SQL Course

Flavio Copes

Updated August 3, 2026

Contents

Learn relational database fundamentals, create and change tables, write useful queries, connect related data, and use SQL safely in applications.

What you'll learn: Design a small relational database and write safe SQL queries that create, read, update, and connect its data.

Database fundamentals

Databases, DBMSs, relational structure, schemas, and SQL.

What is a Database? And a DBMS?

Learn what a database and a DBMS are, and the key properties of a database management system, from being efficient and persistent to secure and shared.

A **database** is a collection of information carefully organized into a system.

The technology that in a computer system lets us **organize data** and **represent the information** that's essential for an information system is called **Database Management System**.

A DBMS is a software that encapsulates the data of a database, and provides us a centralized way to store it, retrieve it, edit it, persist it, and much more.

Let's define some of the main properties of a DBMS:

- **Efficient**: a database needs to provide the best performance to store and retrieve data.
- **Persistent**: data stored in the database is stored permanently. When the database software is terminated or the machine reboots, the data (excluding hardware failures) should still be there.
- **Privacy and security**: a database provides us the ability to privately and securely store data. It allows access to multiple users, and each user should be able to only access and edit the data it's allowed to. Some users might only be able to access data and not edit or delete it.
- **Shared access**: multiple users need to be able, with the proper permissions, to access shared data. Multiple applications can access the same database, too.
- **Big**: a database can handle huge amounts of data, and it can scale according to your needs, using various advanced techniques. That does not mean a database is just useful when you have a lot of data - it can be useful even with very few data entries, due to the properties listed above.

There are a lot of different DBMS on the market. Some similar to each other, some vastly different.

Relational DBMS, built on top of the relational model, are some of the most common ones you can use in the real world.

I'll soon talk in details about 3 popular Open Source Relational Database Management Systems: PostgreSQL, MySQL and SQLite.

If you're not sure which kind of database fits your project, I built a free database chooser that asks a few questions and gives you a recommendation.

Relational Databases

Relational Databases are the software implementation of the concepts expressed by the theory introduced by the Relational Model.

Relational Databases are the software implementation of the concepts expressed by the theory introduced by the Relational Model.

In a Relational Database, data is stored in **tables**.

Each table contains one or more **columns**, that hold data of a specific **type**, like strings, numbers and so on.

The set of a table and all the rules about its columns is called a **schema**.

Each table can define **constrains** upon the data that each row can contain.

Tables can reference each other, forming relationships, using **foreign keys**.

A **Database Management System** is the software that implements the database in a computer system.

Commonly, relational databases use the **SQL language** to let us give instructions to create a database, define its tables **schema**, fill tables with data, and finally query the data when needed.

Some examples of software implementing relational databases are PostgreSQL, MySQL, SQLite, Oracle and MicroSoft SQL Server.

If you're trying to pick between the most popular ones, I built a free Postgres vs SQLite vs MySQL comparison tool that shows their differences side by side.

The Relational Model

The Relational Model is the most popular of the logic data models listed in the Data Models post, and it's at the basis of SQL databases.

The Relational Model is the most popular of the logic data models, and it's at the basis of SQL databases.

The **Relational Model** is based on two simple concepts:

- **tables**
- **relations**

The relational model dates back to 1969 and the work of Edgar F. Codd, an English computer scientist. Although as computer programmers we're used to look with curiosity at new shiny things, a technology that can be central in everything about computers for 50 years is definitely worth studying.

The fact that the model is based upon tables makes it very intuitive to use, because we are used to using tables to organize things. Think about an Excel spreadsheet, for example.

With SQL-based databases, like PostgreSQL, Oracle, MySQL, SQLite and MS SQL Server, and

many others, the data analyzed using the ER Model can be modeled using the relational model and be almost immediately transformed into a SQL database format, which can be considered a real-world implementation of the relational model, but we'll talk about this in other posts.

In this post I want to talk about the theory and the concepts that the relational model is based upon, not expressed in mathematical terms but what it means in practice.

> If you're a student, you might find that what I write here is not what is written on your textbook, but maybe you can read it more easily to grasp the more formal concepts expressed in your learning material.

Tables

In a relational model, a **table** is a collection of items.

It is organized in rows and columns:

Name	Age
Flavio	36
Roger	7
Syd	6

Tuples

Each entry in the table is called a **tuple**. You can also use the terms **record** or **row**.

A tuple represents a row of the table, like this:

Flavio 36

Attributes

An attribute is one single item in the tuple.

In this example:

Flavio 36

”Flavio” is an attribute. 36 is another attribute.

Tuples are unique

Every tuple in the table is unique.

In the relational model, we can't have duplicate data, meaning every row in the table must be different in at least one attribute.

The relation key

The thing that ensures a tuple is unique is the **relation key**.

The key is one attribute that must **uniquely identify** a tuple.

If the relation key is a set of attributes, it must be **non-redundant**. This means that if we remove one of the attributes of the key, the key can't guarantee its uniqueness.

If more than one keys can be determined, one of those keys will be identified as the **primary key**.

Key integrity constraint

They key attribute(s) of any tuple in the table must **never be null**, and must **never repeat**.

Given a key, we must be able to point to a specific tuple/row without ambiguity.

The domain constraints

Every attribute has **rules about what value it can hold**.

If we decide to store numbers, we can't store strings, for example. And we might decide to not store strings longer than 10 characters for names.

We can also call this **type**.

The referential integrity constraint

If a table contains reference to a secondary table, or other tuples in the same table, then we must abide to rules that prevent the reference to break.

In particular, we must avoid breaking the reference by:

- avoiding deleting or editing the primary key of the record that we point to, in the other table.
- avoid inserting a new record with a non-existing key to point to in the other table.
- avoid changing the key of the record we point to, without ensuring that the new key exists in the other table.

A DBMS (Data Base Management System) will implement measures to help us implement referential integrity.

Introduction to SQL

An introduction to SQL, the Structured Query Language used to interact with a database management system like PostgreSQL, MySQL, SQLite, or Oracle.

After we introduced the most popular conceptual Data Model, Entity-Relationship (ER), and the most popular logic data model (the Relational Model), it's time to introduce SQL.

SQL (Structured Query Language) is a language we use to interact with a Database Management System (DBMS).

As the name suggests, it's not a programming language, but it was born as a querying language, and later evolved to an interface to doing more advanced operations with a database than just performing queries.

I said ”evolved”, but in reality SQL is always evolving. It's a standard that was first published in 1986, then updated in 1989, 1992, 1999, 2003, 2006, 2008, 2011, 2016 and as its latest version at

the time of writing, 2019.

SQL is implemented in many popular DBMS: PostgreSQL, MySQL, Oracle, SQLite, MicroSoft SQL Server, and many more. Each different database implements the standard, or a particular version of it, and adds custom features on top of it, to simplify creating queries or adding a specific functionality.

> Unless noted otherwise, every time I talk about SQL I talk about the SQL standard, not a particular implementation of it.

If you're writing SQL and it gets messy, I built a free SQL formatter that prettifies your queries in the browser.

SQL is a huge subject. I cover many of its topics in different blog posts, including:

SQL, creating a table SQL, adding data to a table SQL, how to use SELECT SQL, how to update a table structure SQL, how to update data SQL, how to delete data and tables SQL, Handling empty cells SQL, Unique and Primary keys SQL Views SQL Joins

Tables, rows, and columns

Connect the relational model to the tables, rows, and columns you work with in a SQL database.

A relational database stores related facts in **tables**. A table should describe one kind of thing, such as a customer, order, or product.

Consider this `orders` table:

id	customer_email	status	total
101	ada@example.com	paid	49.00
102	lin@example.com	pending	18.50

Each horizontal **row** is one order. Each **column** describes one attribute shared by every order. The value at one row and column is sometimes called a field or cell.

The schema exists before the data. It says that `id` identifies the row, `customer_email` contains text, `status` accepts the chosen representation, and `total` stores an exact amount.

That structure lets SQL ask useful questions:

```
SELECT id, total
FROM orders
WHERE status = 'paid';
```

The result contains order `101`, because that row makes the condition true.

A row needs a stable identity. Names and email addresses can change or repeat, so databases normally use a primary-key column such as `id`. The key lets an update target one order without depending on mutable business data.

Do not put several independent values into one cell. A comma-separated list of product IDs cannot be constrained or joined like real rows. Related facts belong in related tables, which we will model later.

Your action is to sketch a `books` table with four columns. Mark the primary key, choose one required column, and write two example rows. Then explain what one row represents.

When you need a database

Decide when structured, durable, searchable data needs a database and when a file or in-memory value is enough.

Use a database when data must survive restarts, be queried in several ways, enforce relationships, or support concurrent changes.

You do not need one for every program. A static configuration file or an in-memory array can be the simpler tool when the data is small and does not change independently.

Take a settings file:

```
{
  "theme": "dark",
  "language": "en"
}
```

This data is read at startup, changes rarely, and is always loaded whole. A JSON file is the right tool here. A database would add setup and complexity for zero benefit.

Now look at the four signals from the first sentence, one at a time.

