SQL & Databases

The Complete Beginner's Guide

From "what is a database" to writing joins, subqueries, and transactions with confidence no prior experience required.

ZolaXTech | Student Resource Library

Prepared July 2026

How to Use This Guide?

This guide assumes no prior experience with databases or programming. It builds one idea on top of the last, using a single running example a small online bookstore so that everything you learn in one chapter is put to direct use in the next. You do not need to install anything to begin: Chapter 4 points you to a free, no signup website where you can start typing real SQL within a minute of reading it.

Every SQL example in this guide is shown in a shaded code box, and every chapter's ideas are reinforced through a hands on capstone project in the final chapter. Two appendices at the end a quick reference cheat sheet and a glossary are designed for you to return to long after your first read through, whenever you need a fast reminder.

Table of Contents

Chapter 1: What Is a Database?
Chapter 2: What Is SQL, and Why Learn It?
Chapter 3: Types of Databases and Popular Database Systems
Chapter 4: Getting Started Installing and Accessing a Database
Chapter 5: Understanding Tables, Rows, Columns, and Data Types
Chapter 6: Creating Databases and Tables
Chapter 7: Inserting Data into Tables
Chapter 8: Retrieving Data with SELECT
Chapter 9: Filtering Data with WHERE and Operators
Chapter 10: Sorting and Limiting Results
Chapter 11: Aggregate Functions and GROUP BY
Chapter 12: Updating and Deleting Data
Chapter 13: Understanding Keys and Table Relationships
Chapter 14: Joining Tables
Chapter 15: Subqueries
Chapter 16: Database Normalization Basics
Chapter 17: Transactions and Data Integrity
Chapter 18: Views and Indexes
Chapter 19: Common Mistakes and Best Practices
Chapter 20: Practice Project Build the Bookstore Database from Scratch
Appendix A: SQL Quick Reference Cheat Sheet
Appendix B: Glossary of Key Terms
Appendix C: Where to Practice and Learn More

Chapter 1: What Is a Database?

A database is an organized collection of information stored electronically so it can be easily accessed, managed, and updated. Think of it as a digital filing cabinet: instead of loose paper folders, information is stored in structured tables, and instead of a person searching through drawers, a computer program searches through the data in a fraction of a second.

Almost every application you use daily is backed by a database. When you log into an online store, your account details, order history, and the product catalog are all stored in a database. When a bank shows your balance, that number is read from a database. Databases are the backbone of virtually all modern software.

Why not just use a spreadsheet?

A spreadsheet works fine for a small, personal list, but it breaks down quickly as data grows or as many people need to use it at once. Databases solve problems that spreadsheets cannot handle well:

- Scale: databases can efficiently store and search millions or billions of records; spreadsheets slow down and become unmanageable after a few thousand rows.
- Multiple users: databases are built to let many people read and write data at the same time without corrupting it or overwriting each other's changes.
- Data integrity: databases can enforce rules (for example, an order cannot reference a customer that does not exist), keeping data accurate and consistent.
- Complex queries: databases can quickly answer questions that combine many conditions across multiple related tables, something spreadsheets struggle with.

What is a relational database?

A relational database organizes data into tables (also called relations), where each table represents one type of thing for example, a table of customers, a table of products, and a table of orders. Each table is made up of rows (individual records) and columns (the attributes or fields describing each record). Tables can be linked to one another through shared values, which is what makes the data "relational." This guide focuses entirely on relational databases, because they are the most widely used type in business, and SQL is the language used to work with them.

> **KEY IDEA**
> A table is like a single sheet in a spreadsheet: rows are records, columns are fields. A database is a collection of these tables, linked together in meaningful ways.

Chapter 2: What Is SQL, and Why Learn It?

SQL stands for Structured Query Language. It is the standard language used to communicate with relational databases to create tables, insert data, retrieve information, update records, and remove data that is no longer needed. SQL is not a general purpose programming language like Python or JavaScript; it is a specialized language designed for one job: working with structured data.

SQL is pronounced either as individual letters ("S Q L") or as the word "sequel" both are common, and there is no single correct way.

Why SQL is worth learning

- It is everywhere: SQL is used by nearly every major database system, so the skills transfer across companies, industries, and job titles.
- It is a career essential: roles in data analysis, business intelligence, software development, product management, and finance all regularly require reading or writing SQL.
- It answers real questions fast: SQL lets you ask precise questions of your data "which customers spent over $500 last month" and get an answer in seconds.
- It has stayed relevant for decades: first developed in the 1970s, SQL remains the dominant way to interact with structured data today, even as newer tools have emerged around it.

