
# SQL HANDBOOK

FLAVIO COPES

# Table of Contents

# Preface

The SQL Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

In particular, the goal is to get you up to speed quickly with SQL.

This book is written by Flavio. I **publish programming tutorials** on my blog flaviocopes.com and The Valley Of Code.

You can reach me on Twitter @flaviocopes.

Enjoy!

# The SQL Handbook



The goal of this handbook is to get you up and running with SQL, starting from zero knowledge.

You will learn the terms and the main ideas, what is a DBMS, how to structure a database, how to insert data into it, how to query the database.

I will only teach you the basics, and once you're done with this you'll have the knowledge you need to dive deeper.

I called this “SQL Handbook” but I assume zero database knowledge, so I’m first going to give you an introduction to databases, in particular relational databases.

# 1. Information systems, data and information

The modern world is completely centered around **information** and **data.**

What's the difference between information and data?

**Data** is a singular unit of knowledge. It has no intrinsic value on its own. We can't extract meaning out of it, without knowing more about it.

**Information** is something that we can link to data, to be able to attach a meaning to it.

Let me make an example. The number 36 is *data*. Knowing that 36 is the age of Joe is *information*.

Information that the data represents the age is essential knowledge that is key in an information system.

# 2. What is a Database? And a DBMS?

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** (**DBMS**).

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

We ask a DBMS to be efficient, to privately and securely store data, to handle large amounts of data.

# 3. Do you always need a database?

Do you always need a database for your app? Of course not.

Many times you don't even need to store data.

But if you need to store data, you have various solutions.

As with everything in technology, nothing is ideal in every situation.

Computers offer many various ways to store data. The most obvious alternative is **files**.

A common example is a website. Some websites use a database to store data (like WordPress), some do not.

Not using a database in that case means a simpler deploy to a hosting service, since you won't need to use and maintain a database in the first place.

But when handling lots of data, a database is definitely a great way to simplify your life in the long term.

As always, it's all a balance of complexity vs convenience.

# 4. Relational Databases

We have many different kinds of DBMS.

Computer professionals experimented with many different options in the past, and one of these options got very popular: the **Relational Database Management System (**RDBMS).

For simplicity we generally call them **relational databases**.

You might have heard of PostgreSQL, MySQL, Oracle, SQLite, MariaDB, SQL Server.

Those are all relational databases.

Note the “SQL” in their name. They are also called SQL databases, to contrast NoSQL databases (like MongoDB or DynamoDB for example), which are DBMS that do not use this relational model and do not use SQL as their *language*.

Relational databases under the hood organize data using two simple concepts: **tables** and **relations**.

This makes them very intuitive to use, because we are used to using tables to organize things. Think about an Excel or Google Sheets, for example.

A relational database, at a first glance, is similar.

Data is stored in one or more tables.

A **table** is a collection of items organized in rows and columns:

```
| Name   | Age |
| ------ | --- |
| Tony   | 36  |
| Rose   | 28  |
| Juliet | 16  |
```

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, like the type of data stored, is called **schema**.

Each table can define **constrains** upon the data that each row can contain. For example, in the simplest case we can say that the value of a column cannot be empty.

Tables can reference each other, forming relationships.

For example we can say the row of the `car` table with `id` `1` is owned by the user assigned to `id` `2` in the table `users`.

In this way we can link data together and build more complex applications.

Relational databases offer us, users, the ability to interact with them through the **SQL language.**

It looks like this:

```
SELECT * FROM cars

INSERT INTO cars VALUES ('Lamborghini', 2010)
```

We use this language to define the tables **schema**, fill tables with data, and finally query the data when needed.

The rest of this handbook will be focused on SQL.

SQL is rather old, being born in 1986, and it's a battle-tested technology used in all sorts of applications.

# 5. Introduction to SQL

SQL (Structured Query Language) is a language we use to interact with a Database Management System (DBMS) to exchange information with it, in a standardized way.

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, 2019.

SQL is implemented in many popular DBMS: Postgres, 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.

SQL is a huge subject, and you can literally spend years to master all its features.

In this handbook we'll only cover the basics to get you up and running.

I recommend you try running the SQL I explain in an online playground experimenting tool like DB Fiddle or similar.

A great application to run SQL locally on your own database is TablePlus.

As a developer in your applications in your day-to-day you might use a ORM (object relational mapping) library that abstracts SQL and simplifies your life, but it's absolutely essential to know how things work under the hood, and much of the terminology of ORMs reflect the underlying SQL naming conventions.

