------+------+
```

The difference between the two queries is that the first query returns four columns including `s.name` and `p.name`, while the second query returns three columns because the two `name` columns get merged together as one and is simply called `name`.

NATURAL JOIN

MySQL, *Oracle*, *PostgreSQL*, and *SQLite* support a `NATURAL JOIN`.

You can use the `NATURAL JOIN` shortcut in place of the `INNER JOIN .. ON ..` syntax to join two tables based on all columns of the exact same name. The join must be an equi-join (= in the `ON` clause) to use a `NATURAL JOIN`.

```
-- 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 |
+------+------+-----+------+-----+------+
-- Equivalent NATURAL JOIN shortcut
SELECT *
FROM states_ages NATURAL JOIN pets_ages;

+------+-----+------+------+
| name | age  | state | pet  |
+------+-----+------+------+
| Ada  |   30 | AZ    | ant  |
+------+-----+------+------+
```

The difference between the two queries is that the first query returns six columns including `s.name`, `s.age`, `p.name`, and `p.age`, while the second query returns four columns because the duplicate `name` and `age` columns get merged together and are simply called `name` and `age`.

WARNING

Be careful when using a `NATURAL JOIN`. It saves quite a bit of typing, but can do an unexpected join if a column of a matching name is added or removed from a table. It is better to use for quick queries versus production code.

CROSS JOIN and Self Join

Another way of joining tables together is by displaying all combinations of the rows in two tables. This can be done with a `CROSS JOIN`. If this is done on a table with itself, it is called a *self join*. A self join is useful when you want to compare rows within the same table.

CROSS JOIN

Use `CROSS JOIN` to return all combinations of the rows in two tables. It is equivalent to listing out the tables in the `FROM` clause (which is sometimes referred to as "old join syntax").

```
-- CROSS JOIN
SELECT *
```

```
FROM **states CROSS JOIN pets**;

-- Equivalent table list
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  |
+------+-------+------+------+
```

Once all combinations are listed out, you can choose to filter on the results by adding a `WHERE` clause to return fewer rows based on what you're looking for.

Self join

You can join a table with itself using a self join. There are typically two steps to a self join:

1. Display all combinations of the rows in a table with itself.
2. Filter on the resulting rows based on some criteria.

The following are two examples of self joins in practice.

Here is a table of employees and their managers:

```
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 |
+------+--------+----------+--------+
```

Example 1: Return a list of employees and their managers.

```
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   |
+----------+----------+
```

Example 2: Match each employee with another employee in their department.

```
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    |
+------+----------+----------+
```

NOTE

The preceding query has duplicate rows (monica/lisa and lisa/monica). To remove the duplicates and return just four rows instead of eight, you can add the line:

```
AND e.emp_name < matching_emp.emp_name
```

to the WHERE clause to only return rows where the first name is before the second name alphabetically. Here is the output without duplicates:

```
+------+----------+----------+
| dept | emp_name | emp_name |
+------+----------+----------+
| tech | lisa     | monica   |
| data | nancy    | olivia   |
| data | nancy    | penny    |
| data | olivia   | penny    |
+------+----------+----------+
```

Union Operators

Use the UNION keyword to combine the results of two or more SELECT statements. The difference between a JOIN and a UNION is that JOIN links together multiple tables within a single query, whereas UNION stacks the results of multiple queries:

```
-- JOIN example
SELECT *
FROM birthdays b JOIN candles c
     ON b.name = c.name;

