SQL, Structured Query Language, is a programming language designed to manage data stored in relational databases. SQL operates through simple, declarative statements. This keeps data accurate and secure, and helps maintain the integrity of databases, regardless of size.

The SQL language is widely used today across **web frameworks and database applications**. Knowing SQL gives you the freedom to explore your data, and the power to make better decisions. By learning SQL, you will also learn concepts that apply to nearly every data storage system.
The statements covered in this course use SQLite Relational Database Management System (RDBMS). You can also access a glossary of all the SQL commands taught in this course.

Clauses often written in capital letters. Clauses are can also be referred to as commands.

1- DDL (Data Definition Language)

Data Types
INTEGER - a whole num
TEXT - a text string
DATE - date
REAL - a decimal num

Statement

- **CREATE TABLE** table_name (col1, col2, col3

- **SELECT FROM**
- **INSERT INTO** table_name (col1, col2, col3)
- **VALUES**()

add a new col to a table

```
ALTER TABLE table_name
ADD COLUMN col_name  data_type;
```

edits a row in a table

```
UPDATE table_name
SET col_name = ''
WHERE col_name(key) = xx;
```

Q: how is **ALTER** different from **UPDATE**?

S: The **ALTER** statement is used to modify columns. With ALTER, you can add columns, remove them, or even modify them.
The **UPDATE** statement is used to modify rows. However, UPDATE can only update a row, and cannot remove or add rows.

deletes one or more rows from a table

- **DELETE FROM** table_name
- **WHERE** col_name IS NULL;

Constraints

- PRIMARY KEY
- UNIQUE (This is similar to PRIMARY KEY except a table can have many different UNIQUE columns.)
- NOT NULL
- DEFAULT 'abcd'

2- DML (Data Manipulation Language)

AS is a keyword in SQL that allows you to rename a column or table using an alias. The new name can be anything you want as long as you put it inside of single quotes. Here we renamed the *name* column as *Titles*.

It is important to remember that the columns have not been renamed in the table. The aliases only appear in the result.

DISTINCT
When we are examining data in a table, it can be helpful to know what distinct values exist in a particular column.

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

eg:

```
SELECT DISTINCT col_name
FROM table_name;
```

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

WHERE clause filters the result set to only include rows where the following condition is true.

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

LIKE can be a useful operator when you want to compare similar values.

eg:
How could we select all movies that start with 'Se' and end with 'en' and have exactly one character in the middle?

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

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

The percentage sign % is another wildcard character that can be used with LIKE.

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

eg:

- A% matches all movies with names that begin with letter 'A'
- %a matches all movies that end with 'a'

Q: When using SQL LIKE operators, how do we search for patterns containing the actual characters "%" or "_"?
S: Add \ before

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

Q: When storing missing data, should I store them as NULL?
S: It can depend entirely on how you need the data to be stored and utilized.

Let's say that you have a table of employee information, which included their address. Say that we wanted to check all rows of this table and find where any addresses are missing. If we stored the addresses as **TEXT** values, we might choose to store all the missing values as either ' ' or as NULL.

If we stored the missing address values as an empty **string** ' ' then these values are not NULL. Empty strings are seen as a string of length 0.

So, if we ran a query using
```
WHERE address IS NULL
```
it would not give us the rows with missing address values. We would have to check using
```
WHERE address = ' '
```

With a table containing many different data types, it may be helpful and more convenient to store any missing values in general as just NULL so that we can utilize the IS NULL and IS NOT NULL operators.

Between ... and ...
The **BETWEEN** operator can be used in a **WHERE** clause to filter the result set within a certain range. The values can be numbers, text or dates.

Really interesting point to emphasize again:

- BETWEEN two **letters** is **not inclusive** of the 2nd letter.
- BETWEEN two **numbers** is **inclusive** of the 2nd number.

eg:
```
SELECT *
FROM movies
WHERE name BETWEEN 'A' AND 'J';
```

This statement filters the result set to only include movies with names that begin with letters ‘A’ up to, but not including 'J'.

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

In this statement, the BETWEEN operator is being used to filter the result set to only include movies with years between 1990 up to, and including 1999.

And

Sometimes we want to combine multiple conditions in a **WHERE** clause to make the result set more specific and useful.

One way of doing this is to use the AND operator.

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

Or
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.

With **OR**, if any of the conditions are true, then the row is added to the result.

Order By
It is often useful to list the data in our result set in a particular order.