Survive restarts. Data held in memory disappears when the process stops. If losing it is a problem, you need durable storage: a file at minimum, a database when the other signals show up too.

Queried in several ways. A file is easy to read top to bottom. But a question like ”all orders from March, grouped by customer, newest first” means loading everything and filtering in your own code. A database answers questions like that directly, and stays fast as the data grows.

Enforce relationships. When an order must always point at a real customer, a file leaves that rule to your code, in every code path, forever. A database enforces it once, at the storage layer.

Concurrent changes. This is the one that breaks file-based storage in practice. Picture two requests updating the same JSON file: both read it, both modify their own copy, both write it back. The second write silently erases the first one's change. That is a **lost update**, and you usually discover it weeks later as missing data. A database serializes writes and gives you transactions, so this class of bug disappears.

Start with the requirements: what must be stored, who changes it, and how it must be retrieved. One process, small data, whole-file reads: keep the file. Several writers, several questions, rules to enforce: that is a database.

Check your understanding: database fundamentals

Check databases, DBMS responsibilities, relational terminology, schemas, keys, and what SQL is for.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Tables and data

Types, constraints, and the commands that change stored data.

SQL, creating a table

Learn how to create a table in a SQL database with the CREATE TABLE command, defining column names and data types like INT, VARCHAR, DATE, and TEXT.

A database is composed by one or more tables.

Creating a table in SQL is done using the `CREATE TABLE` command.

At creation time you need to specify the table columns names, and the type of data they are going to hold.

SQL defines several kinds of data.

The most important and the ones you'll see more often are:

- `CHAR`
- `TEXT`
- `VARCHAR`
- `DATE`
- `TIME`
- `DATETIME`
- `TIMESTAMP`

Numeric types include

- `TINYINT` 1 byte
- `INT` 4 bytes
- `BIGINT` 8 bytes
- `SMALLINT` 2 bytes
- `DECIMAL`
- `FLOAT`

They all hold numbers. What changes is the size that this number can be.

A TINYINT goes goes 0 to 255. An INT from -2^31 to +2^31.

The bigger the size in bytes, the more space will be needed in storage.

This is the syntax to create a `people` table with 2 columns, one an integer and the other a variable length string:

```
CREATE TABLE people (
  age INT,
  name CHAR(20)
);
```

If you use an ORM, I built a free schema converter that turns `CREATE TABLE` statements into Prisma or Drizzle schemas (and back).

Choose column data types

Choose types that represent the data accurately instead of storing every value as text.

A column type describes the values the column is meant to hold. It also affects comparison, sorting, validation, storage, and calculations.

For an orders table, the choices might look like this:

```
CREATE TABLE orders (
  id INTEGER PRIMARY KEY,
  customer_email VARCHAR(320) NOT NULL,
  item_count INTEGER NOT NULL,
  total NUMERIC(10, 2) NOT NULL,
  paid BOOLEAN NOT NULL,
  created_at TIMESTAMP NOT NULL
);
```

Use integer types for counts and identifiers. Use an exact decimal or numeric type for money. Binary floating-point types are useful for measurements, but values such as `0.1` cannot always be represented exactly, which is a bad property for account balances.

Use date and time types when the database needs to order, compare, or calculate with time. Decide whether a value is an instant, a calendar date, or a local wall-clock time. Those are different facts.

Not every number is numeric data. Phone numbers, postal codes, and product codes can contain leading zeroes or punctuation and are not added together. Store them as text.

The declared type is only part of the contract. `INTEGER` still accepts `-4`, so a quantity also needs a `CHECK` constraint when negative values are invalid. A timestamp can be valid syntax while representing the wrong time zone.

Changing a type later can require scanning, converting, or locking a large table. Choose from real values and queries, not from the shortest-looking type name.

Exact names, ranges, timestamp behavior, and automatic conversions differ between SQLite, PostgreSQL, and MySQL. Check the documentation for the database you will run.

Your action is to choose types for a price, birth date, phone number, inventory count, and creation instant. Write one sentence explaining each choice.

SQL, adding data to a table

Learn how to insert data into a SQL database table using the INSERT INTO command, adding a single row or multiple rows at once with one statement.

Once you have a table, you can insert data into it.

Take this table:

```
CREATE TABLE people (
  age INT,
  name CHAR(20)
);
```

You can now start adding data into it with the `INSERT INTO` command:

```
INSERT INTO people VALUES (37, 'Flavio');
```

This form relies on the order of the columns in the table definition. The first value goes into `age`, the second into `name`, because that is the order they were declared in.

That works, but it's fragile. If someone later adds a column or reorders the table, the same statement inserts values into the wrong places, or fails.

My advice is to always name the columns you are inserting into:

```
INSERT INTO people (age, name) VALUES (37, 'Flavio');
```

Now the statement says exactly which value goes where. It keeps working even if the table gains new columns, and anyone reading it understands it without looking up the schema.

Naming columns also lets you skip some. Any column you leave out gets its default value, or `NULL` if it has no default:

```
INSERT INTO people (name) VALUES ('Syd');
```

Here `age` is `NULL` for Syd, because we didn't provide it.

Inserting multiple rows

You can insert multiple items separating each one with a comma:

```
INSERT INTO people VALUES (37, 'Flavio'), (8, 'Roger');
```

One statement with many rows is much faster than many statements with one row each, because the database does the parsing and the transaction work once. When you import hundreds of rows, batch them.

Verify the insert worked

The database confirms each insert with a count. PostgreSQL replies `INSERT 0 1` for one row, `INSERT 0 2` for two. MySQL says `Query OK, 1 row affected`.

To see the actual data, run a `SELECT`:

```
SELECT * FROM people;
```

```
 age |  name
-----+--------
  37 | Flavio
   8 | Roger
```

One common failure: inserting a value with the wrong type, like a string into `age`. The database rejects it with an error such as `invalid input syntax for type integer`. Read the error, fix the value, and run the statement again. Nothing was written, so there is nothing to clean up.

SQL, how to update data

Learn how to update data in a SQL table with the UPDATE command, and why the WHERE clause matters so you do not accidentally change every row.

Use `UPDATE` to change existing rows. Target a stable key when you intend to change one record:

```
UPDATE people
SET age = 8
WHERE id = 2;
```

The database evaluates the `WHERE` condition for every candidate row. Only rows where it is true are changed.

Preview the same condition first:

```
SELECT id, name, age
FROM people
WHERE id = 2;
```

After the update, check both the affected-row count reported by your client and the stored result. An affected count of zero can mean the identifier was wrong or another process already changed the row.

Without `WHERE`, every row is updated:

```
UPDATE people
SET age = 8;
```

That syntax is valid when you deliberately want a table-wide change. It is dangerous when one condition was forgotten.

NULL also matters. `WHERE email <> 'ada@example.com'` does not include rows whose email is NULL, because the comparison is unknown. Add `OR email IS NULL` when those rows should change too.

For an important update, start a transaction, run the update, inspect the rows, then commit. Roll back when the result is not what you intended.

Your action is to update one row by primary key, verify it, and then write—but do not run—a table-wide version. Point to the exact clause that changes the scope.

SQL, how to delete data and tables

Learn how to delete data from a SQL table with DELETE FROM and the WHERE clause, and how to remove an entire table using the DROP TABLE command.

Use `DELETE` to remove rows while keeping the table:

```
DELETE FROM people
WHERE id = 2;
```

Start with a `SELECT` using the same condition:

```
SELECT id, name
FROM people
WHERE id = 2;
```

This preview does not make the later delete safe by itself. Data can change between statements. For important work, use a transaction and inspect the affected-row count before committing.

This valid statement removes every row:

```
DELETE FROM people;
```

The table and its schema remain. `DROP TABLE` removes the table definition and its data:

```
DROP TABLE people;
```

Treat those as different operations. A migration may drop a retired table after a compatibility period. An application normally deletes particular rows.

Foreign keys can reject a delete when other rows still reference the target. A cascading foreign key

can remove those child rows automatically. Read the relationship before assuming either behavior.

NULL follows the same three-valued logic as other filters. `WHERE deleted_at < CURRENT_TIMESTAMP` ignores rows where `deleted_at` is NULL, which is often exactly what a cleanup wants.

A backup is the recovery plan for a committed accidental delete. A transaction rollback works only before commit.

Your action is to write a `SELECT` and matching `DELETE` for one disposable row. Run both inside a transaction, inspect the result, and roll it back.

SQL, Handling empty cells

Learn how to handle null and empty cells in a SQL database, and how to prevent them by adding the NOT NULL constraint to your table columns.

When we create a table in this way:

```
CREATE TABLE people (
  age INT,
  name CHAR(20)
);
```

SQL freely accepts empty values as records:

```
INSERT INTO people VALUES (null, null);
```

This might be a problem, because now we have a row with null values:

```
 age |  name
-----+--------
  37 | Flavio
   8 | Roger
     |
```

NULL is a special marker that means "no value here". It is not zero, and it is not an empty string. It records the absence of data: we never told the database that person's age or name.

NULL exists because real data is incomplete. A user skips a form field, or you will only know a value later. The database needs an honest way to store "unknown" instead of forcing you to invent a fake number.