-- UNION example
SELECT * FROM writers
UNION
SELECT * FROM artists;
```

Figure 9-1 shows the difference between the results of a JOIN and a UNION, based on the preceding code.

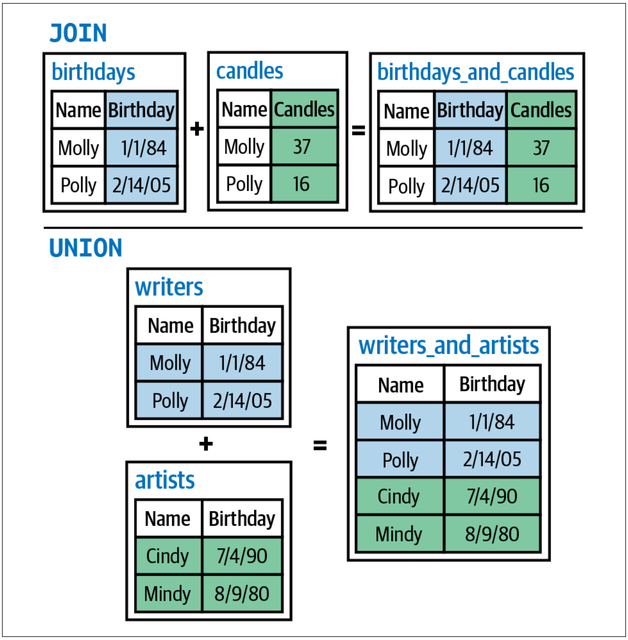


Figure 9-1. `JOIN` versus `UNION`

There are three ways to combine the rows of two tables together. These are also known as *union operators*:

`UNION`
: Combines the results of multiple statements.

`EXCEPT` *(`MINUS` in Oracle)*
: Returns the results minus another set of results.

`INTERSECT`
: Returns overlapping results.

UNION

The `UNION` keyword combines the results of two or more `SELECT` statements into one output.

Here are two tables we'd like to combine:

```
-- 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         |
+---------+---------+------------+
```

Use UNION to combine the two tables and eliminate any duplicate rows:

```
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     |
+---------+---------+
```

Note that tahani/england appears in both the staff and residents tables. However, it only shows up as one row in the result set because UNION removes duplicate rows from the output.

Which Queries Can You Union Together?

When doing a UNION on two queries, some characteristics of the queries must match and others do not have to match.

Number of Columns: MUST MATCH
: When you union together two queries, you must specify the same number of columns in both queries.

Column Names: DO NOT HAVE TO MATCH
: The column names of the two queries do not need to match to do a UNION. The column names used in the first SELECT statement in a UNION query become the names of the output columns.

Data Types: MUST MATCH
: The data types of the two queries need to match to do a UNION. If they do not match, you can use the CAST function to cast them into the same data type before doing a UNION.

UNION ALL

Use UNION ALL to combine the two tables and preserve duplicate rows:

```
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     |
+---------+---------+
```

TIP

If you know with certainty that no duplicate rows are possible, use `UNION ALL` to improve performance. `UNION` does an additional sort behind the scenes to identify the duplicates.

UNION with other clauses

You can also include other clauses when using a `UNION`, such as `WHERE`, `JOIN`, etc. However, only one `ORDER BY` clause is allowed for the whole query, and it should be at the very end.

Filter out null values and sort the results of a `UNION` query:

```
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 with more than two tables

You can union together more than two tables by including additional UNION clauses.

Combine the rows of more than two tables:

```
SELECT name, origin
FROM staff

UNION

SELECT name, country
FROM residents

UNION

SELECT name, country
FROM pets;
```

TIP

UNION is typically used to combine results from multiple tables. If you are combining results from a single table, it is better to write a single query instead and use the appropriate WHERE clause, CASE statement, etc.

EXCEPT and INTERSECT

In addition to using a UNION to combine the rows of multiple tables, you can use EXCEPT and INTERSECT to combine the rows in different ways.

EXCEPT

Use EXCEPT to "subtract" the results of one query from another query.

Return the staff members that are not residents:

```
SELECT name FROM staff
EXCEPT
SELECT name FROM residents;

+---------+
| name    |
+---------+
| michael |
| janet   |
+---------+
```

MySQL does not support `EXCEPT`. Instead, you can use the `NOT IN` keywords as a workaround:

```
SELECT name
FROM staff
WHERE name NOT IN (SELECT name FROM residents);
```

Oracle uses `MINUS` instead of `EXCEPT`.

PostgreSQL also supports `EXCEPT ALL`, which does not remove duplicates. `EXCEPT` removes all occurrences of a value, while `EXCEPT ALL` removes specific instances.

INTERSECT

Use `INTERSECT` to find the rows in common between two queries.

Return the staff members that are residents as well:

```
SELECT name, origin FROM staff
INTERSECT
SELECT name, country FROM residents;

+---------+---------+
| name    | origin  |
+---------+---------+
| tahani  | england |
+---------+---------+
```

MySQL does not support `INTERSECT`. Instead, you can use an `INNER JOIN` as a workaround:

```
SELECT s.name, s.origin
FROM staff s INNER JOIN residents r
     ON s.name = r.name;
```

PostgreSQL also supports `INTERSECT ALL`, which preserves duplicate values.