The main categories of SQL commands

Category	Purpose	Example commands
DDL (Data Definition Language)	Defines and modifies the structure of the database	CREATE, ALTER, DROP
DML (Data Manipulation Language)	Adds, changes, or removes the actual data	INSERT, UPDATE, DELETE
DQL (Data Query Language)	Retrieves data from the database	SELECT
DCL (Data Control Language)	Controls who is allowed to access or change data	GRANT, REVOKE
TCL (Transaction Control Language)	Manages groups of changes as a single unit	COMMIT, ROLLBACK

This guide walks through each of these categories in order, building your knowledge step by step using one consistent example database, so that by the end you will be comfortable writing real SQL from scratch.

Chapter 3: Types of Databases and Popular Database Systems

Before writing any SQL, it helps to know the landscape of database systems you are likely to encounter. All of the systems below understand SQL, though each has small differences in syntax and extra features.

System	Best known for	Good for beginners?
SQLite	A lightweight database stored in a single file, with no separate server required	Yes the easiest way to start; used throughout this guide
MySQL	One of the most popular open source databases, widely used for websites and web apps	Yes free, well documented, huge community
PostgreSQL	A powerful, standards compliant open source database favoured for complex applications	Yes slightly more advanced features than MySQL
Microsoft SQL Server	A commercial database common in corporate and enterprise environments	Possible, though typically used in Windows centric business settings
Oracle Database	A long established enterprise database used by large organizations	Less common for beginners; mostly encountered in enterprise jobs

It is worth noting that relational databases like the ones above are sometimes called SQL databases, in contrast to NoSQL databases (such as MongoDB or Redis), which store data in different formats such as documents or key value pairs and are optimized for different use cases. NoSQL is outside the scope of this guide, but it is a useful term to recognize.

> **WHY THIS GUIDE USES SQLITE**
> SQLite requires no installation of a server, no configuration, and no login credentials it is simply a file. This removes every barrier to getting started, and everything you learn transfers almost directly to MySQL, PostgreSQL, and other systems, since the SQL itself is nearly identical.

Chapter 4: Getting Started Installing and Accessing a Database

You do not need to install anything complicated to start practicing SQL today. Below are three beginner friendly options, from easiest to most involved.

Option 1: Practice in the browser (no installation)

The fastest way to start writing real SQL right now is to use a free online SQL sandbox. Two good options are SQLite Online (sqliteonline.com) and DB Fiddle (dbfiddle.uk). Both let you type SQL into a browser window and see results instantly, with no account or installation needed. This is the recommended starting point for anyone brand new to SQL.

Option 2: Install DB Browser for SQLite

DB Browser for SQLite is a free, visual application (available for Windows, macOS, and Linux) that lets you create SQLite database files, run SQL commands, and browse the resulting tables in a spreadsheet like grid. It is an excellent tool for learning because it shows you exactly what your commands produced, side by side with the SQL you wrote. Download it from sqlitebrowser.org.

Option 3: Install a full database server (MySQL or PostgreSQL)

If you already know you want to work toward a specific system used at your job or in a course, you can install MySQL Community Server (dev.mysql.com) or PostgreSQL (postgresql.org) directly on your computer, along with a graphical client such as MySQL Workbench or pgAdmin. This route takes more setup time, so it is best attempted after you are already comfortable with basic SQL from Options 1 or 2.

> **RECOMMENDATION FOR BEGINNERS**
> Start with Option 1 or 2. Every example of code in this guide will run without any changes in a plain SQLite environment, so you can follow along immediately and build real confidence before ever touching a server installation.

Chapter 5: Understanding Tables, Rows, Columns, and Data Types

Before creating a real database, it helps to see the finished shape of one. Throughout this guide, we will build a small online bookstore database made up of five related tables:

- **authors** the writers whose books are sold
- **books** the products available for sale, each linked to one author
- **customers** the people who create accounts and place orders
- **orders** a record of each purchase a customer makes
- **order items** the individual books included within each order

Every table is made up of columns, and every column has a data type that defines what kind of value it can hold. Choosing the right data type keeps your data accurate and helps the database store and search it efficiently.

Data type	Stores	Example value
INTEGER	Whole numbers	42
REAL / FLOAT / DECIMAL	Numbers with decimal points	19.99
TEXT / VARCHAR(n)	Text of any length (VARCHAR often has a maximum length n)	"Jane Austen"
DATE	A calendar date	2026 07 22
DATETIME / TIMESTAMP	A date and time together	2026 07 22 14:30:00
BOOLEAN	True or false values (some systems use 0/1 instead)	TRUE / FALSE

Different database systems name these types slightly differently for example, MySQL uses VARCHAR (255) for limited length text, while PostgreSQL commonly uses TEXT for unlimited text. SQLite is unusually flexible and will accept most type names loosely. Do not worry about memorizing every variant; the concepts here are what matter, and you can always look up exact type names for your specific system when needed.