Finding NULL values

Here is the classic mistake. This query returns zero rows, even though the table clearly contains a null:

```
SELECT * FROM people WHERE age = NULL;
```

In SQL, `NULL` is never equal to anything, not even to another `NULL`. The comparison does not evaluate to true, so every row is filtered out. Use the dedicated operators instead:

```
SELECT * FROM people WHERE age IS NULL;
SELECT * FROM people WHERE age IS NOT NULL;
```

`IS NULL` finds the rows with missing data. `IS NOT NULL` finds everything else.

When you display data, `COALESCE()` lets you swap a null for a fallback. It returns the first non-null argument it receives:

```
SELECT COALESCE(name, 'unknown') FROM people;
```

Preventing NULL values

To solve this, we can declare constraints on our table rows. `NOT NULL` prevents null values:

```
CREATE TABLE people (
  age INT NOT NULL,
  name CHAR(20) NOT NULL
);
```

If we try to execute this query again:

```
INSERT INTO people VALUES (null, null);
```

We'd get an error, like this:

```
ERROR:  null value in column "age" violates not-null constraint
DETAIL:  Failing row contains (null, null).
```

The error names the exact column that failed, and nothing was written to the table. Fix the statement to provide real values and run it again.

> Note that an empty string is a valid non-null value. A `NOT NULL` column of type `CHAR` still accepts `''`. If empty strings are also unacceptable for your data, you need a `CHECK` constraint on top.

SQL, how to update a table structure

Learn how to change a SQL table structure with the ALTER TABLE command, adding a new column with ADD COLUMN or removing one with DROP COLUMN.

Use `ALTER TABLE` to change an existing schema. Adding a nullable column is the simplest example:

```
ALTER TABLE people
ADD COLUMN born_year INTEGER;
```

Existing rows do not suddenly gain known birth years. Their new value is NULL:

```
id | name   | born_year
---+--------+----------
1  | Flavio | NULL
2  | Roger  | NULL
```

If the column must eventually be required, do not assume one statement is safe for a large live table. A compatible sequence is:

1. add the nullable column
2. deploy code that understands both schemas
3. backfill values in controlled batches
4. verify no NULL values remain
5. add the required constraint

This lets old and new application versions overlap during deployment.

Dropping a column destroys its stored values:

```
ALTER TABLE people
DROP COLUMN born_year;
```

Remove application reads and writes first. Keep the column through a compatibility window, then drop it in a later migration.

`ALTER TABLE` capabilities and locking behavior differ between database systems and versions. Some changes update metadata quickly. Others scan or rewrite the table and can block application work. Test the exact operation on representative data.

A down migration is not always a recovery plan. Recreating a dropped column does not restore its values. Back up important data and decide how to reverse the application deployment before changing production.

Your action is to add a nullable column to a practice table, inspect old and new rows, then write the verification query required before making it `NOT NULL`.

Use constraints to protect data

Put essential rules in the schema with NOT NULL, UNIQUE, CHECK, primary keys, and foreign keys.

Constraints put durable rules beside the data. Every application, migration, script, and import receives the same protection.

For example:

```
CREATE TABLE products (
  id INTEGER PRIMARY KEY,
  sku VARCHAR(40) NOT NULL UNIQUE,
  name VARCHAR(200) NOT NULL,
  price NUMERIC(10, 2) NOT NULL CHECK (price >= 0),
  stock INTEGER NOT NULL CHECK (stock >= 0)
);
```

Each constraint has one job:

- `NOT NULL` requires a known value
- `UNIQUE` prevents two rows from using the same SKU
- `CHECK` rejects values outside a business rule
- the primary key gives every row a stable identity
- a foreign key protects a relationship with another table

Now this insert should fail:

```
INSERT INTO products (id, sku, name, price, stock)
VALUES (1, 'BOOK-1', 'SQL Notes', -4.00, 10);
```

The failure is useful. It prevents an impossible price from becoming somebody else's debugging problem.

NULL needs special attention. A check such as `CHECK (price >= 0)` can evaluate to unknown when `price` is NULL, and SQL check constraints generally accept true or unknown. Pair the check with `NOT NULL` when absence is also invalid.

Application validation still matters because it can produce friendly messages before sending a query. The database constraint is the final boundary when several code paths write to the same table.

Adding a constraint to an existing table can fail because old rows already violate it. Find and repair those rows before enforcing the rule.

Your action is to attempt one valid and three invalid inserts against this table. Record which constraint rejects each invalid row.

Check your understanding: tables and data

Check table creation, types, inserts, updates, deletes, schema changes, NULL, and common data-changing mistakes.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Query data

Select, filter, sort, group, aggregate, and limit results.

SQL, how to use SELECT

Learn how to get data from a SQL table with the SELECT command, choosing columns, counting rows with COUNT, and filtering results with the WHERE clause.

You can get data out of tables using the `SELECT` command.

Get all rows and columns:

```
SELECT * FROM people;
```

```
 age |  name
-----+--------
  37 | Flavio
   8 | Roger
```

`*` means "every column". It's handy while exploring, but in application code my advice is to list the columns you need. The query keeps returning the same shape when the table changes, and you don't transfer data you never use.

Get only the `name` column:

```
SELECT name FROM people;
```

```
  name
--------
 Flavio
 Roger
```

You can ask for several columns, separated by commas, and rename them in the output with `AS`:

```
SELECT name AS person, age FROM people;
```

The result columns follow the order you list them, not the order in the table definition.

Count the items in the table:

```
SELECT COUNT(*) from people;
```

```
 count
-------
     2
```

By the way, once your tables grow, queries with `WHERE` clauses can get slow without the right index. I built a free index advisor that suggests the `CREATE INDEX` statement for the shape of your query.

You can filter rows in a table adding the `WHERE` clause:

```
SELECT age FROM people WHERE name='Flavio';
```

```
 age
-----
  37
```

Notice the single quotes around `'Flavio'`. In SQL, single quotes wrap string values. Double quotes wrap identifiers such as column names, so `WHERE name="Flavio"` fails in PostgreSQL with `ERROR: column "Flavio" does not exist`. MySQL is more forgiving here, but don't rely on that.

The results of a query can be ordered by column value, ascending (the default) or descending, using ORDER BY:

```
SELECT * FROM people ORDER BY name;
```

```
SELECT * FROM people ORDER BY name DESC;
```

Without `ORDER BY`, the database returns rows in whatever order is convenient for it. If order matters to you, say so explicitly. Never assume insertion order.

One more common failure: misspelling a column name. `SELECT nmae FROM people;` stops with `ERROR: column "nmae" does not exist`. The database won't guess what you meant. Check the table definition and run the query again.

Filter rows with WHERE

Write precise conditions with comparison, logical, range, set, pattern, and NULL operators.

A `WHERE` clause keeps rows where its condition is **true**. False and unknown conditions are both excluded.

Suppose `people` contains:

name	active	age	email
Ada	true	36	ada@example.com
Lin	true	15	NULL
Sam	false	42	sam@example.com

This query returns only Ada:

```
SELECT name, email
FROM people
WHERE active = TRUE AND age >= 18;
```

SQL has three-valued logic: true, false, and unknown. A comparison with NULL normally produces unknown:

```
WHERE email = NULL
```

That condition does not find missing email addresses. Use:

```
WHERE email IS NULL
```

The same issue appears with inequality. `email <> 'ada@example.com'` returns Sam, but not Lin, because Lin's comparison is unknown. Use `email <> 'ada@example.com' OR email IS NULL` when missing values belong in the result.

Parentheses make mixed conditions explicit:

```
WHERE active = TRUE
  AND (country = 'Italy' OR country = 'Denmark')
```

Without the parentheses, operator precedence can include inactive rows from one country.

Use `IN` for a set of values, `BETWEEN` for an inclusive range, and `LIKE` for a pattern when those forms express the requirement clearly. They do not remove the need to reason about NULL.

Your action is to predict which three sample rows match four conditions: `age >= 18`, `email IS NULL`, `email <> 'ada@example.com'`, and `active = TRUE OR age >= 18`. Then run the queries.

Sort and limit results

Use ORDER BY and LIMIT to make query results predictable and return only the slice an interface needs.

SQL does not promise a row order unless you request one.

Return the newest five orders like this:

```
SELECT id, created_at, total
FROM orders
ORDER BY created_at DESC, id DESC
LIMIT 5;
```

`DESC` puts later timestamps first. The second key matters when two rows share the same timestamp. Because `id` is unique, the complete ordering is stable.

Without `ORDER BY`, `LIMIT 5` means "any five rows the current plan produces." An index change, database restart, or updated statistics can change which rows happen to appear.

You can sort ascending with `ASC`, which is normally the default:

```
ORDER BY total ASC
```

NULL placement differs between database systems and can also be controlled with database-specific syntax. Decide where missing values belong instead of relying on an accidental default.

`LIMIT` is useful for previews and pagination. Large `OFFSET` values can become slow because the database still finds and skips earlier rows. They can also produce duplicates or gaps while new rows are inserted. For large changing datasets, paginate from the last ordered key instead.