Union Operators: Order of Evaluation

When writing a statement with multiple union operators (`UNION`, `EXCEPT`, `INTERSECT`), use parentheses to specify the order in which the operations should occur.

```
SELECT * FROM staff
EXCEPT
(SELECT * FROM residents
UNION
SELECT * FROM pets);
```

Unless otherwise specified, union operators are performed in top-down order, except that `INTERSECT` takes precedence over `UNION` and `EXCEPT`.

Common Table Expressions

A *common table expression (CTE)* is a temporary result set. In other words, it temporarily saves the output of a query for you to write other queries that reference it.

You can spot a CTE when you see the `WITH` keyword. There are two types of CTEs:

Nonrecursive CTE
: A query for other queries to reference (see "CTEs Versus Subqueries" on page 293).

Recursive CTE
: A query that references itself (see "Recursive CTEs" on page 295).

NOTE

Nonrecursive CTEs are a lot more common than recursive CTEs. Most of the time, if someone mentions a CTE, they are referring to a nonrecursive CTE.

Here is an example of a nonrecursive CTE in practice:

```
-- Query the results of 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;
```

Here is an example of a recursive CTE in practice:

```
-- Generate the numbers 1 through 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;
```

In *MySQL* and *PostgreSQL*, the `RECURSIVE` keyword is required. In *Oracle* and *SQL Server*, the `RECURSIVE` keyword must be left out. *SQLite* works with either syntax.

In *Oracle*, you may see older code that uses the `CONNECT BY` syntax for recursive queries, but CTEs are much more common these days.

CTEs Versus Subqueries

Both CTEs and subqueries allow you to write a query, and then write another query that references the first query. This section describes the difference between the two approaches.

Imagine your goal is to find the department that has the largest average salary. This can be done in two steps: write a query that returns the average salary for each department; use a CTE or subquery to write a query around the first query to return the department with the largest average salary.

Step 1. Query that finds the average salary for each department

```
SELECT dept, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept;

+-------+------------+
| dept  | avg_salary |
+-------+------------+
| mktg  |      78000 |
| sales |      61000 |
| tech  |      83000 |
+-------+------------+
```

Step 2. CTE and subquery that find the department with the largest average salary using the preceding query

```
-- CTE approach
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;

-- Equivalent subquery approach
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 |
+------+------------+
```

The `LIMIT` clause syntax differs by software. Replace `LIMIT 1` with `ROWNUM = 1` in *Oracle* and `TOP 1` in *SQL Server*. More details can be found in The LIMIT Clause section in Chapter 4.

Advantages of a CTE Versus a Subquery

There are a few advantages to using a CTE instead of a subquery.

Multiple References
: Once a CTE is defined, you can reference it by name multiple times within the `SELECT` queries that follow:

```
WITH my_cte AS (...)

SELECT * FROM my_cte WHERE id > 10
UNION
SELECT * FROM my_cte WHERE score > 90;
```

With a subquery, you would need to write out the full subquery each time.

Multiple Tables
: CTE syntax is more readable when working with multiple tables because you can list all the CTEs up front:

```
WITH my_cte1 AS (...),
     my_cte2 AS (...)

SELECT *
```

```
        FROM my_cte1 m1
             INNER JOIN my_cte2 m2
             ON m1.id = m2.id;
```

With subqueries, the subqueries would be scattered throughout the overall query.

CTEs are *not supported* in older SQL software, which is why subqueries are still commonly used.

Recursive CTEs

This section walks through two practical situations where a recursive CTE would be useful.

Fill in missing rows in a sequence of data

The following table includes dates and prices. Note that the `date` column is missing data for the second and fifth of the month.

```
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 |
+------------+--------+
```

Fill in the dates with a two-step process:

1. Use a recursive CTE to generate a sequence of dates.
2. Join the sequence of dates with the original table.

NOTE

The following code runs in *MySQL*. Table 9-4 has the syntax for each RDBMS.

Step 1: Use a recursive CTE to generate a sequence of dates called `my_dates`.

