**Learn SQL-**

Common data types – Integer, text, date, real(decimal)

**Create basic table –**

CREATE TABLE friends (

id INTEGER,

name TEXT,

birthday DATE

);

**Add to table-**

INSERT INTO table\_name (column1, column2, column3)

VALUES (value1, value2, value3);

INSERT INTO friends (id, name, birthday)

VALUES (1, 'Jane Doe', '1990-05-30');

SELECT \* From friends;

**Update table-**

UPDATE table\_name

SET column1 = value1

WHERE condition;

UPDATE friends

SET name = 'Jane Smith'

WHERE id = 1;

**Add Column –**

ALTER TABLE table\_name

ADD COLUMN column DATA\_TYPE;

ALTER TABLE friends

ADD COLUMN email TEXT;

**Remove from table-**

DELETE FROM friends

WHERE id = 1;

**Queries –**

**Return all -**

SELECT \* From table\_name;

**Return certain columns -**

SELECT column1, column2 FROM table\_name;

**Return columns with a new name –**

SELECT column AS ‘New\_name” FROM table\_name;

**Distinct(returns unique values, filters out duplicates)-**

SELECT DISTINCT column FROM table\_name;

**Where(only data we want)-**

SELECT \* FROM table\_name WHERE column meets you condition;

SELECT \* FROM movies WHERE imdb\_rating > 8;

Could be !=, <, >, <=, >=, or =

**LIKE(operator used with the WHERE clause to search for a specific pattern)-**

SELECT \* FROM table\_name WHERE column LIKE ‘whatever string you want’;

**the \_ is a wild card- To get all movies that start with SE and end with en-**

SELECT \* FROM movies WHERE name LIKE ‘Se\_en’;

**the % is a wild card- To get all movies that start with an A-**

SELECT \* FROM movies WHERE name LIKE ‘A%’;

**To end with a-**

SELECT \* FROM movies WHERE name LIKE ‘%a’;

**To find something with man in the name-**

SELECT \* FROM movies WHERE name LIKE ‘%man%’

**To find something start starts with a certain word-**

SELECT \* FROM movies WHERE name LIKE “The %’;

**Is NULL, IS NOT NULL(finds items with or without filled columns)-**

**Will only return items for colomnA where columnB is populated-(use IS NULL for opposite)**

SELECT columnA FROM table\_name WHERE columnB IS NOT NULL;

SELECT name FROM movies WHERE imdb\_rating IS NOT NULL;

**BETWEEN(find anything in a range) Between 2 letters is not inclusive of 2nd letter, but both numbers are inclusive.-**

SELECT \* FROM table\_name WHERE column BETWEEN valueA AND valueB;

To get from A – I, but not J

SELECT \* FROM movies WHERE name BETWEEN ‘A’ AND ‘J’;

For years 1990 – 2000, including 2000-

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

**Queries cont.**

**AND(lets you combine multiple condition)-**

SELECT \* FROM table\_name WHERE columnA BETWEEN conditonA AND conditionB

AND column = condition

SELECT \* FROM movies WHERE year BETWEEN 1990 AND 1999

AND genre = ‘romance;

SELECT \* FROM movies WHERE year < 1985

AND genre = 'horror';

**OR(similar to AND) –**

SELECT \* FROM movies WHERE year > 2014

OR genre = 'action';

SELECT \* FROM movies WHERE genre = 'comedy'

OR genre = 'romance';

**Order By(sorts results in order numeric or alphabetically ASC for ascending, DESC for descending)-**

SELECT name, year FROM movies ORDER BY name;

SELECT name, year, imdb\_rating FROM movies ORDER BY imdb\_rating DESC;

**Limit(limits your results a certain number)-**

SELECT \* FROM movies LIMIT 10;

SELECT \* FROM movies ORDER BY imdb\_rating DESC LIMIT 3;

**Case(SQL’s way to handle if-then logic)-**

SELECT name,

CASE

WHEN imdb\_rating > 8 THEN 'Fantastic'

WHEN imdb\_rating > 6 THEN 'Poorly Received'

ELSE 'Avoid at All Costs'

END

FROM movies;

SELECT name,

CASE

WHEN genre = 'romance' OR genre = 'comedy'

THEN 'Chill'

ELSE 'Intense'

END AS 'Mood'

FROM movies;

**Code Challenge 1 –**

Find the number of girls who were named Lillian for the full span of time of the database.

Select only the year and number columns.