For example, after receiving `(created_at, id)` from the last row, the next query can continue after that pair using syntax suited to your database.

Your action is to insert two rows with the same timestamp. Compare ordering by `created_at` alone with ordering by `created_at, id`, then explain which result is stable.

Summarize with aggregate functions

Use COUNT, SUM, AVG, MIN, and MAX to calculate one result from several rows.

Aggregate functions turn several input rows into a summary.

Suppose the paid orders contain totals `49.00`, `18.50`, and NULL. This query returns one row:

```
SELECT
  COUNT(*) AS order_count,
  COUNT(total) AS priced_orders,
  SUM(total) AS revenue,
  AVG(total) AS average_total,
  MIN(total) AS smallest_total,
  MAX(total) AS largest_total
FROM orders
WHERE status = 'paid';
```

The result is:

```
order_count | priced_orders | revenue | average_total
3           | 2             | 67.50   | 33.75
```

`COUNT(*)` counts rows. `COUNT(total)` counts only rows whose total is not NULL. `SUM`, `AVG`, `MIN`, and `MAX` also ignore NULL inputs.

This difference is important. A NULL total does not become zero. It means the value is missing or unknown, so the average uses two known totals rather than three orders.

When no rows match, `COUNT(*)` returns zero. Other aggregates commonly return NULL. If an interface needs a numeric zero, express that choice:

```
SELECT COALESCE(SUM(total), 0) AS revenue
FROM orders
WHERE status = 'refunded';
```

Filtering happens before aggregation. If the requirement says paid revenue, the `WHERE` clause must exclude pending and refunded rows.

Your action is to predict every aggregate result for three values: `10`, `20`, and NULL. Then run one query and compare your prediction.

Group and filter summaries

Use GROUP BY to calculate one aggregate per category and HAVING to filter the grouped results.

`GROUP BY` forms one group for each distinct grouping value, then calculates aggregates inside each group.

Suppose paid orders belong to Italy, Italy, and Denmark. This query returns one summary row per country:

```
SELECT country, COUNT(*) AS orders, SUM(total) AS revenue
FROM orders
```

```
WHERE status = 'paid'
GROUP BY country
ORDER BY revenue DESC;
```

The two Italian rows contribute to one Italian count and sum. Denmark produces its own summary.

`WHERE` filters individual rows before grouping. `HAVING` filters completed groups:

```
SELECT country, COUNT(*) AS orders
FROM orders
WHERE status = 'paid'
GROUP BY country
HAVING COUNT(*) >= 10;
```

This means "count paid orders by country, then keep countries with at least ten." Putting `COUNT(*) >= 10` in `WHERE` is invalid because the groups do not exist yet.

Every selected column must normally be aggregated or listed in `GROUP BY`:

```
SELECT country, customer_email, COUNT(*)
FROM orders
GROUP BY country;
```

That query is invalid or ambiguous: a country group can contain several customer emails. Decide whether email belongs in the grouping or should not appear.

NULL grouping values normally form one group of their own. Label that group with `COALESCE` when the report needs readable output.

Your action is to group a practice orders table by status. Predict the row count and total for each group, then add a `HAVING` condition that removes one group.

Return distinct values

Use DISTINCT when the result needs unique combinations instead of every matching row.

Use `DISTINCT` when the result needs unique values instead of every matching row:

```
SELECT DISTINCT country
FROM users
ORDER BY country;
```

If 50 users live in Italy and 30 live in Denmark, the result contains two country rows.

With several selected columns, uniqueness applies to the complete combination:

```
SELECT DISTINCT country, city
FROM users;
```

`Italy, Rome` and `Italy, Milan` remain separate. Adding a unique column such as `id` would make every row distinct and remove the benefit.

Several NULL values normally appear as one distinct result value. This is result-set behavior; it does not mean ordinary comparisons consider NULL equal to NULL.

Be suspicious when `DISTINCT` is added only to hide duplicates after a join. The join may be matching more related rows than expected, or the schema may be missing a uniqueness rule. Removing visible duplicates does not fix an incorrect total or an accidental many-to-many relationship.

`DISTINCT` can require sorting or hashing a large result. Use it because the question asks for unique combinations, not as a default decoration.

Your action is to run `DISTINCT` on one column, two columns, and two columns including a unique ID. Predict the number of result rows before each query.

Check your understanding: SQL queries

Check SELECT, WHERE, LIKE, sorting, grouping, aggregates, DISTINCT, ranges, and result limits.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Relationships and joins

Keys, one-to-many and many-to-many relations, joins, and views.

SQL, Unique and Primary keys

Learn how to use the UNIQUE constraint to stop duplicate values in a SQL column, and how a PRIMARY KEY uniquely identifies each row in a table.

With a table created with this command:

```
CREATE TABLE people (
  age INT NOT NULL,
  name CHAR(20) NOT NULL
);
```

We can insert an item more than once.

And in particular, we can have columns that repeat the same value.

We can force a column to have only unique values using the `UNIQUE` key constraint:

```
CREATE TABLE people (
  age INT NOT NULL,
  name CHAR(20) NOT NULL UNIQUE
);
```

Now if you try to add the 'Flavio' twice:

```
INSERT INTO people VALUES (37, 'Flavio');
INSERT INTO people VALUES (20, 'Flavio');
```

You'd get an error:

```
ERROR:  duplicate key value violates unique constraint "people_name_key"
DETAIL:  Key (name)=(Flavio) already exists.
```

A **primary key** is a unique key that has another property: it's the primary way we identify a row in the table.

```
CREATE TABLE people (
```

```
  age INT NOT NULL,
  name CHAR(20) NOT NULL PRIMARY KEY
);
```

The primary key can be an email in a list of users, for example.

The primary key can be a unique `id` that we assign to each record automatically.

Whatever that value is, we know we can use it to reference a row in the table.

Unique and primary keys are backed by indexes under the hood. If you're wondering which other columns in your queries deserve an index, I built a free index advisor that suggests the right `CREATE INDEX` statement.

Connect rows with foreign keys

Model a relationship by storing one table's primary key in a related row and enforcing it with a foreign-key constraint.

A foreign key says that a value must identify an existing row in another table.

Suppose every post belongs to one author:

```
CREATE TABLE authors (
  id INTEGER PRIMARY KEY,
  name VARCHAR(200) NOT NULL
);

CREATE TABLE posts (
  id INTEGER PRIMARY KEY,
  author_id INTEGER NOT NULL,
  title VARCHAR(300) NOT NULL,
  FOREIGN KEY (author_id) REFERENCES authors(id)
);
```

Now this insert fails when author `99` does not exist:

```
INSERT INTO posts (id, author_id, title)
VALUES (1, 99, 'Learning SQL');
```

That rejection protects the relationship even when data arrives from a script or migration instead of the main application.

You must also choose what happens when a referenced author is deleted. Common actions are:

- reject the delete while posts still exist
- `CASCADE` and delete the posts too
- `SET NULL` when an authorless post is valid and the column permits NULL

Choose from the meaning of the data. Cascade is convenient when a child cannot exist alone, but one parent delete can become a large destructive operation.

The foreign key checks integrity. It does not automatically make every join fast in every database. Index foreign-key columns when the workload frequently joins or checks rows through them, then verify the query plan.

Your action is to insert one valid and one invalid post. Then attempt to delete the referenced author

and record the database's configured behavior.

Model one-to-many relationships

Place the foreign key on the many side of a relationship and query the related rows by that key.

One author can write many posts, while each post belongs to one author. This is a **one-to-many** relationship.

The foreign key belongs on the many side:

```
CREATE TABLE posts (
  id INTEGER PRIMARY KEY,
  author_id INTEGER NOT NULL REFERENCES authors(id),
  title VARCHAR(300) NOT NULL
);
```

Each post stores one `author_id`. Several post rows can store the same value.

Find all posts by author `7`:

```
SELECT id, title
FROM posts
WHERE author_id = 7
ORDER BY id;
```

Or join the author name into the result:

```
SELECT authors.name, posts.title
FROM authors
JOIN posts ON posts.author_id = authors.id
WHERE authors.id = 7;
```

Do not store a comma-separated list such as `'4,8,15'` on the author row. The database cannot enforce each listed identifier as a foreign key, and ordinary joins cannot treat the pieces as rows.

The relationship also contains a business decision. If a draft post may exist before an author is assigned, `author_id` can allow NULL. If every post must always have an author, use `NOT NULL`.

Your action is to insert one author and three posts that reference it. Query the posts by foreign key, then explain why the same author data should not be copied into every post row.

Model many-to-many relationships

Use a junction table when rows on either side can relate to several rows on the other side.

A post can have many tags, and a tag can belong to many posts. Neither table has one column that can represent the complete relationship.

Add a **junction table**:

```
CREATE TABLE post_tags (
  post_id INTEGER NOT NULL REFERENCES posts(id) ON DELETE CASCADE,
  tag_id INTEGER NOT NULL REFERENCES tags(id) ON DELETE CASCADE,
  PRIMARY KEY (post_id, tag_id)
);
```

Each row represents one relationship. The composite primary key prevents the same tag from being

attached to the same post twice.

