SQL

Relational Databases

A relational database is a database that organizes information into one or more tables.

A *table* is a collection of data organized into rows and columns. Tables are sometimes referred to as *relations*.

A *column* is a set of data values of a particular type.

```
SELECT * FROM celebs;
```

Statements

A *statement* is text that the database recognizes as a valid command. Statements always end in a semicolon `;`.

```
CREATE TABLE table_name (
   column_1 data_type,
   column_2 data_type,
   column_3 data_type
);
```

`CREATE TABLE` is a *clause*.
Clauses perform specific tasks in SQL. By convention, clauses are
written in capital letters. Clauses can also be referred to as commands.

`table_name` refers to the name of the table that the command is applied to.

`(column_1 data_type, column_2 data_type, column_3 data_type)` is a *parameter*. A parameter is a list of columns, data types,
or values that are passed to a clause as an argument. Here, the
parameter is a list of column names and the associated data type.

Create

`CREATE` statements allow us to create a new table in the database.

Insert

The `INSERT` statement inserts a new row into a table.

```
INSERT INTO celebs (id, name, age)
VALUES (1, 'Justin Bieber', 29);
```

Select

`SELECT` statements are used to fetch data from a database.

```
SELECT name FROM celebs;
```

`*` is a special wildcard
character that we have been using. It allows you to select every column
in a table without having to name each one individually.

```
SELECT * FROM celebs;
```

When using `AS`, the columns are not being renamed in the table. The aliases only appear in the result.

```
SELECT name AS 'Titles
FROM celebs;
```

Alter

The `ALTER TABLE` statement adds a new column to a table.

```
ALTER TABLE celebs
ADD COLUMN twitter_handle TEXT;
```

Update

The `UPDATE` statement edits a row in a table.

```
UPDATE celebs
SET twitter_handle = '@taylorswift13'
WHERE id = 4;
```

Delete

The `DELETE FROM` statement deletes one or more rows from a table.

```
DELETE FROM celebs
WHERE twitter_handle IS NULL;
```

Constraints

Constraints that add information about how a column can be used are invoked after
specifying the data type for a column. They can be used to tell the
database to reject inserted data that does not adhere to a certain
restriction.

```
CREATE TABLE celebs (
   id INTEGER PRIMARY KEY,
   name TEXT UNIQUE,
   date_of_birth TEXT NOT NULL,
   date_of_death TEXT DEFAULT 'Not Applicable'
);
```

Distinct

`DISTINCT` is used to return unique values in the output. It filters out all duplicate values in the specified column(s).

```
SELECT DISTINCT genre
FROM movies;
```

Where

We can restrict our query results using the `WHERE` clause in order to obtain only the information we want.

```
SELECT * FROM movies
WHERE imdb_rating > 8;
```

Comparison operators used with the `WHERE` clause are:

- `=` equal to
- `!=` not equal to
- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to

`LIKE` is a special operator used with the `WHERE` clause to search for a specific pattern in a column.

```
SELECT *
FROM movies
WHERE name LIKE 'Se_en';
```

The `_` means you can substitute any individual character here without breaking the pattern.

`%` is a wildcard character that matches zero or more missing characters in the pattern.

```
SELECT *
FROM movies
WHERE name LIKE 'A%';
```

It is not possible to test for `NULL` values with comparison operators, such as `=` and `!=`.

Instead, we will have to use these operators:

- `IS NULL`
- `IS NOT NULL`

```
SELECT name
FROM movies
WHERE imdb_rating IS NOT NULL;
```

The `BETWEEN` operator is used in a `WHERE` clause to filter the result set within a certain *range*. It accepts two values that are either numbers, text or dates.

```
SELECT *
FROM movies
WHERE year BETWEEN 1990 AND 1999;
```

With `AND`, *both* conditions must be true for the row to be included in the result.

```
SELECT *
FROM movies
```

```
WHERE year BETWEEN 1990 AND 1999
   AND genre = 'romance';
```

Similar to `AND`, the `OR` operator can also be used to combine multiple conditions in `WHERE`, but there is a fundamental difference:

- `AND` operator displays a row if *all* the conditions are true.
- `OR` operator displays a row if *any* condition is true.

Order by

We can *sort* the results using `ORDER BY`, either alphabetically or numerically.

```
SELECT *
FROM movies
ORDER BY year DESC;
```

- `DESC` is a keyword used in `ORDER BY` to sort the results in *descending order* (high to low or Z-A).
- `ASC` is a keyword used in `ORDER BY` to sort the results in *ascending* order (low to high or A-Z).

Note: `ORDER BY` always goes after `WHERE` (if `WHERE` is present).

Limit

`LIMIT` is a clause that lets you specify the maximum number of rows the result set will have.

```
SELECT *
FROM movies
LIMIT 10;
```

Case

A `CASE` statement allows us to create different outputs (usually in the `SELECT` statement). It is SQL's way of handling if-then

logic.