Chapter 6: Creating Databases and Tables

The CREATE TABLE statement defines a new table: its name, its columns, each column's data type, and any rules (constraints) that values in that column must follow. Let's create the first table in our bookstore database, authors.

```
CREATE TABLE authors (
    author_id     INTEGER PRIMARY KEY,
    first_name    TEXT NOT NULL,
    last_name     TEXT NOT NULL,
    country       TEXT
);
```

Reading this line by line: CREATE TABLE authors tells the database to make a new table named authors. Each line inside the parentheses defines one column: its name, its data type, and any constraints. A semicolon ends the statement SQL statements should always end with a semicolon.

Common constraints

Constraint	What it does
PRIMARY KEY	Uniquely identifies each row in the table; no two rows can share the same value, and it cannot be empty
NOT NULL	Requires that a value must always be provided for this column
UNIQUE	Ensures no two rows can have the same value in this column (but, unlike a primary key, a table can have several UNIQUE columns)
DEFAULT	Automatically fills in a value if none is provided when a row is inserted
FOREIGN KEY	Links a column to the primary key of another table, enforcing that the value must already exist there
CHECK	Requires that values in the column satisfy a specific condition, such as price >= 0

Now let's create the books table, which links to authors through a foreign key:

```
CREATE TABLE books (
    book_id          INTEGER PRIMARY KEY,
    title             TEXT NOT NULL,
    author_id         INTEGER NOT NULL,
    genre             TEXT,
    price             DECIMAL(6,2) NOT NULL CHECK (price >= 0),
    stock_quantity    INTEGER NOT NULL DEFAULT 0,
    published_year    INTEGER,
    FOREIGN KEY (author_id) REFERENCES authors (author_id)
);
```

The FOREIGN KEY line at the bottom tells the database that every value in the books. author_id column must correspond to a real author_id that already exists in the authors table. This is the mechanism that keeps related tables consistent with one another it is impossible to add a book for an author who does not exist.

Let's finish the schema with the three remaining tables:

```
CREATE TABLE customers (
    customer_id    INTEGER PRIMARY KEY,
    first_name     TEXT NOT NULL,
    last_name      TEXT NOT NULL,
    email          TEXT NOT NULL UNIQUE,
    city           TEXT
);

CREATE TABLE orders (
    order_id       INTEGER PRIMARY KEY,
    customer_id    INTEGER NOT NULL,
    order_date     DATE NOT NULL DEFAULT CURRENT_DATE,
    status         TEXT NOT NULL DEFAULT 'pending',
    FOREIGN KEY (customer_id) REFERENCES customers (customer_id)
);

CREATE TABLE order_items (
    order_item_id    INTEGER PRIMARY KEY,
    order_id         INTEGER NOT NULL,
    book_id          INTEGER NOT NULL,
    quantity         INTEGER NOT NULL CHECK (quantity > 0),
    unit_price       DECIMAL(6,2) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES orders (order_id),
    FOREIGN KEY (book_id) REFERENCES books (book_id)
);
```

> **TIP**
> Notice that order_items exists purely to connect orders and books. This kind of table is called a junction (or linking) table, and it is the standard way to represent the fact that one order can contain many books, and one book can appear in many orders. Chapter 13 explains this pattern in more depth.

Removing or modifying a table

If you need to change a table's structure later, use ALTER TABLE. If you need to permanently delete a table and all of its data, use DROP TABLE. Both are shown below use them carefully, since DROP TABLE cannot be undone.

```
   Add a new column to an existing table
ALTER TABLE customers ADD COLUMN phone TEXT;

   Permanently delete a table and all its data
DROP TABLE order_items;
```

Chapter 7: Inserting Data into Tables

With the tables created, the next step is to add data using INSERT INTO. You specify the table name, the columns you are filling in, and the values, in matching order.

```
INSERT INTO authors (author_id, first_name, last_name, country)
VALUES (1, 'Chimamanda', 'Ngozi Adichie', 'Nigeria');

INSERT INTO authors (author_id, first_name, last_name, country)
VALUES (2, 'Haruki', 'Murakami', 'Japan');
```

Text values are wrapped in single quotes ('like this'); numbers are written without quotes. You can insert several rows in one statement by separating each set of values with a comma, which is more efficient than writing a separate INSERT for every row:

```
INSERT INTO books (book_id, title, author_id, genre, price, stock_quantity, published_year)
VALUES
    (1, 'Half of a Yellow Sun', 1, 'Historical Fiction', 14.99, 32, 2006),
    (2, 'Americanah', 1, 'Fiction', 12.50, 18, 2013),
    (3, 'Norwegian Wood', 2, 'Literary Fiction', 11.75, 25, 1987),
    (4, 'Kafka on the Shore', 2, 'Magical Realism', 13.20, 15, 2002);
```

```
INSERT INTO customers (customer_id, first_name, last_name, email, city)
VALUES
    (1, 'Meron', 'Tesfaye', 'meron.t@example.com', 'Addis Ababa'),
    (2, 'Daniel', 'Okafor', 'daniel.o@example.com', 'Lagos'),
    (3, 'Sara', 'Bekele', 'sara.b@example.com', 'Addis Ababa');

INSERT INTO orders (order_id, customer_id, order_date, status)
VALUES
    (1, 1, '2026 06 01', 'completed'),
    (2, 2, '2026 06 03', 'completed'),
    (3, 1, '2026 06 15', 'pending');

INSERT INTO order_items (order_item_id, order_id, book_id, quantity, unit_price)
VALUES
    (1, 1, 1, 1, 14.99),
    (2, 1, 3, 2, 11.75),
    (3, 2, 2, 1, 12.50),
    (4, 3, 4, 1, 13.20);
```

If a column has a DEFAULT value (like status in the orders table) or allows NULL, you can leave it out of the INSERT statement entirely and the database will fill it in automatically or leave it empty.