Add a tag to a post with an ordinary insert:

```
INSERT INTO post_tags (post_id, tag_id)
VALUES (12, 3);
```

Read all tags for the post:

```
SELECT tags.name
FROM tags
JOIN post_tags ON post_tags.tag_id = tags.id
WHERE post_tags.post_id = 12
ORDER BY tags.name;
```

The junction can hold facts about the relationship itself. If tags are attached by users, add columns such as `added_by` and `added_at` to `post_tags`, not to `posts` or `tags`.

Without the primary key or a unique constraint, duplicate junction rows can multiply join results and counts. `DISTINCT` might hide the symptom, but enforcing the real rule prevents the bad state.

The cascading deletes above are deliberate: a relationship row has no meaning after its post or tag disappears. Review that choice for your own data.

Your action is to connect two posts to two tags, attempt a duplicate relationship, and query both directions: tags for one post and posts for one tag.

SQL Joins

Learn how to perform a join between two SQL tables using JOIN ON to correlate matching columns, with a clear example combining a people and cars table.

Joins are a very powerful tool. Remember relational algebra from the database intro module?

Joins are **applied relational algebra**.

Suppose you have 2 tables, `people` and `cars`:

```
CREATE TABLE people (
  age INT NOT NULL,
  name CHAR(20) NOT NULL PRIMARY KEY
);

CREATE TABLE cars (
  brand CHAR(20) NOT NULL,
  model CHAR(20) NOT NULL,
  owner CHAR(20) NOT NULL PRIMARY KEY
);
```

We add some data:

```
INSERT INTO people VALUES (37, 'Flavio');
INSERT INTO people VALUES (8, 'Roger');
INSERT INTO cars VALUES ('Ford', 'Fiesta', 'Flavio');
INSERT INTO cars VALUES ('Ford', 'Mustang', 'Roger');
```

Now say that we want to correlate the two tables, because the police stopped Roger driving, looks

young, and want to know his age from their database.

Roger is my dog, but let's suppose dogs can drive cars.

We can create a **join** with this syntax:

```
SELECT age FROM people JOIN cars ON people.name = cars.owner WHERE cars.model='Mustang';
```

We'll get this result back:

```
 age
-----
   8
```

What is happening? We are joining the two tables cars on two specific columns: `name` from the `people` table, and `owner` from the `cars` table.

Joins are a topic that can grow in complexity because there are many different kind of joins that you can use to do fancier things with multiple tables, but here is the most basic example.

When join queries get long and hard to read, I built a free SQL formatter that prettifies them for you.

INNER JOIN and LEFT JOIN

Choose whether a query should keep only matching rows or every row from the left table.

Suppose `authors` contains Ada and Lin, but only Ada has a post.

An inner join returns matching pairs:

```
SELECT authors.name, posts.title
FROM authors
INNER JOIN posts ON posts.author_id = authors.id;
```

The result contains Ada and her post. Lin disappears because no post row matches.

A left join keeps every row from the left table:

```
SELECT authors.name, posts.title
FROM authors
LEFT JOIN posts ON posts.author_id = authors.id;
```

Lin now appears with a NULL title. That NULL was produced by the missing match; it is not a stored post.

Use an inner join when the result requires related rows. Use a left join when the absence is meaningful, such as finding authors with no posts:

```
SELECT authors.name
FROM authors
LEFT JOIN posts ON posts.author_id = authors.id
WHERE posts.id IS NULL;
```

Be careful when filtering the right table. This condition removes NULL matches and effectively turns the left join into an inner join:

```
WHERE posts.status = 'published'
```

When the requirement is “all authors, plus their published posts,” place that condition in the join:

```
LEFT JOIN posts
  ON posts.author_id = authors.id
  AND posts.status = 'published'
```

One-to-many joins can return several rows for one author. That is correct when the author has several posts; do not add `DISTINCT` automatically.

Your action is to run both joins with one author who has no posts. Then move a right-table filter between `WHERE` and `ON` and explain the changed result.

SQL Views

Learn how to create a SQL view, a virtual table built dynamically from the result of a SELECT query, and how to delete it later with DROP VIEW.

An interesting thing you can do with SQL is to create a **view**.

A view is like a table, except instead of being a real table, on its own, it is dynamically built by the result of a SELECT query.

Views exist to give a name to a query you keep repeating. Instead of pasting the same join into every report, you define it once and query the view like any other table. It also hides complexity: whoever reads from the view doesn't need to know how the join works.

Let's use the example we used in the joins lesson:

```
CREATE TABLE people (
  age INT NOT NULL,
  name CHAR(20) NOT NULL PRIMARY KEY
);

CREATE TABLE cars (
  brand CHAR(20) NOT NULL,
  model CHAR(20) NOT NULL,
  owner CHAR(20) NOT NULL PRIMARY KEY
);
```

We add some data:

```
INSERT INTO people VALUES (37, 'Flavio');
INSERT INTO people VALUES (8, 'Roger');
INSERT INTO cars VALUES ('Ford', 'Fiesta', 'Flavio');
INSERT INTO cars VALUES ('Ford', 'Mustang', 'Roger');
```

We can create a view that we call `car_age` that always contains the correlation between a car model and its owner's age:

```
CREATE VIEW car_age AS SELECT model, age AS owner_age FROM people JOIN cars ON people.name =
```

Here is the result we can inspect with `SELECT * FROM car_age`:

```
        model         | owner_age
----------------------+-----------
 Fiesta               |        37
 Mustang              |         8
```

The word ”always” matters here. A view stores the query, not the data. Every time you select from it, the database runs the underlying SELECT again, so the result reflects the current state of the tables. Add another owner and car:

```
INSERT INTO people VALUES (41, 'Anna');
INSERT INTO cars VALUES ('Fiat', 'Panda', 'Anna');
```

Query the view again and the Panda row is already there. No refresh step needed.

That is also the catch. A view does not make a slow query fast, because the query still runs on every access. If you need stored, precomputed results, PostgreSQL offers materialized views, which trade freshness for speed. A plain view is about naming and reuse, not performance.

To change a view's definition, PostgreSQL and MySQL accept `CREATE OR REPLACE VIEW car_age AS ...`. In SQLite you drop the view and create it again.

The view is persistent, and will look like a table in your database. You can delete a view using `DROP VIEW`:

```
DROP VIEW car_age
```

Dropping a view never touches the underlying tables. Only the saved query is removed. The reverse is not true: if you try to drop a table a view depends on, PostgreSQL refuses with a dependency error until the view is gone.

Check your understanding: joins and relationships

Check primary and foreign keys, cardinality, INNER and LEFT JOIN behavior, uniqueness, junction tables, and views.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Databases in practice

Choose a database, protect queries, and keep changes reliable.

How to install SQLite on macOS

Learn how to install and use SQLite on macOS, where it comes preinstalled, by running the sqlite3 command and connecting to a database file with TablePlus.

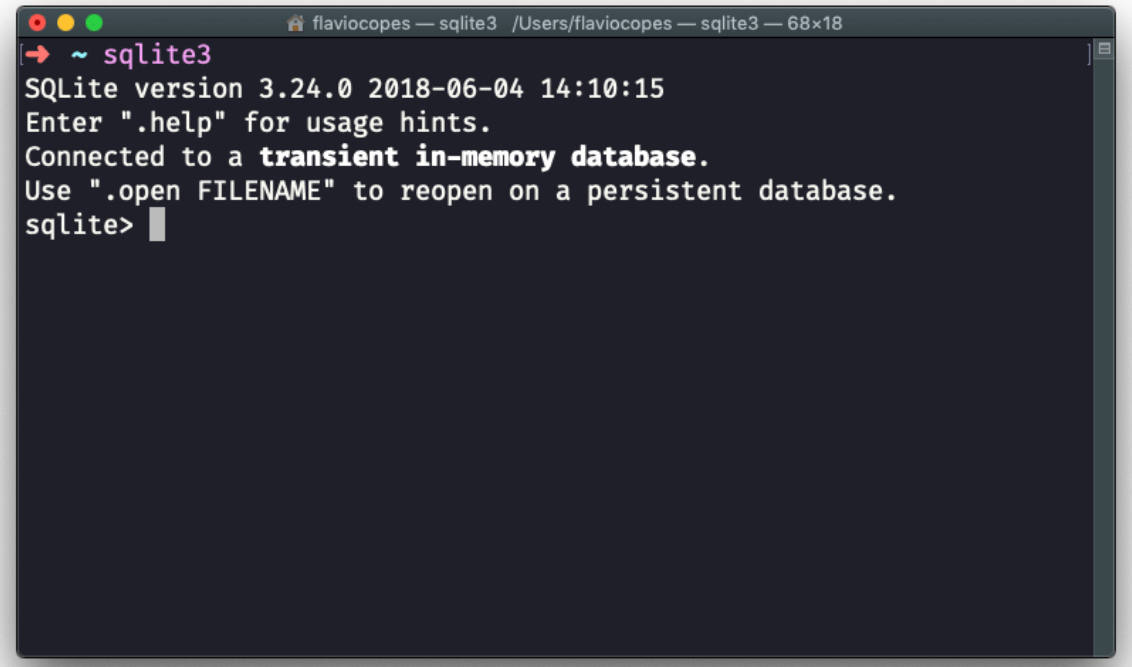
> I'm a Mac user and I do not own a Windows computer, so I can't write the instructions for Windows. Google ”how to install sqlite on windows” for specific instructions. Same goes for Linux.