We can sort the results using **ORDER BY**, either alphabetically or numerically. Sorting the results often makes the data more useful and easier to analyze.

eg:

```
SELECT *
FROM table_name
WHERE [condition]
ORDER BY col_name 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).
- do not have to specify if u want to order alphabetically

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. This saves space on our screen and makes our queries run faster.

LIMIT always goes at the very end of the query. Also, it is not supported in all SQL databases.

Case ... End (If/Then)
A **CASE** statement allows us to create different outputs (usually in the SELECT statement). It is SQL's way of handling if-then logic.

- The **CASE** statement must end with **END**.

eg:

```
SELECT col_name,
  CASE
    WHEN [condition] THEN 'a_string'
    WHEN [condition] THEN 'a_text'
    WHEN [condition] THEN 'a_integer/real'
    ELSE 'a_string/text/integer/real'
  END (AS [Alias])
FROM table_name;
```

Let's summarize:
SELECT is the clause we use every time we want to query information from a database.
AS renames a column or table.
DISTINCT return unique values.
WHERE is a popular command that lets you filter the results of the query based on conditions that you specify.
LIKE and **BETWEEN** are special operators.
AND and **OR** combines multiple conditions.
ORDER BY sorts the result.
LIMIT specifies the maximum number of rows that the query will return.
CASE creates different outputs.

3- Aggregate Functions

We've learned how to write queries to retrieve information from the database. Now, we are going to learn how to perform calculations using SQL.

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

COUNT(): count the number of rows

SUM(): the sum of the values in a column
MAX()/MIN(): the largest/smallest value
AVG(): the average of the values in a column
Q: how can we get the average of only the unique values of a column?
S: use the DISTINCT clause right before the column name, such as: AVG(DISTINCT col_name)

ROUND(col_name, INTEGER): round the values in the column
takes two arguments inside the parenthesis:

- a column name
- an integer

GROUP BY
GROUP BY is a clause in SQL that is used with aggregate functions. It 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.

Sometimes, we want to **GROUP BY** a calculation done on a column.

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.
eg:

```
SELECT ROUND(imdb_rating), COUNT(name)
FROM movies
GROUP BY ROUND(imdb_rating)
ORDER BY ROUND(imdb_rating);
```

The following query is equivalent to the one above:

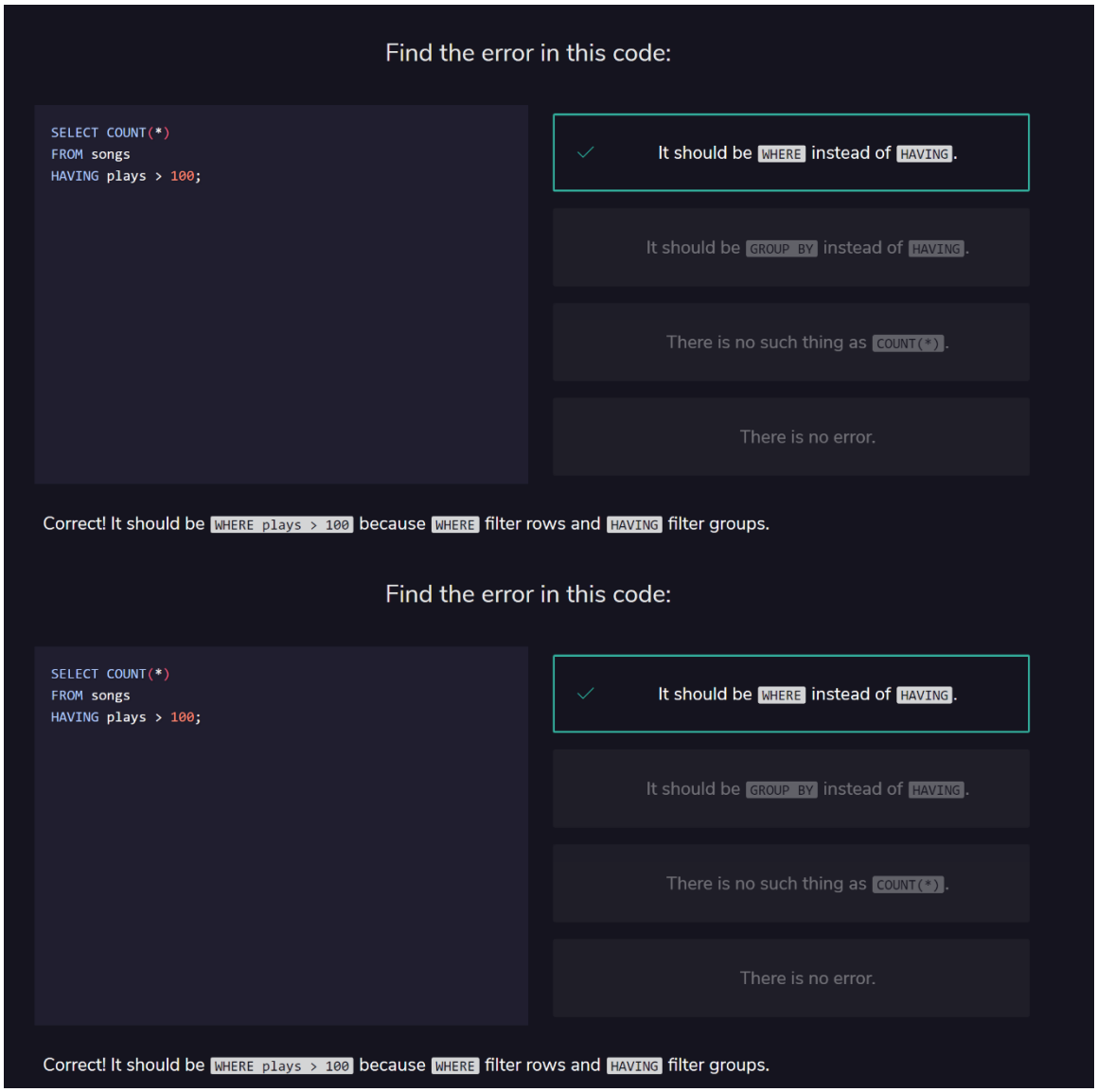
```
SELECT ROUND(imdb_rating), COUNT(name)
FROM movies
GROUP BY 1
ORDER BY 1;
```

Here, the 1 refers to the first column in our SELECT statement, ROUND(imdb_rating).

Having

In addition to being able to group data using GROUP BY, SQL also allows you to filter which groups to include and which to exclude.

- similar to WHERE
- used to filter groups
- 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.

Q: Can a WHERE clause be applied with a HAVING statement in the same query?
S: Yes, you can absolutely apply a WHERE clause in a query that also utilizes a HAVING statement.

When you apply a WHERE clause in the same query, it must always be before any GROUP BY, which in turn must be before any HAVING.

As a result, the data is essentially filtered on the WHERE condition first. Then, from this filtered data, it is grouped by specified columns and then further filtered based on the HAVING condition.

eg:

```
SELECT genre, ROUND(AVG(score))
FROM movies
WHERE box_office > 500000
GROUP BY genre
HAVING COUNT(*) > 5;
```

This will first filter the movies with a box_office > 500000. Then, it will group those results by genre, and finally restrict the query to genres that have more than 5 movies.

4- Multiple Tables

Learn how to query tables by using JOIN function

JOIN

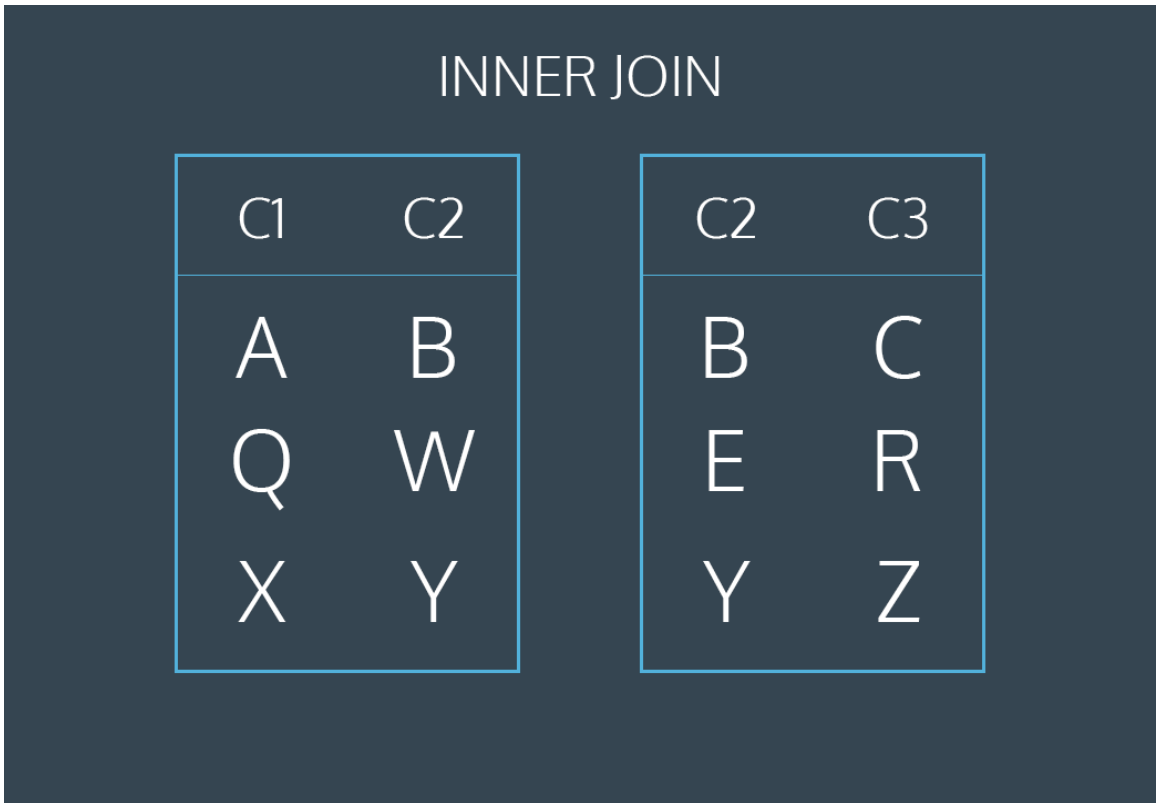
join two or more tables together

eg:

```
SELECT *
FROM table1_name
JOIN table2_name
  ON table1_name.column_name(usually is id) = table2_name.column_name(usually is id);
```

INNER JOIN

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

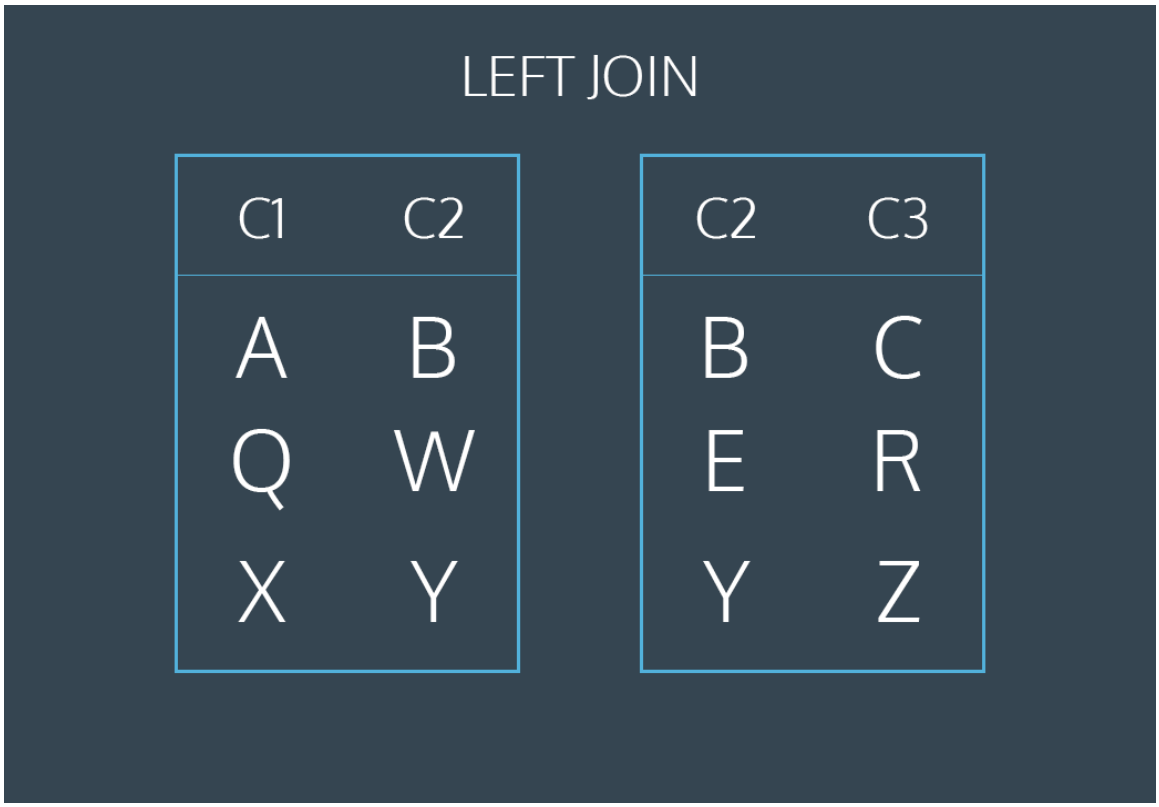


LEFT JOIN
What if we want to combine two tables and keep some of the unmatched rows?

SQL lets us do this through a command called **LEFT JOIN**. A left join will keep all rows from the first table, regardless of whether there is a matching row in the second table.

eg:

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



The final result will keep all rows of the first table but will omit the un-matched row from the second table.

Q: When should I use INNER JOIN vs LEFT JOIN?
S: Depending on how we want to select the combined data, it can determine whether to use an INNER JOIN or a LEFT JOIN.

Generally, we use INNER JOIN when we want to select only rows that match an ON condition. If no rows match the ON condition, then it will not return any results. This can be somewhat stricter than using a LEFT JOIN.

We use a LEFT JOIN when we want every row from the first table, regardless of whether there is a matching row from the second table. This is similar to saying,
"Return all the data from the first table no matter what. If there are any matches with the second table, provide that information as well, but if not, just fill the missing data with NULL values."

In a way, LEFT JOIN is less strict than INNER JOIN. Furthermore, the results of a LEFT JOIN will actually include all results that an INNER JOIN would have provided for the same given condition.

Primary Key vs Foreign Key

The most common types of joins will be joining a foreign key from one table with the primary key from another table.

Cross Join
So far, we've focused on matching rows that have some information in common.

Sometimes, we just want to combine all rows of one table with all rows of another table.

Notice that cross joins don't require an **ON** statement. You're not really joining on any columns!

A more common usage of **CROSS JOIN** is when we need to compare each row of a table to a list of values.

Union
the UNION operator allows us to stack one dataset on top of the other.

SQL has strict rules for appending data:

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

eg:

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

Q: What happens if the tables we perform the UNION operator on have duplicate rows?
S: When you combine tables with UNION, duplicate rows will be excluded.

To explain why this is the case, recall a Venn Diagram, which shows the relations between sets. If we perform **UNION** on two sets of data (tables), say A and B, then the data returned in the result will essentially be A + B - (A intersect B)

WITH

- The WITH statement allows us to perform a separate query (such as aggregating customer's subscriptions)
- previous_results is the alias that we will use to reference any columns from the query inside of the WITH clause

Essentially, we are putting a whole first query inside the parentheses () and giving it a name. After that, we can use this name as if it's a table and write a new query using the first query.

eg:

```
   SELECT customer_id, COUNT(subscription_id) AS 'subscriptions'
   FROM orders
   GROUP BY customer_id;

   WITH previous_results AS (
  SELECT customer_id, COUNT(subscription_id) AS 'subscriptions'
   FROM orders
   GROUP BY customer_id
)
   SELECT *
   FROM previous_results
   JOIN customers
      ON _____ = _____;
```

Let's summarize what we've learned so far:
JOIN will combine rows from different tables if the join condition is true.
LEFT JOIN will return every row in the left table, and if the join condition is not met, NULL values are used to fill in the columns from the right table.
Primary key is a column that serves a unique identifier for the rows in the table.
Foreign key is a column that contains the primary key to another table.
CROSS JOIN lets us combine all rows of one table with all rows of another table.
UNION stacks one dataset on top of another.
WITH allows us to define one or more temporary tables that can be used in the final query.

Q: When should we use each type of JOIN covered in the lesson?
S: Depending on how you need to select the data, each type of JOIN may accomplish a specific goal.

JOIN (or INNER JOIN)
Use JOIN, or INNER JOIN, when you want to strictly select rows of data that match some condition, provided by an **ON** clause. For example, if we had tables of employee information, and require their company information, we could **JOIN** the tables together so that we only obtain results that match, and provide us all the information we need, excluding results that are missing information.

LEFT JOIN

LEFT JOIN can be used when you want to see all the results from the first table no matter what, but also want to include matches, if any, with the second table. For example, say we had a table for customers and another table for purchase information. If we wanted to obtain all customer information, and any purchase information, then a LEFT JOIN might be useful. If the customers made no purchases, their information will still be returned.

CROSS JOIN

CROSS JOIN can be used when you want to get combinations of rows from a table with other tables. A simplified example of using this would be, say we wanted to get every possible combination of a meal, which consists of an appetizer, main course, and dessert. Using a CROSS JOIN can give us every possible combination.

UNION

Although not necessarily a JOIN, UNION can be used when you wish to combine multiple tables together quickly. One helpful feature of a UNION is that it will only return unique rows, so there will be no duplicates in the combined table.