> **COMMON ERROR**

> A frequent beginner mistake is forgetting the single quotes around text values, or mismatching the number of columns and values. If SQL reports a syntax error, the very first thing to check is that every text value is quoted and that your column list and value list line up exactly.

Chapter 8: Retrieving Data with SELECT

SELECT is the command you will use more than any other in SQL it retrieves data from one or more tables. The most basic form retrieves every column and every row from a table:

```
SELECT * FROM books;
```

The asterisk (*) means "all columns." While convenient for exploring a table, it is best practice in real applications to list only the columns you actually need, which is both clearer and more efficient:

```
SELECT title, price, stock_quantity FROM books;
```

You can rename columns in the output using AS, which is useful for making results more readable:

```
SELECT title AS book_title, price AS price_usd FROM books;
```

To avoid duplicate values in the results, use DISTINCT for example, to see the unique list of genres currently in stock:

```
SELECT DISTINCT genre FROM books;
```

You can also perform simple calculations directly inside a SELECT statement. For example, to see the total value of stock on hand for each book (price multiplied by quantity):

```
SELECT title, price, stock_quantity, price * stock_quantity AS stock_value
FROM books;
```

Chapter 9: Filtering Data with WHERE and Operators

The WHERE clause narrows a query down to only the rows that meet a specific condition. It is placed after the table name in a SELECT statement.

```
SELECT title, price
FROM books
WHERE price < 13.00;
```

Comparison operators

Operator	Meaning
=	Equal to
<> or !=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Combining conditions with AND, OR, and NOT

```
   Books under $15 AND published after 2000
SELECT title, price, published_year
FROM books
WHERE price < 15.00 AND published_year > 2000;

   Books in either of two genres
SELECT title, genre
FROM books
WHERE genre = 'Fiction' OR genre = 'Literary Fiction';
```

Useful filtering keywords

Keyword	Purpose	Example
IN	Matches any value in a list	WHERE genre IN ('Fiction', 'Literary Fiction')
BETWEEN	Matches a range (inclusive)	WHERE price BETWEEN 10 AND 15
LIKE	Matches a text pattern using wildcards	WHERE title LIKE 'The%'
IS NULL	Matches rows where a column has no value	WHERE country IS NULL

Keyword	Purpose	Example
IS NOT NULL	Matches rows where a column does have a value	WHERE country IS NOT NULL

The LIKE operator uses two wildcard symbols: the percent sign (%) matches any sequence of characters (including none), and the underscore (_) matches exactly one character. For example, WHERE title LIKE '%Wood%' would match any title containing the word "Wood" anywhere within it.

> **TIP**
> Never use = to check for NULL values always use IS NULL or IS NOT NULL. In SQL, NULL represents "unknown," and a comparison like column = NULL does not behave the way most beginners expect; it will not match anything, even rows that are actually NULL.

Chapter 10: Sorting and Limiting Results

ORDER BY sorts your results by one or more columns. By default, sorting is ascending (smallest to largest, or A to Z); add DESC for descending order.

```
   Cheapest books first
SELECT title, price FROM books ORDER BY price ASC;

   Most expensive books first
SELECT title, price FROM books ORDER BY price DESC;

   Sort by genre, then by price within each genre
SELECT title, genre, price
FROM books
ORDER BY genre ASC, price DESC;
```

LIMIT restricts how many rows are returned, which is useful for previewing large result sets or answering questions like "what are the 3 most expensive books."

```
SELECT title, price
FROM books
ORDER BY price DESC
LIMIT 3;
```

Note: Microsoft SQL Server uses a slightly different syntax, TOP 3, placed right after SELECT instead of LIMIT at the end. This is one of the small dialect differences between database systems mentioned in Chapter 3.

Chapter 11: Aggregate Functions and GROUP BY

Aggregate functions perform a calculation across many rows and return a single summary value. They are essential for answering business questions like "how many customers do we have" or "what is our average order value."