On macOS, you don't need to do anything to install sqlite. It's preinstalled in all modern versions of macOS.

All you need to do is to open a terminal and run the

```
sqlite3
```

command.



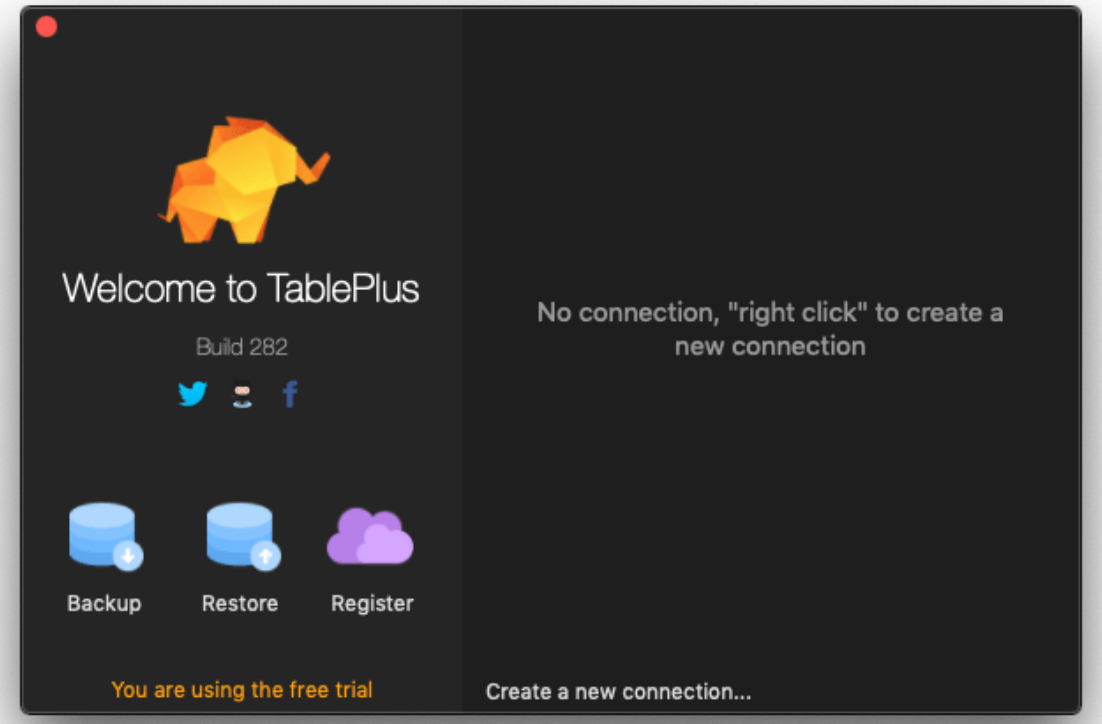
Press ctrl-C 2 times to exit the SQLite executable.

This is pretty cool!

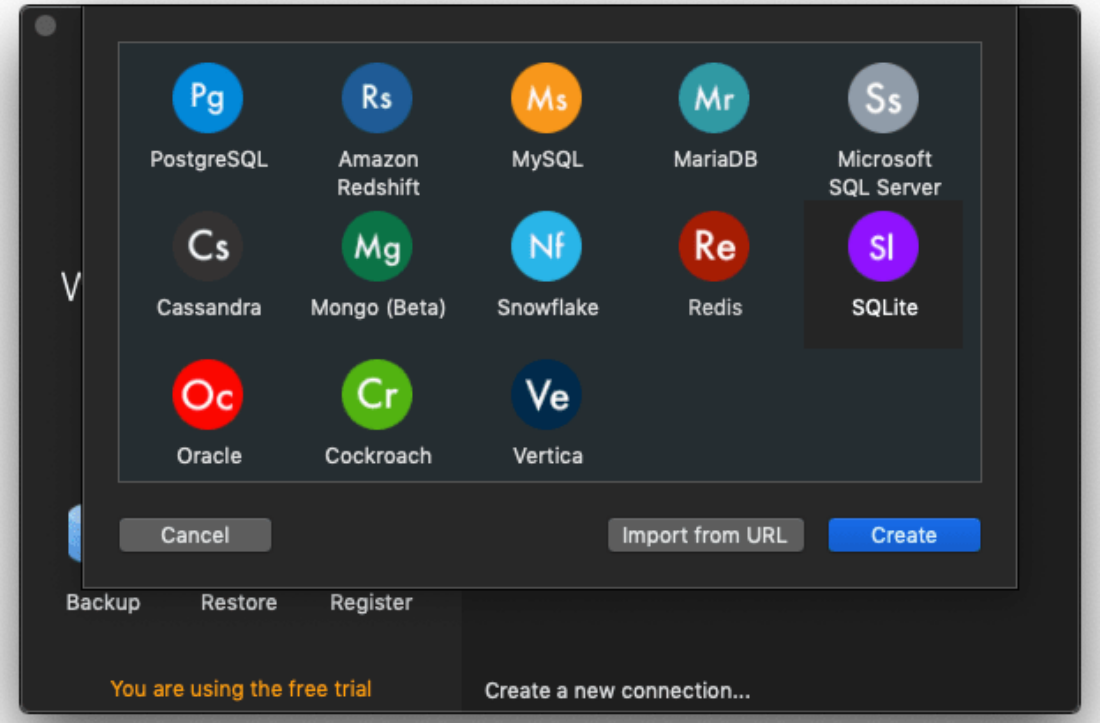
> My macOS with Mojave comes with SQLite 3.24, and the latest version (at the time of writing) is SQLite 3.30. You can upgrade SQLite using Homebrew, but for the sake of simplicity, I'm not going to cover this.

A great software we can use to interact with a SQLite database is TablePlus. It comes with a free trial that's perfect for our usage, because it's not time-based but rather it limits the amount of concurrent connections you can make to the database.

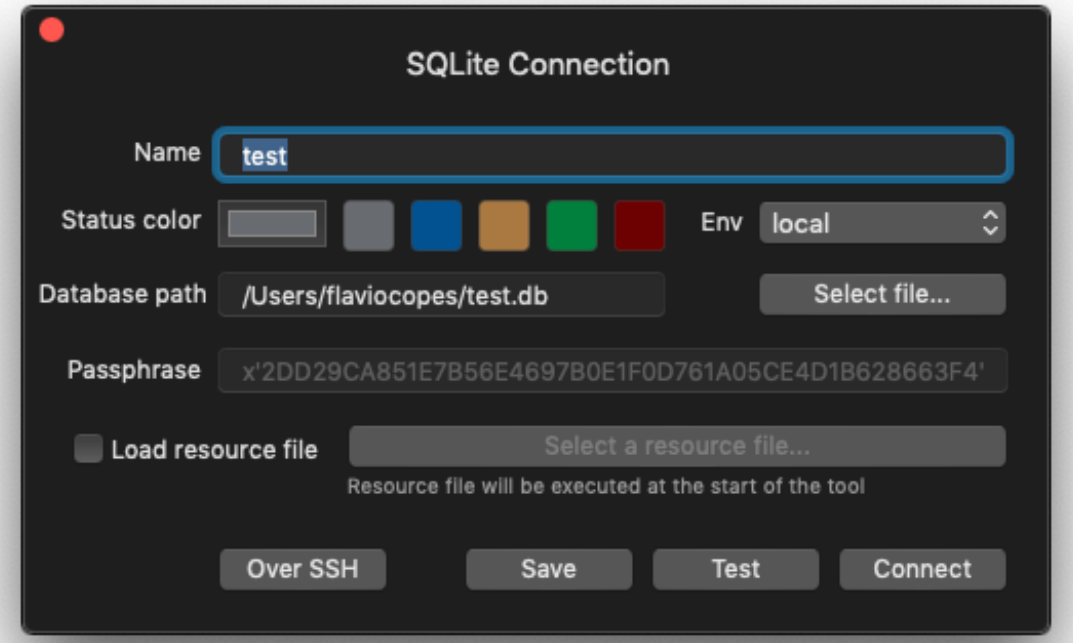
Download it from https://tableplus.com. I know there are macOS, Windows and Linux versions.