Plus, some things are just doable using "plain SQL".

So, let's go!

# 6. Creating a table

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`
- `VARCHAR`

- `DATE`
- `TIME`
- `DATETIME`
- `TIMESTAMP`

Numeric types include:

- `SMALLINT`
- `INTEGER`
- `DECIMAL`
- `FLOAT`

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

Consult the manual of your DBMS to see the exact values because this is one of the implementation details that can change.

Also, each DBMS can introduce non-standard types like `BIGINT` or `TINYINT` or `TEXT`, and even advanced ones like JSON or array types.

You can absolutely use those, as long as you know that switching DBMS to a different one, say from PostgreSQL to SQLite, could be problematic if those nonstandard types (and features) are used.

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 VARCHAR(20)
);
```

# 7. Inserting data into a table

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

Take this table:

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

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

```
INSERT INTO people VALUES (37, 'Joe')
```

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

```
INSERT INTO people VALUES (37, 'Joe'), (8, 'Ruby');
```

# 8. Querying data

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

Get all rows and columns:

```
SELECT * FROM people;
```

```
 age | name
-----+--------
  37 | Joe
   8 | Ruby
```

Get only the `name` column:

```
SELECT name FROM people;
```

```
  name
--------
 Joe
 Ruby
```

Count the items in the table:

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

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

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

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

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

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;
```

# 9. Handling empty cell

When we create a table in this way:

```
CREATE TABLE people (
  age INT,
  name VARCHAR(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 | Joe
   8 | Ruby
     |
```

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

```
CREATE TABLE people (
  age INT NOT NULL,
  name VARCHAR(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).
```

> Note that an empty string is a valid non-null value.

# 10. Unique and Primary keys

With a table created with this command:

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

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

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

You'd get an error:

```
ERROR:  duplicate key value violates unique constraint "people_name_key"
DETAIL:  Key (name)=(Joe) 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 VARCHAR(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.

# 11. Update data in a table

The data stored in a table can be updated using the `UPDATE` command:

```
UPDATE people SET age=2 WHERE name='Ruby'
```

It's important to add the `WHERE` clause, otherwise this instruction:

```
UPDATE people SET age=2
```

would update all rows in the table.

# 12. Update a table structure

We can alter an existing table structure using the `ALTER TABLE` command, followed by the alteration you want to make:

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

This will add a new column with empty values:

```
 age |  name   | born_year
-----+--------+-----------
  37 | Joe    |
   8 | Ruby   |
```

To drop a column:

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

This will result in:

```
 age |  name
-----+--------
  37 | Joe
   8 | Ruby
```

# 13. Deleting data and tables

To remove data from a table, use the `DELETE FROM` command.

This deletes all rows:

```
DELETE FROM people;
```

You can use the `WHERE` clause to only remove specific rows:

```
DELETE FROM people WHERE name='Joe';
```

To delete a table instead of the data inside the table, use the `DROP TABLE` command:

```
DROP TABLE people;
```

# 14. Joins

Joins are a very powerful tool to merge data contained into 2 different tables.

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

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

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

We add some data:

```
INSERT INTO people VALUES (37, 'Joe');
INSERT INTO people VALUES (8, 'Ruby');
INSERT INTO cars VALUES ('Ford', 'Bronco', 'Joe');
INSERT INTO cars VALUES ('Ford', 'Mustang', 'Ruby');
```

Now say that we want to correlate the two tables, because the police stopped Ruby driving, looks young, and want to know his age from their database.

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.

# 15. Removing duplicates using DISTINCT

Sometimes you want to filter duplicate values in a table.

For example say you have the following schema:

```
CREATE TABLE people (
  name VARCHAR(20)
);
```

and we add entries to this table.

We have lots of "Joe", apparently:

```
INSERT INTO people VALUES
('Joe'), ('Anna'),
('Roxanne'), ('Paul'),
('Joe'), ('Joe'), ('Joe'),
('Joe'), ('Joe'), ('Joe'),
('Joe'), ('Joe'), ('Joe');
```

If we run the query

```
SELECT * FROM people;
```

we get all the duplicates:

```
| name                |
| ------------------- |
| Joe                 |
| Anna                |
| Roxanne             |
| Paul                |
| Joe                 |
| Joe                 |
| Joe                 |
| Joe                 |
| Joe                 |
| Joe                 |
| Joe                 |
| Joe                 |
| Joe                 |
```

Adding the `DISTINCT` keyword filters out the duplicates:

```
SELECT DISTINCT * FROM people;
```

```
| name                |
| ------------------- |
| Joe                 |
| Anna                |
| Roxanne             |
| Paul                |
```

# 16. Using wildcards with LIKE

Using `LIKE` you can select rows from a table using wildcards.

For example you have this schema and this data:

```
CREATE TABLE people (
  name VARCHAR(20)
);

INSERT INTO people VALUES
('Joe'), ('John'), ('Johanna'), ('Zoe');
```

This query will return the first 3 rows, as they all start with "Jo":

```
SELECT * FROM people WHERE name LIKE 'Jo%';
```

Looking for rows like `Joh%` will only select John and Johanna:

```
SELECT * FROM people WHERE name LIKE 'Joh%';
```

Notice I used the `%` sign to match multiple characters.

You can also use `_` to match one character, so you can pick Joe and Zoe using:

```
SELECT * FROM people WHERE name LIKE '_oe';
```

# 17. Indexes

When the data in a database table becomes a lot the database could start becoming slow in doing some operations.

To prevent this you can add indexes.

When you try to look for specific information in book, you look at the index and jump to the page it tells you.

That's pretty similar to what indexes do in databases. It's a way to tell the database what to optimize for when looking for data.

Indexes start to become a rather advanced topic especially when relations between tables get complex.

To put things simply, which indexes to add does not depend on the schema, but rather on the queries you do.

For example if you have a `users` table with a `name` field, and you frequently get that value in a SELECT, then `name` should be an index.

You create an index using this syntax:

```
CREATE INDEX index_name ON users (name);
```

The same goes for rows using in WHERE clauses on big tables.

Indexes can improve performance, at a cost however: they make selection faster, but slow down insertions and updates.

Also, they increase disk space usage.

As with everything, it's a tradeoff and the performance benefits must be carefully measured.

The good thing is you can add and remove indexes when you run into performance issues, so it's not something you have to optimize for in the beginning.

# 18. Aggregating data

You can ask the database to perform a few operations on the data stored in a table.

For example doing a sum of the values of a numeric field.

Or calculating the average.

For example let's take this database schema and sample data:

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

INSERT INTO people VALUES
('Joe', 20),
('John', 30),
('Johanna', 25),
('Zoe', 23);
```

`AVG()` calculates the average.

If you want to get the average age of those people, you can do this:

```
SELECT AVG(age) FROM people;
```

And here's the result:

```
| avg  |
| ---- |
| 24.5 |
```

You typically alias the result of the function to a name of your choosing:

```
SELECT AVG(age) as average FROM people;
```

```
| average |
| ------- |
| 24.5    |
```

In the same vein, `SUM()` calculates the sum.

```
SELECT SUM(age) as total FROM people;
```

```
| total |
| ----- |
| 98    |
```

Use `MAX()` or `MIN()` to get the maxiumum or minumum value:

```
SELECT MAX(age), MIN(age) FROM people;
```

```
| max | min |
| --- | --- |
| 30  | 20  |
```

# 19. Grouping data

You can group data using the `GROUP BY` clause:

```
SELECT <...> FROM people GROUP BY department;
```

For example let's take this database schema and sample data:

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

INSERT INTO people VALUES
('Joe', 'Sales', 20),
('John', 'Sales', 30),
('Johanna', 'IT', 25),
('Zoe', 'IT', 23);
```

We want to get the average age of each department.

Here's how we can do that:

```
SELECT department, AVG(age) as total FROM people GROUP BY department;
```

And here's the result:

```
| department | total  |
| ---------- | ------ |
| Sales      | 25     |
| IT         | 24     |
```

# 20. Limit and offset

You can limit the number of rows retrieved. This is not a standard SQL and each database does this differently. MySQL and PostgreSQL use `LIMIT`:

```
SELECT * FROM people ORDER BY name LIMIT 10;
```

and you can set an offset to return, in this case, the rows from 11 to 20:

```
SELECT * FROM people ORDER BY name LIMIT 10 OFFSET 10;
```

# 21. Comments

You can add comments to SQL queries using two hyphens at the beginning of a line:

```
-- just a test
SELECT * FROM people;
```

Or use `/* */` to comment a portion of SQL inside a line, or to add multiple lines comments:

```
SELECT * FROM /* test */ people;
```

```
SELECT * FROM /*
this is just a test */
people;
```

# Conclusion

Thanks a lot for reading this book.

For more, head over to The Valley Of Code.

Send any feedback, errata or opinions at hello@thevalleyofcode.com