Suppose we want to condense the ratings in `movies` to three levels:

- *If the rating is above 8, then it is Fantastic.*
- *If the rating is above 6, then it is Poorly Received.*
- *Else, Avoid at All Costs.*

```
SELECT name,
 CASE
  WHEN imdb_rating > 8 THEN 'Fantastic'
  WHEN imdb_rating > 6 THEN 'Poorly Received'
  ELSE 'Avoid at All Costs'
 END
FROM movies;
```

We can rename the column to ‘Review’ using `AS`:

```
SELECT name,
 CASE
  WHEN imdb_rating > 8 THEN 'Fantastic'
  WHEN imdb_rating > 6 THEN 'Poorly Received'
  ELSE 'Avoid at All Costs'
 END AS 'Review'
FROM movies;
```

Agregates

Calculations performed on multiple rows of a table are called **aggregates**.

Count

The fastest way to calculate how many rows are in a table is to use the `COUNT()` function.

```
SELECT COUNT(*)
FROM table_name;
```

```
SELECT COUNT(*)
 FROM fake_apps
 WHERE price=0;
```

Sum

`SUM()` is a function that takes the name of a column as an argument and returns the sum of all the values in that column.

```
SELECT SUM(downloads)
FROM fake_apps;
```

Max/Min

The `MAX()` and `MIN()` functions return the highest and lowest values in a column, respectively.

```
SELECT MAX(downloads)
FROM fake_apps;
```

Average

The `AVG()` function works by taking a column name as an argument and returns the average value for that column.

```
SELECT AVG(downloads)
FROM fake_apps;
```

Round

`ROUND()` function takes two arguments inside the parenthesis:

1. a column name

2. an integer

It rounds the values in the column to the number of decimal places specified by the integer.

```
SELECT ROUND(price, 0)
FROM fake_apps;
```

```
SELECT ROUND(AVG(price), 2)
FROM fake_apps;
```

Group by

`GROUP BY` is used in collaboration with the `SELECT` statement to arrange identical data into *groups*.

The `GROUP BY` statement comes after any `WHERE` statements, but before `ORDER BY` or `LIMIT`.

```
SELECT year,
   AVG(imdb_rating)
FROM movies
GROUP BY year
ORDER BY year;
```

SQL lets us use column reference(s) in our `GROUP BY` that will make our lives easier.

- `1` is the first column selected
- `2` is the second column selected
- `3` is the third column selected

and so on.

```
SELECT ROUND(imdb_rating),
   COUNT(name)
FROM movies
```

```
GROUP BY 1
ORDER BY 1;
```

Having

`HAVING` is very similar to `WHERE`. In fact, all types of `WHERE` clauses you learned about thus far can be used with `HAVING`.

- When we want to limit the results of a query based on values of the individual rows, use `WHERE`.
- When we want to limit the results of a query based on an aggregate property, use `HAVING`.

`HAVING` statement always comes after `GROUP BY`, but before `ORDER BY` and `LIMIT`.

```
SELECT year,
   genre,
   COUNT(name)
FROM movies
GROUP BY 1, 2
HAVING COUNT(name) > 10;
```

Join

To combine tables.

```
SELECT *
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id;
```

When we perform a simple `JOIN` (often called an *inner join*) our result only includes rows that match our `ON` condition.

A *left join* will keep all rows from the first table, regardless of whether there is a matching row in the second table.

```
SELECT *
FROM table1
LEFT JOIN table2
  ON table1.c2 = table2.c2;
```

There are special columns called Primary keys that have a few requirements:

- None of the values can be `NULL`.
- Each value must be unique (i.e., you can't have two customers with the same `customer_id` in the `customers` table).
- A table can not have more than one primary key column.

When the primary key for one table appears in a different table, it is called a foreign key. The most common types of joins will be joining a foreign key from one table with the primary key from another table.

Sometimes, we just want to combine all rows of one table with all rows of another table. Those are Cross Joins! They don't require an `ON` statement

```
SELECT shirts.shirt_color,
   pants.pants_color
FROM shirts
CROSS JOIN pants;
```

Union

Sometimes we just want to stack one dataset on top of the other. Well, the `UNION` operator allows us to do that.

```
SELECT *
FROM table1
UNION
SELECT *
FROM table2;
```

- Tables must have the same number of columns.
- The columns must have the same data types in the same order as the first table.

With

Often times, we want to combine two tables, but one of the tables is the result of another calculation.

```
WITH previous_query AS (
   SELECT customer_id,
      COUNT(subscription_id) AS 'subscriptions'
   FROM orders
   GROUP BY customer_id
)
SELECT customers.customer_name,
   previous_query.subscriptions
FROM previous_query
JOIN customers
  ON previous_query.customer_id = customers.customer_id;
```