The `my_dates` table starts with the date `2021-03-01`, and adds on the next date again and again, up until the date `2021-03-06`:

```
-- MySQL syntax
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 |
+------------+
```

Step 2: Left join the recursive CTE with the original table.

```
-- MySQL syntax
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 |
+------------+--------+
```

Step 3 (Optional): Fill in the null values with the previous day's price.

Replace the `SELECT` clause (`SELECT d.dt, s.price`) with:

```
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 |
+------------+--------+
```

There are syntax differences for each RDBMS.

Here is the general syntax for generating a date column. The bolded portions differ by RDBMS, and the software-specific code is listed in Table 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;
```

Table 9-4. Generating a date column in each RDBMS

RDBMS	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'`
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( '2021-03-01')`	`DATE(dt, '1 day')`	`'2021-03-06'`

Return all the parents of a child row

The following table includes the roles of various family members. The rightmost column includes the `id` of a person's parent.

```
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 |
+------+---------+----------+-----------+
```

NOTE

The following code runs in *MySQL*. Table 9-5 has the syntax for each RDBMS.

You can list each person's parents and grandparents with a recursive CTE:

```
-- MySQL syntax
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    |
+------+----------+---------------------------+
```

In the preceding code (also known as a *hierarchical query*), `my_cte` contains two statements that are unioned together:

- The first `SELECT` statement is the starting point. The rows where the `parent_id` is `NULL` are treated as the tree roots.
- The second `SELECT` statement defines the recursive link between the parent and child rows. The children of each tree root are returned and tacked on to the `lineage` column until the full lineage is spelled out.

There are syntax differences for each RDBMS.

Here is the general syntax for listing all the parents. The bolded portions differ by RDBMS, and the software-specific code is listed in Table 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;
```

Table 9-5. Listing all the parents in each RDBMS

RDBMS	WITH	NAME	LINEAGE
MySQL	`WITH RECURSIVE`	`name`	`CONCAT(mc.lineage, ' > ', ft.name)`

RDBMS	WITH	NAME	LINEAGE
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`

CHAPTER 10

How Do I...?

This chapter is intended to be a quick reference for frequently asked SQL questions that combine multiple concepts:

- Find the rows containing duplicate values
- Select rows with the max value for another column
- Concatenate text from multiple fields into a single field
- Find all tables containing a specific column name
- Update a table where the ID matches another table

Find the Rows Containing Duplicate Values

The following table lists seven types of teas and the temperatures they should be steeped at. Note that there are two sets of duplicate `tea`/`temperature` values, which are in bold.

```
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 |
+----+--------+-------------+
```

This section covers two different scenarios:

- Return all unique `tea/temperature` combinations
- Return only the rows with duplicate `tea/temperature` values

Return All Unique Combinations

To exclude duplicate values and return only the unique rows of a table, use the `DISTINCT` keyword.

```
SELECT DISTINCT tea, temperature
FROM teas;

+--------+-------------+
| tea    | temperature |
+--------+-------------+
| green  |         170 |
| black  |         200 |
| herbal |         212 |
| herbal |         210 |
| oolong |         185 |
+--------+-------------+
```

Potential extensions

To return the number of unique rows in a table, use the `COUNT` and `DISTINCT` keywords together. More details can be found in the DISTINCT section in Chapter 4.

Return Only the Rows with Duplicate Values

The following query identifies the rows in the table with duplicate values.

```
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 |
+----+--------+-------------+
```

Explanation

The bulk of the work happens in the `dup_rows` query. All of the `tea`/`temperature` combinations are counted, and then only the combinations that occur more than once are kept with the `HAVING` clause. This is what `dup_rows` looks like:

```
+--------+-------------+----------+
| tea    | temperature | num_rows |
+--------+-------------+----------+
| black  |         200 |        2 |
| herbal |         212 |        2 |
+--------+-------------+----------+
```

The purpose of the `JOIN` in the second half of the query is to pull the `id` column back into the final output.

Keywords in the query

- **`WITH dup_rows`** is the start of a common table expression, which allows you to work with multiple `SELECT` statements within a single query.
- **`HAVING COUNT(*) > 1`** uses the `HAVING` clause, which allows you to filter on an aggregation like `COUNT()`.
- **`teas t INNER JOIN dup_rows d`** uses an `INNER JOIN`, which allows you to bring together the `teas` table and the `dup_rows` query.