Function	Returns
COUNT()	The number of rows (or non null values in a specific column)
SUM()	The total of a numeric column
AVG()	The average of a numeric column
MIN()	The smallest value in a column
MAX()	The largest value in a column

```
   How many books are in the catalog?
SELECT COUNT(*) AS total_books FROM books;

   What is the average book price?
SELECT AVG(price) AS average_price FROM books;

   What is the total value of all stock on hand?
SELECT SUM(price * stock_quantity) AS total_stock_value FROM books;
```

Aggregate functions become far more powerful when combined with GROUP BY, which splits your rows into groups (for example, one group per genre) and calculates the aggregate separately for each group.

```
   Number of books and average price, per genre
SELECT genre, COUNT(*) AS num_books, AVG(price) AS avg_price
FROM books
GROUP BY genre;
```

To filter groups after aggregation, use HAVING instead of WHERE WHERE filters individual rows before grouping, while HAVING filters the grouped results afterward.

```
   Only genres with more than one book
SELECT genre, COUNT(*) AS num_books
FROM books
GROUP BY genre
HAVING COUNT(*) > 1;
```

> **REMEMBER THE ORDER**

A SELECT statement using every clause is written in this order: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT. The database actually processes them in a different internal order, but writing them in this sequence is required syntax.

Chapter 12: Updating and Deleting Data

UPDATE changes existing values in a table. It is critical to always include a WHERE clause without one, the update applies to every single row in the table.

```
   Correct: updates only order 3
UPDATE orders
SET status = 'completed'
WHERE order_id = 3;

   Increase the price of one specific book by $1.00
UPDATE books
SET price = price + 1.00
WHERE book_id = 2;
```

DELETE removes rows from a table. The same warning applies: without a WHERE clause, DELETE removes every row in the table.

```
   Remove a single customer record
DELETE FROM customers
WHERE customer_id = 3;
```

TRUNCATE TABLE (available in most systems other than SQLite, which uses DELETE FROM table_name instead) removes all rows from a table at once, and is faster than DELETE for clearing an entire table, though it cannot be filtered with WHERE.

> **CRITICAL SAFETY HABIT**
> Before running any UPDATE or DELETE, first run the exact same WHERE condition as a SELECT statement to confirm precisely which rows will be affected. For example, before UPDATE orders SET status = 'completed' WHERE order_id = 3, run SELECT * FROM orders WHERE order_id = 3 first. This one habit prevents the vast majority of costly, accidental data changes.

Chapter 13: Understanding Keys and Table Relationships

Relationships between tables are what make relational databases powerful instead of repeating a customer's full details on every single order, an order simply stores a reference (the customer's ID) back to the customers table. This avoids duplication and keeps data consistent.

Primary key

A primary key uniquely identifies each row in a table. No two rows can share a primary key value, and it can never be empty. In our schema, author_id, book_id, customer_id, order_id, and order_item_id are all primary keys of their respective tables.

Foreign key

A foreign key is a column in one table that refers to the primary key of another table, creating a link between them. In our schema, books.author_id is a foreign key referencing authors.author_id, meaning every book must belong to an author that actually exists in the authors table.

The three types of relationships

Relationship type	Description	Example in our schema
One to many	One row in Table A can relate to many rows in Table B, but each row in Table B relates to only one row in Table A	One author can have many books; each book has exactly one author
One to one	Each row in Table A relates to exactly one row in Table B, and vice versa	Less common; e.g., a customer's table and a separate customer_profile_details table sharing one record each
Many to many	Many rows in Table A can relate to many rows in Table B	One order can include many books, and one book can appear in many orders

A many to many relationships cannot be represented directly with a single foreign key it requires a junction table (also called a linking or bridge table) sitting between the two tables, holding a foreign key to each. This is exactly the role order_items plays in our schema: it connects orders and books, and also stores details specific to that combination, like quantity and unit_price at the time of purchase.

Chapter 14: Joining Tables

A JOIN combines rows from two or more tables based on a related column, most often a foreign key matching a primary key. Joins are what let you ask questions that span multiple tables for example, "show me each order along with the customer's name," which requires combining data from both orders and customers.

INNER JOIN

An INNER JOIN returns only the rows that have a match in both tables. This is the most commonly used join.

```
SELECT orders.order_id, customers.first_name, customers.last_name, orders.order_date
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

The ON clause tells the database how the two tables relate: match each order to the customer whose customer_id equals the order's customer_id. Table names can be shortened using aliases to make longer queries easier to read:

```
SELECT o.order_id, c.first_name, c.last_name, o.order_date
FROM orders AS o
INNER JOIN customers AS c ON o.customer_id = c.customer_id;
```

LEFT JOIN

A LEFT JOIN returns every row from the left (first) table, along with matching rows from the right table and if there is no match, the right table's columns simply appear as NULL. This is useful for finding records that do not have a related match, such as books that have never been ordered.