You create a new connection, choosing SQLite:



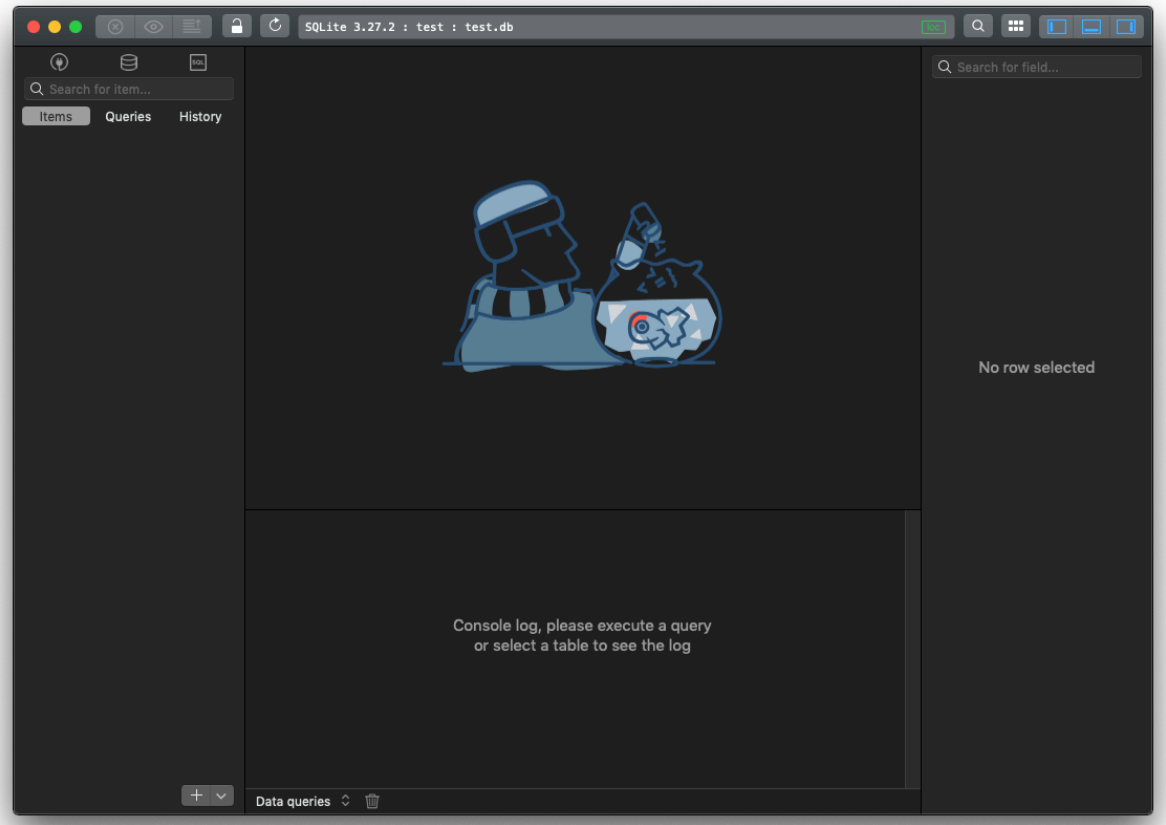
You select a name, and type a database path. I choose `test.db`, in the `/Users/flaviocopes/` folder:



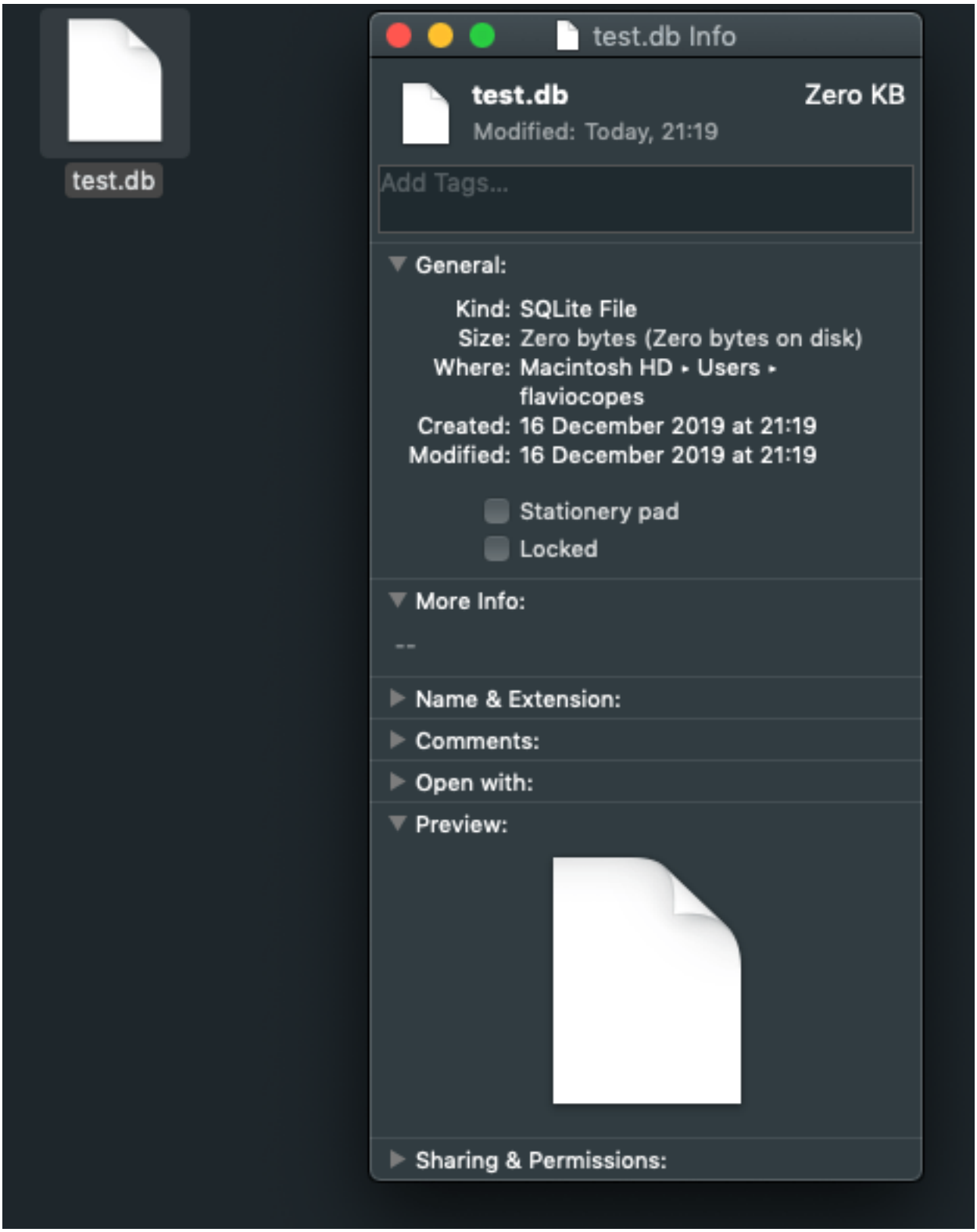
SQLite is pretty cool because the database is contained in a file, which you can put pretty much everywhere you want. This is radically different from PostgreSQL, and MySQL and other big DBMS.

If you're wondering how SQLite compares to those bigger databases, I built a free Postgres vs SQLite vs MySQL comparison tool that lays out the differences.

Pressing `Connect`, the connection was successfully created:



and I can see the file created in that folder, with zero KB of size:



That's it.

I'm going to make more tutorials on how to use this SQLite database, soon.

Choose SQLite, PostgreSQL, or MySQL

Choose a relational database based on deployment, concurrency, operational needs, and the environment that will run it.

SQLite stores a database in one file and runs inside your application process. It is excellent for local tools, prototypes, and many small applications.

There is no server to install and nothing to keep running. Your application opens the file directly:

```
sqlite3 app.db "CREATE TABLE notes (id INTEGER PRIMARY KEY, body TEXT);"
```

That one command created a complete database. Copying `app.db` is a full backup. This is why SQLite is everywhere: phones, browsers, and a huge number of single-server web apps.

The limit is concurrency. SQLite handles many readers at once, but only one writer at a time. If your app runs on one machine and writes are not constant, that's fine. When several processes fight over writes, you start seeing `SQLITE_BUSY` errors, and that is the signal you have outgrown it.

When you want a server

PostgreSQL and MySQL run as database servers. They are a better fit when several application instances or users need concurrent network access and stronger operational tooling.

Your application connects to a server database over the network with a connection string:

```
postgres://app_user:secret@db.example.com:5432/app
mysql://app_user:secret@db.example.com:3306/app
```

A server database accepts many concurrent connections, manages its own users and permissions, and supports replication and point-in-time recovery. The price is operations: someone must install it, patch it, back it up, and keep it running. That someone is you, or a managed hosting provider you pay.

Which one?

Both servers are proven at enormous scale. My advice for a new project is PostgreSQL: it is stricter about your data, and features like rich column types and strong JSON support age well. Pick MySQL when your team or your hosting is already built around it.

A common mistake is starting a small single-server project on a database server "to be ready for scale". You pay the operational cost immediately, for capacity you may never need. Moving from SQLite to PostgreSQL later is a well-understood migration.

All three are capable relational databases. Prefer the simplest one that satisfies the real requirements.

Use indexes deliberately

Understand how an index can speed reads while adding storage and write cost.

An index gives the database another structure for finding rows without scanning the entire table.