Potential extensions

To delete particular duplicate rows from a table, use a `DELETE` statement. More details can be found in Chapter 5

Select Rows with the Max Value for Another Column

The following table lists employees and the number of sales they've made. You want to return each employee's most recent number of sales, which are in bold.

```
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 |
+------+----------+------------+-------+
```

Solution

The following query returns the number of sales that each employee made on their most recent sale date (aka each employee's largest date value).

```
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 |
+------+----------+-------------+-------+
```

Explanation

The key to this problem is to break it down into two parts. The first goal is to identify the most recent sale date for each employee. This is what the output of the subquery `r` looks like:

```
+----------+-------------+
| employee | recent_date |
+----------+-------------+
| Emma     | 2021-08-07  |
| Jack     | 2021-08-05  |
+----------+-------------+
```

The second goal is to pull the `id` and `sales` columns back into the final output, which is done using the `JOIN` in the second half of the query.

Keywords in the query

- **`GROUP BY employee`** uses the `GROUP BY` clause, which splits up the table by `employee` and finds the **`MAX(date)`** for each employee.
- **`r INNER JOIN sales s`** uses an `INNER JOIN`, which allows you to bring together the `r` subquery and the `sales` table.

Potential extensions

An alternative to the `GROUP BY` solution is to use a window function (`OVER … PARTITION BY …`) with a `FIRST_VALUE` function, which would return the same results. More details can be found in the "Window Functions" on page 250 section in Chapter 8.

Concatenate Text from Multiple Fields into a Single Field

This section covers two different scenarios:

- Concatenate text from fields *in a single row* into a single value
- Concatenate text from fields *in multiple rows* into a single value

Concatenate Text from Fields in a Single Row

The following table has two columns, and you want to concatenate them into one column.

```
+----+---------+        +-----------+
| id | name    |        | id_name   |
+----+---------+        +-----------+
|  1 | Boots   |  --->  | 1_Boots   |
|  2 | Pumpkin |        | 2_Pumpkin |
|  3 | Tiger   |        | 3_Tiger   |
+----+---------+        +-----------+
```

Use the `CONCAT` function or the concatenation operator (`||`) to bring together the values:

```
-- MySQL, PostgreSQL, and SQL Server
SELECT CONCAT(id, '_', name) AS id_name
FROM my_table;

-- Oracle, PostgreSQL, and SQLite
SELECT id || '_' || name AS id_name
FROM my_table;

+-----------+
| id_name   |
+-----------+
| 1_Boots   |
| 2_Pumpkin |
| 3_Tiger   |
+-----------+
```

Potential extensions

Chapter 7, *Operators and Functions*, covers other ways to work with string values in addition to `CONCAT`, including:

- Finding the length of a string
- Finding words in a string
- Extracting text from a string

Concatenate Text from Fields in Multiple Rows

The following table lists the calories burned by each person. You want to concatenate the calories for each person into a single row.

```
+------+----------+      +------+----------+
| name | calories |      | name | calories |
+------+----------+      +------+----------+
| ally |       80 | ---> | ally | 80,75,90 |
| ally |       75 |      | jess | 100,92   |
| ally |       90 |      +------+----------+
| jess |      100 |
| jess |       92 |
+------+----------+
```

Use a function like `GROUP_CONCAT`, `LISTAGG`, `ARRAY_AGG`, or `STRING_AGG` to create the list.

```
SELECT name,
       GROUP_CONCAT(calories) AS calories_list
FROM workouts
GROUP BY name;

+------+---------------+
| name | calories_list |
+------+---------------+
| ally | 80,75,90      |
| jess | 100,92        |
+------+---------------+
```

This code works in *MySQL* and *SQLite*. Replace `GROUP_CONCAT(calories)` with the following in other RDBMSs:

Oracle

```
LISTAGG(calories, ',')
```

PostgreSQL

```
ARRAY_AGG(calories)
```

SQL Server

```
STRING_AGG(calories, ',')
```

Potential extensions

The aggregate rows into a single value or list section in Chapter 8 includes details on how to use other separators besides the comma (`,`), how to sort the values, and how to return unique values.

Find All Tables Containing a Specific Column Name

Imagine you have a database with many tables. You want to quickly find all tables that contain a column name with the word `city` in it.