```
SELECT b.title, oi.order_item_id
FROM books AS b
LEFT JOIN order_items AS oi ON b.book_id = oi.book_id
WHERE oi.order_item_id IS NULL;
```

RIGHT JOIN and FULL OUTER JOIN

A RIGHT JOIN is the mirror image of a LEFT JOIN: it keeps every row from the right table, with unmatched left table columns showing as NULL. A FULL OUTER JOIN keeps every row from both tables, matching where possible and filling in NULL where there is no match on either side. Note that SQLite and MySQL (before recent versions) do not support FULL OUTER JOIN directly; PostgreSQL and SQL Server do.

Joining three or more tables

Joins can be chained to pull together information spread across several tables. Here is a complete query showing exactly what each customer ordered, combining four tables at once:

```
SELECT
    c.first_name || ' ' || c.last_name AS customer_name,
```

```
    o.order_date,
    b.title,
    oi.quantity,
    oi.unit_price
FROM orders AS o
INNER JOIN customers AS c ON o.customer_id = c.customer_id
INNER JOIN order_items AS oi ON o.order_id = oi.order_id
INNER JOIN books AS b ON oi.book_id = b.book_id
ORDER BY o.order_date;
```

TIP

The || symbol used above concatenates (joins together) text values in SQLite and PostgreSQL. MySQL uses a function instead: CONCAT (c.first_name, ' ', c.last_name). SQL Server uses the + symbol. This is another small but common dialect difference to watch for.

Chapter 15: Subqueries

A subquery is a SELECT statement nested inside another SQL statement. It lets you use the result of one query as an input to another, which is useful when a question cannot be answered in a single, direct step.

Subquery in a WHERE clause

For example, to find all books priced above the average price of all books, you first need to calculate that average, then compare each book to it:

```
SELECT title, price
FROM books
WHERE price > (SELECT AVG(price) FROM books);
```

The inner query (SELECT AVG(price) FROM books) runs first and produces a single value, which the outer query then uses for comparison.

Subquery with IN

To find all customers who have placed at least one order, you can use a subquery that returns a list of customer IDs:

```
SELECT first_name, last_name
FROM customers
WHERE customer_id IN (SELECT DISTINCT customer_id FROM orders);
```

Subquery in the FROM clause

Subqueries can also act as a temporary, named result set that the outer query then treats like a regular table:

```
SELECT genre, avg_price
FROM (
    SELECT genre, AVG(price) AS avg_price
    FROM books
    GROUP BY genre
) AS genre_averages
WHERE avg_price > 12.00;
```

As a general guideline, many queries that use subqueries can also be written using JOINs, and vice versa. Subqueries are often easier to read when a question naturally breaks into a clear step by step logic ("first calculate this, then compare to that"), while JOINs are usually preferred when you need to display columns from multiple tables side by side in the same result.

Chapter 16: Database Normalization Basics

Normalization is the process of organizing tables to reduce data duplication and prevent inconsistencies. A well normalized database stores each piece of information in exactly one place, so updating it in one location keeps the entire database accurate.

Consider what would happen if, instead of a separate authors table, we stored the author's full name and country directly inside every row of the books table. If that author wrote ten books, their country would be repeated ten times. If the author later moved to a new country, we would have to remember to update all ten rows and if we missed even one, the database would contain contradictory information. Splitting authors into its own table, referenced by author_id, solves this: the country is stored once, and every book simply points to it.

The first three normal forms

Normal form	Rule	Example fix
1NF (First Normal Form)	Each column holds a single, indivisible value no lists or repeated groups within one field	Instead of one column holding 'Fiction, Historical', split genres into a separate related table if a book can have several
2NF (Second Normal Form)	Builds on 1NF; every non key column must depend on the entire primary key, not just part of it (relevant mainly for tables with multi column keys)	In order_items, quantity and unit_price depend on the specific order_item_id, not just order_id or book_id alone
3NF (Third Normal Form)	Builds on 2NF; non key columns must not depend on other non-key columns	Don't store both city and country_code_for_city in the same table if country_code_for_city is fully determined by city store that relationship separately instead

In practice, most well designed business databases aim for at least 3NF. However, normalization is not an absolute rule to follow at all costs in some cases, especially for reporting or performance reasons, database designers deliberately introduce some duplication (called denormalization) to make certain queries faster. As a beginner, the key habit to build is simply asking, for every piece of data: "is this stored in exactly one place, or could it become inconsistent if updated?"

Chapter 17: Transactions and Data Integrity

A transaction is a group of one or more SQL statements that are executed together as a single, all or nothing unit. Transactions matter whenever multiple related changes must succeed or fail together if only some of them applied, the data would be left in an inconsistent state.

Consider placing an order in our bookstore: it requires inserting a new row into orders and one or more rows into order_items, and typically also reducing stock_quantity in books. If the database crashed partway through after creating the order but before recording the items the database would show an order with nothing in it. Transactions prevent this.