Suppose the application repeatedly loads one customer's recent orders:

```
SELECT id, total_cents, created_at
FROM orders
WHERE customer_id = 42
ORDER BY created_at DESC
LIMIT 20;
```

A composite index can support both the filter and the requested order:

```
CREATE INDEX orders_customer_created_idx
ON orders (customer_id, created_at DESC);
```

Column order matters. This index starts with `customer_id`, so it is a natural fit for queries that first narrow rows by customer. It may be less useful for a query that filters only by `created_at`.

An index is not a command that forces the database to use it. The query planner estimates whether an index lookup is cheaper than scanning the table. A scan can be faster for a small table or a condition that matches most rows.

Inspect the plan instead of guessing:

```
EXPLAIN
SELECT id, total_cents, created_at
FROM orders
WHERE customer_id = 42
ORDER BY created_at DESC
LIMIT 20;
```

The exact `EXPLAIN` output differs between databases. Test with representative data: an empty development table tells you little about production behavior.

Every index consumes disk and cache space. The database must also maintain it during `INSERT`, `UPDATE`, and `DELETE`, so indexing every column can make writes slower without improving important queries.

Your action: run `EXPLAIN` for one frequent query before and after adding a matching index. Record whether the plan changes, then explain why that read benefit is worth the extra write cost.

Keep related changes together with transactions

Wrap related statements in a transaction so they all succeed or all roll back.

A money transfer changes at least two balances. Saving only one change would leave the data inconsistent.

Treat the transfer as one unit:

```
BEGIN;

UPDATE accounts
SET balance_cents = balance_cents - 5000
WHERE id = 10;

UPDATE accounts
SET balance_cents = balance_cents + 5000
WHERE id = 20;

COMMIT;
```

`COMMIT` makes both changes durable. If a statement fails, the application should issue `ROLLBACK` so neither change remains:

```
ROLLBACK;
```

Atomicity does not make incorrect business logic correct. The first update must also prove that

account 10 exists and has enough funds. You might enforce a non-negative balance with a constraint and check the affected-row count, or lock and read the account using the concurrency features of your database.

Application code must execute every statement on the same database connection. Sending `BEGIN` on one pooled connection and the updates on another does not create one transaction.

Keep transactions short. Do not wait for an HTTP request, user input, or a slow unrelated job while holding database locks. Concurrent transactions can block each other or deadlock. Applications commonly retry the complete transaction after a transient deadlock—not just the final statement—because the database rolled back the whole unit.

Transaction and locking syntax differs between database systems, but the boundary should always match one business operation.

Your action: run a two-statement transaction against test data, deliberately make the second statement fail, issue `ROLLBACK`, and verify with a `SELECT` that the first change did not survive.

SQL injection

Learn what SQL injection is, how attackers exploit unsanitized input to run malicious SQL queries, and why input sanitization is your main defense.

SQL injection is one of the biggest threats to applications that are database-driven and use SQL queries, and it's all linked to input sanitization.

Suppose we use Node.js to run a simple query like this (I'm using pseudocode):

```
const color = //coming from user input
const query = `select * from cars where color = '${color}'`
```

If `color` is a string that contains a color like `red` or `blue`, everything works as planned.

But what if you accept this string from an `input` field in a form, and the attacker enters the string `"blue'; drop table cars;"`

Do you see what happens?

The value of `query` now is

```
select * from cars where color = 'blue'; drop table cars;'
```

And if you run this query, unless you removed the option to drop the table from the database permission of the database user, that is going to wipe out all of your data.

Another example.

Suppose you perform a query like this:

```
const query = 'SELECT * FROM users where name = "' + name + '"'
```

If you accept the `name` variable from a form, for example, and don't sanitize it, a person could enter the value

```
flavio"; DELETE * FROM users; SELECT * FROM users where name ="flavio
```

See? Now the query will become

```
SELECT * FROM users where name = "flavio"; DELETE * FROM users; SELECT * FROM users where nam
```

This will cause the users table to be wiped out.

We solve this problem by properly sanitizing the input, escaping quotes, and using a proper ORM like Prisma or Sequelize (JS) or Eloquent (Laravel) instead of performing SQL queries directly.

Use parameterized queries

Keep SQL syntax separate from untrusted values so submitted data cannot become executable query text.

Never build a query by concatenating user input into the SQL string.

This is unsafe because `submittedEmail` becomes part of the SQL program:

```
const sql = `SELECT id, email FROM users WHERE email = '${submittedEmail}'`
const result = await db.query(sql)
```

Use a placeholder and pass the value separately:

```
const result = await db.query(
  'SELECT id, email FROM users WHERE email = $1',
  [submittedEmail]
)
```

The database receives the SQL structure and the value through separate channels. A value such as `' OR 1=1 --` remains a string to compare with `email`; it cannot add an `OR` condition to the query.

Placeholder syntax depends on the driver. PostgreSQL drivers commonly use `$1`, while other drivers use `?` or named placeholders. Use the parameter API documented by your driver rather than trying to escape input yourself.

Parameters represent values. They usually cannot stand for table names, column names, keywords, or sort directions. If a user can choose a sort order, map a small allow-listed value to SQL you control:

```
const orderBy = submittedSort === 'oldest'
  ? 'created_at ASC'
  : 'created_at DESC'
```

Parameterized queries prevent SQL injection through values. They do not replace input validation or authorization: a safely encoded user ID is still dangerous if the signed-in user is not allowed to access it.

Your action: pass a quote-heavy string such as `' OR 1=1 --` through a parameterized lookup. Verify that it is treated as one email value and that no unrelated rows are returned.

Store password hashes, not passwords

Use a password-hashing function and compare submitted passwords without ever storing the original value.

Never store a password as plain text or reversible encrypted text.

Do not use a fast general-purpose hash such as plain SHA-256 either. Attackers can test enormous numbers of guesses quickly after stealing the database. Password hashing is deliberately slow and configurable.

Use a maintained password-hashing library. Argon2id is the preferred modern choice; scrypt is a

suitable fallback, while bcrypt is mainly appropriate for legacy systems. Follow the library and current security guidance when selecting memory and time costs rather than inventing your own algorithm.

At registration, hash the password and store only the encoded result:

```
CREATE TABLE users (
  id BIGINT PRIMARY KEY,
  email TEXT NOT NULL UNIQUE,
  password_hash TEXT NOT NULL
);
```

A good library generates a unique random salt and includes the salt, algorithm, and cost parameters in the encoded hash. You usually do not need a separate `salt` column.

At sign-in, load `password_hash` and call the library's verification function with the submitted password. Do not hash the submitted value yourself and compare two strings: the library understands the encoded parameters and performs the comparison safely.

Work factors change as hardware improves. Many libraries can tell you that an old hash needs upgrading. After a successful login, rehash the password with the current policy and replace the stored hash.

Hashing limits the damage of a database leak; it does not make authentication complete. Add rate limiting, secure password reset flows, session protection, and multi-factor authentication where appropriate. Password-reset tokens are secrets too: store a hash of the token, give it a short expiry, and make it single-use.

Never log passwords, even during debugging.

Your action: use your language's maintained Argon2id library to hash one test password. Verify that the correct password succeeds, a wrong password fails, and two hashes of the same password differ because they use different salts.

Plan backups and schema migrations

Treat both stored rows and schema changes as durable application state that needs repeatable recovery and deployment procedures.

A backup is useful only if you can restore it. Automate backups, keep copies away from the primary system, and test recovery.

Use the backup or snapshot tools designed for your database. A copy made while writes are in progress must still be a consistent snapshot; copying raw database files is not automatically safe. Encrypt backups, restrict access, keep at least one copy away from the primary system, and define a retention policy.

Two targets make the plan concrete:

- **Recovery point objective (RPO):** how much recent data can you afford to lose?
- **Recovery time objective (RTO):** how long can the application be unavailable?

Daily backups cannot meet a one-hour RPO. A large backup that takes eight hours to restore cannot meet a one-hour RTO.

Test restoration into an isolated database. Confirm row counts, constraints, and a few important

application flows. Record how long the restore takes. A successful backup job proves that a file was created, not that the business can recover.

Schema migrations solve a related problem. Store each ordered migration in version control and apply the same reviewed sequence in development, staging, and production. Do not edit production tables by hand.

For risky changes, prefer an expand-and-contract rollout:

1. Add the new column or table without breaking the old application.
2. Deploy code that can work with both shapes and backfill existing data.
3. Verify the backfill and switch reads to the new shape.
4. Remove the old column in a later deployment.

This avoids requiring every application instance and every row to change at the same instant.

Some databases make many schema changes transactional; others auto-commit or rebuild/lock a table. Know the behavior before production. Make long backfills restartable, and separate application rollback from data rollback: deploying the old code may not reverse a destructive migration.

A backup taken before a migration is valuable, but it is not a rollback plan until you know it restores correctly and fits your RTO.

Your action: apply a small migration to a copy of the database, restore the latest backup into a second isolated database, and measure both operations. Verify the restored schema, row counts, constraints, and one critical application query.

Check your understanding: databases in practice

Check database choices, password storage, SQL injection, parameterized queries, application connections, and common relational products.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/sql/.