Solution

In most RDBMSs, there is a special table that contains all table names and column names. Table 10-1 shows how to query that table in each RDBMS.

The last line of each code block is optional. You can include it if you want to narrow down the results for a particular database or user. If excluded, all tables will be returned.

Table 10-1. Find all tables containing a specific column name

RDBMS	Code

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';`

The output will display all column names that contain the term `city` along with the tables they are in:

```
+------------+-------------+
| TABLE_NAME | COLUMN_NAME |
+------------+-------------+
| customers  | city        |
| employees  | city        |
| locations  | metro_city  |
+------------+-------------+
```

NOTE

SQLite does not have a table that contains all column names. Instead, you can manually show all tables and then view the column names within each table:

```
.tables
pragma table_info(my_table);
```

Potential extensions

Chapter 5, *Creating, Updating, and Deleting*, covers more ways to interact with databases and tables, including:

- Viewing existing databases
- Viewing existing tables
- Viewing the columns of a table

Chapter 7, *Operators and Functions*, covers more ways to search for text in addition to `LIKE`, including:

- `=` to search for an exact match
- `IN` to search for multiple terms
- Regular expressions to search for a pattern

Update a Table Where the ID Matches Another Table

Imagine you have two tables: `products` and `deals`. You'd like to update the names in the `deals` table with the names of items in the `products` table that have a matching `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
+------+-------------+
```

Solution

Use an `UPDATE` statement to modify values in a table using the `UPDATE ... SET ...` syntax. Table 10-2 shows how to do this in each RDBMS.

Table 10-2. Update a table where the ID matches another table

RDBMS	Code

MySQL	`UPDATE deals d,`
	`       products p`
	`SET    d.name = p.name`
	`WHERE  d.id = p.id;`

RDBMS	Code
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;`

The `deals` table is now updated with the names from the `products` table:

```
SELECT * FROM deals;

+------+-------------------+
| id   | name              |
+------+-------------------+
|  102 | MIDI keyboard     |
|  103 | Mother's day card |
+------+-------------------+
```

WARNING

Once the `UPDATE` statement is executed, the results cannot be undone. The exception is if you start a transaction before executing the `UPDATE` statement.

Potential extensions

Chapter 5, *Creating, Updating, and Deleting*, covers more ways to modify tables, including:

- Updating a column of data
- Updating rows of data
- Updating rows of data with the results of a query
- Adding a column to a table

Final Words

This book covers the most popular concepts and keywords in SQL, but we've only scratched the surface. SQL can be used to perform many tasks, using a variety of different approaches. I encourage you to keep on learning and exploring.

You may have noticed that SQL syntax varies widely by RDBMS. Writing SQL code requires a lot of practice, patience, and looking up syntax. I hope you've found this pocket guide to be helpful for doing so.

Index

About the Author

Alice Zhao is a data scientist who is passionate about teaching and making complex things easy to understand. She has taught numerous courses in SQL, Python, and R as a senior data scientist at Metis and as a cofounder of Best Fit Analytics. Her highly rated technical tutorials on YouTube are known for being practical, entertaining, and visually engaging.

She writes about analytics and pop culture on her blog, *A Dash of Data*. Her work has been featured in *Huffington Post*, *Thrillist*, and *Working Mother*. She has spoken at a variety of conferences including Strata in New York City and ODSC in San Francisco on topics ranging from natural language processing to data visualization. She has her MS in analytics and BS in electrical engineering, both from Northwestern University.

Colophon

The animal on the cover of *SQL Pocket Guide* is an Alpine salamander (*salamandra atra*). Most commonly found in ravines high up in the Alps (upwards of 1,000m), the Alpine salamander stands out for its unusual ability to handle cold weather. The shiny black creatures prefer shady, moist places and the cracks and gaps in stone walls. It feeds on worms, spiders, snails, and small insect larvae.

Unlike other salamanders, the Alpine salamander gives birth to fully formed juveniles. A pregnancy lasts two years, but at even higher altitudes (1,400–1,700m), it can last up to three years. The species is generally protected throughout the Alps, but climate change has more recently impacted their preferred habitat of rocky, not-too-dry landscapes.

Many of the animals on O’Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Lydekker's Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.