```
BEGIN TRANSACTION;

INSERT INTO orders (order_id, customer_id, order_date, status)
VALUES (4, 2, '2026 07 20', 'pending');

INSERT INTO order_items (order_item_id, order_id, book_id, quantity, unit_price)
VALUES (5, 4, 1, 1, 14.99);

UPDATE books SET stock_quantity = stock_quantity   1 WHERE book_id = 1;

COMMIT;
```

COMMIT permanently saves all the changes made since the transaction began. If something goes wrong before COMMIT is reached, you can instead run ROLLBACK, which undoes every change made since BEGIN TRANSACTION, as if none of it had happened.

The ACID properties

Reliable transactions are commonly described using the acronym ACID:

- Atomicity: all statements in the transaction succeed together, or none of them are applied at all.
- Consistency: a transaction moves the database from one valid state to another, never leaving it in a broken, contradictory state.
- Isolation: transactions running at the same time do not interfere with or see each other's incomplete changes.
- Durability: once a transaction is committed, the changes are permanently saved, even if the system crashes immediately afterward.

Chapter 18: Views and Indexes

Views

A view is a saved SELECT query that behaves like a virtual table. It does not store data itself every time you query it, it runs the underlying SELECT statement fresh. Views are useful for simplifying complex or frequently repeated queries.

```
CREATE VIEW order_summary AS
SELECT
    o.order_id,
    c.first_name || ' ' || c.last_name AS customer_name,
    o.order_date,
    SUM(oi.quantity * oi.unit_price) AS order_total
FROM orders AS o
INNER JOIN customers AS c ON o.customer_id = c.customer_id
INNER JOIN order_items AS oi ON o.order_id = oi.order_id
GROUP BY o.order_id, customer_name, o.order_date;
```

Once created, the view can be queried exactly like a table, hiding the underlying complexity from anyone using it:

```
SELECT * FROM order_summary WHERE order_total > 20;
```

Indexes

An index is a special data structure that dramatically speeds up how quickly the database can find rows matching a condition, similar to how the index at the back of a textbook lets you find a topic without reading every page. Without an index, the database must scan every single row in a table to find matches; with an index on the right column, it can jump almost directly to them.

```
CREATE INDEX idx_books_genre ON books (genre);
```

Indexes are typically added to columns that are frequently used in WHERE clauses, JOIN conditions, or ORDER BY clauses, especially on large tables. Primary key columns are automatically indexed by nearly every database system. Indexes are not free, however they take up additional storage space and slightly slow down INSERT, UPDATE, and DELETE operations, since the index itself must also be updated. As a beginner, the key takeaway is simply to know that indexes exist and what problem they solve; deciding exactly which columns to index in a large, real world system is a more advanced topic usually handled as performance needs arise.

Chapter 19: Common Mistakes and Best Practices

- Always test destructive statements first. Before running UPDATE or DELETE, run the same WHERE condition as a SELECT to confirm exactly which rows will be affected.
- Never forget the WHERE clause on UPDATE or DELETE unless you genuinely intend to affect every row in the table.
- Use meaningful, consistent naming. Table and column names should be lowercase, use underscores instead of spaces (order_date, not Order Date), and be descriptive rather than abbreviated.
- Always define a primary key on every table. It ensures every row can be uniquely identified and referenced from other tables.
- Avoid SELECT * in real applications. Explicitly listing the columns, you need makes queries clearer, more efficient, and less likely to break if the table structure changes later.
- Format your SQL for readability. Put each major clause (SELECT, FROM, WHERE, GROUP BY, ORDER BY) on its own line, and indent consistently, especially in longer queries with joins.
- Comment your SQL. Use for a single line comment or /* ... */ for a multi-line comment to explain why a query does something non obvious.
- Back up data before large changes. Especially outside of a transaction, it is good practice to export or copy important data before running bulk updates or deletions.
- Watch for NULL carefully. Remember that comparisons with NULL require IS NULL or IS NOT NULL, and that aggregate functions like AVG () and SUM () ignore NULL values rather than treating them as zero.
- Learn to read error messages slowly. SQL error messages usually point to the exact location of a problem a missing comma, an unclosed quote, or a misspelled column name so read them carefully before assuming the worst.

Chapter 20: Practice Project Build the Bookstore Database from Scratch

This final chapter brings every concept in this guide together into one hands on project. Open your chosen SQL environment from Chapter 4 and follow along, typing each step yourself rather than copying and pasting typing builds muscle memory far more effectively than reading alone.

1. Create all five tables (authors, books, customers, orders, order_items) using the CREATE TABLE statements from Chapter 6, in that order, since later tables reference earlier ones through foreign keys.
2. Insert the sample data from Chapter 7 into each table.
3. Write a query that lists every book together with its author's full name, sorted from most to least expensive. (Hint: this requires an INNER JOIN between books and authors.)
4. Write a query that shows the total amount spent by each customer, using SUM (), GROUP BY, and a JOIN across orders, order_items, and customers.
5. Write a query that finds any book that has never appeared in an order, using a LEFT JOIN and a WHERE ... IS NULL check, as shown in Chapter 14.
6. Add a new column, loyalty_points, to the customers table using ALTER TABLE, then set every existing customer's value to 0 using UPDATE.
7. Wrap the process of placing a brand new order one INSERT into orders, one or more INSERTs into order_items, and an UPDATE reducing the relevant book's stock_quantity inside a single transaction using BEGIN TRANSACTION and COMMIT, exactly as shown in Chapter 17.
8. Create a view named customer_order_history that shows every order alongside the customer's name and the order's total value, then query that view to find every order over $20.

If you can complete all eight steps without referring back to earlier chapters, you have covered the full core of beginner to intermediate SQL: schema design, data manipulation, filtering, aggregation, joins, subqueries, transactions, and views. From here, the best way to keep improving is to find a real dataset that interests you sports statistics, movie ratings, personal finances and practice asking it real questions in SQL.

Appendix A: SQL Quick Reference Cheat Sheet

A fast lookup for the syntax patterns used throughout this guide. Replace table_name, col, and condition with your own values.

Task	Syntax pattern
View all data	SELECT * FROM table_name;
View specific columns	SELECT col1, col2 FROM table_name;
Filter rows	SELECT * FROM table_name WHERE condition;
Sort results	SELECT * FROM table_name ORDER BY col ASC|DESC;
Limit results	SELECT * FROM table_name LIMIT 10;
Remove duplicates	SELECT DISTINCT col FROM table_name;
Count rows	SELECT COUNT(*) FROM table_name;
Sum / average / min / max	SELECT SUM(col), AVG(col), MIN(col), MAX(col) FROM table_name;
Group and summarize	SELECT col, COUNT(*) FROM table_name GROUP BY col;
Filter groups	... GROUP BY col HAVING COUNT(*) > 1;
Create a table	CREATE TABLE t (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
Add a column	ALTER TABLE t ADD COLUMN new_col TEXT;
Delete a table	DROP TABLE t;
Insert a row	INSERT INTO t (col1, col2) VALUES (val1, val2);
Update rows	UPDATE t SET col = value WHERE condition;
Delete rows	DELETE FROM t WHERE condition;
Inner join	SELECT * FROM a INNER JOIN b ON a.id = b.a_id;
Left join	SELECT * FROM a LEFT JOIN b ON a.id = b.a_id;
Subquery	SELECT * FROM t WHERE col > (SELECT AVG(col) FROM t);
Start / save / undo a transaction	BEGIN TRANSACTION; ... COMMIT; / ROLLBACK;
Create a view	CREATE VIEW v AS SELECT ...;
Create an index	CREATE INDEX idx_name ON t (col);

Appendix B: Glossary of Key Terms

Term	Definition
Aggregate function	A function such as COUNT, SUM, AVG, MIN, or MAX that summarizes many rows into a single value.
ACID	Atomicity, Consistency, Isolation, Durability the four properties that make database transactions reliable.
Constraint	A rule enforced on a column or table, such as NOT NULL, UNIQUE, or PRIMARY KEY, that keeps data valid.
DDL / DML / DQL	The sub categories of SQL commands that define structure (DDL), manipulate data (DML), and query data (DQL).
Foreign key	A column that references the primary key of another table, creating a link between the two.
Index	A data structure that speeds up searching for rows matching a given column value.
Join	An operation that combines rows from two or more tables based on a related column.
Junction table	A table used to represent a many to many relationship, holding foreign keys to both related tables.
NULL	A special marker representing an unknown or missing value not the same as zero or an empty string.
Normalization	The process of structuring tables to minimize duplicate data and prevent inconsistencies.
Primary key	A column (or set of columns) that uniquely identifies each row in a table.
Query	A request written in SQL asking the database to retrieve or manipulate data.
Relational database	A database that organizes data into related tables of rows and columns.
Schema	The overall structure of a database its tables, columns, data types, and relationships.
Subquery	A SELECT statement nested inside another SQL statement.
Transaction	A group of SQL statements executed together as a single all or nothing unit.
View	A saved SELECT query that can be queried as if it were a table, without storing data itself.

Appendix C: Where to Practice and Learn More

The single best way to get comfortable with SQL is to keep writing it. The free tools and resources below are good next steps beyond this guide.

Resource	Link	Why it's useful
SQLite Online	sqliteonline.com	Free, no signup browser sandbox the fastest way to practice everything in this guide
DB Fiddle	dbfiddle.uk	Browser based SQL playground supporting several different database systems
DB Browser for SQLite	sqlitebrowser.org	Free desktop app for creating and exploring SQLite database files visually
MySQL Community Server	dev.mysql.com/downloads	Free, official download of MySQL for a full local server installation
PostgreSQL	postgresql.org/download	Free, official download of PostgreSQL for a full local server installation
Mode SQL Tutorial	mode.com/sql tutorial	A well regarded free tutorial with real, downloadable practice datasets