SELECT year, number FROM babies WHERE name = 'Lillian';

**Code Challenge 2 –**

Find 20 distinct names that start with 'S'.

Select only the name column

SELECT DISTINCT name FROM babies WHERE name LIKE 'S%' LIMIT 20;

**Code Challenge 3 –**

What are the top 10 names in 1880?

Select the name, gender, and number columns.

SELECT name, gender, number

FROM babies

WHERE year = 1880

ORDER BY number DESC

LIMIT 10;

**Code Challenge 4 –**

Suppose Abbi and Ilana want to have a fancy dinner date.

Return all the restaurants that are Italian and $$$.

Select all the columns.

SELECT \*

FROM nomnom

WHERE cuisine = 'Italian' and price = '$$$';

**Code Challenge 5 –**

Your coworker Trey can't remember the exact name of a restaurant he went to but he knows it contains the word 'meatball' in it.

Can you find it for him using a query?

Select all the columns.

SELECT \*

FROM nomnom

WHERE name LIKE '%meatball%';

**Code Challenge 6 -**

Some of the restaurants have not been inspected yet or are currently appealing their health grade score.

Find the restaurants that have empty health values.

Select all the columns.

SELECT \*

FROM nomnom

WHERE health IS NULL;

**Code Challenge 7 –**

Order the table by title (from A-Z).

Select only the title and publisher columns.

SELECT title, publisher

FROM news

ORDER BY title ASC;

**Code Challenge 8 –**

Which article names have the word 'bitcoin' in it?

Select all the columns.

SELECT \*

FROM news

WHERE title LIKE '%bitcoin%';

**Code Challenge 9 –**

The category column contains the article category:

'b' stands for Business

't' stands for Technology

What are the 20 *business* articles published most recently?

Select all the columns.

SELECT \*

FROM news

WHERE category = 'b'

ORDER BY timestamp DESC

LIMIT 20;

**Last Query challenge-**

**Start by getting a feel for the nomnomtable:**

SELECT \* FROM nomnom;

**What are the distinct neighborhoods?**

SELECT DISTINCT neighborhood FROM nomnom;

**What are the distinct cuisine types?**

SELECT DISTINCT cuisine FROM nomnom;

**Suppose we would like some Chinesetakeout.**

**What are our options?**

SELECT \* FROM nomnom WHERE cuisine = 'Chinese';

**Return all the restaurants with reviews of 4 and above.**

SELECT \* FROM nomnom WHERE review >= 4;

**Return all the restaurants that are Italian and $$$.**

SELECT \* FROM nomnom WHERE cuisine = 'Italian'

AND price = '$$$';

**Your coworker Trey can't remember the exact name of a restaurant he went to but he knows it contains the word 'meatball' in it.**

**Can you find it for him using a query?**

SELECT \* FROM nomnom WHERE name LIKE '%meatball%';

**Let's order delivery to the house!**

**Find all the close by spots in Midtown, Downtown or Chinatown**.

SELECT \* FROM nomnom

WHERE neighborhood = 'Midtown'

OR neighborhood = 'Downtown'

OR neighborhood = 'Cinatown';

**Find all the health grade pending restaurants (empty values).**

SELECT \* FROM nomnom WHERE health IS NULL;

**Create a Top 10 Restaurants Ranking based on reviews.**

SELECT \* FROM nomnom ORDER BY review DESC LIMIT 10;

**Use a CASE statement to change the rating system to:**

* **review > 4.5 is Extraordinary**
* **review > 4 is Excellent**
* **review > 3 is Good**
* **review > 2 is Fair**
* **Everything else is Poor**

**Don't forget to rename the new column!**

SELECT name,

CASE

WHEN review > 4.5 THEN 'Extraordinary'

WHEN review > 4 THEN 'Excellent'

WHEN review > 3 THEN 'Good'

WHEN review > 2 THEN 'Fair'

ELSE 'Poor'

END AS 'Review'

FROM nomnom;

**Find the full\_names and emails of the transactions listing 20252 as the zip code.**

SELECT full\_name, email, zip

FROM transaction\_data

WHERE zip = 20252;

**Finance has also noticed a number of pseudonyms associated with fraudulent transactions.**

**The fraudsters thought it would be funny to use 'Art Vandelay' for their full name or add a 'der' for their middle name.**

**Use a query to find the names and emails associated with these transactions.**

SELECT full\_name, email

FROM transaction\_data

WHERE full\_name = 'Art Vandelay'

OR full\_name LIKE '% der %';

**There are some irregularities in the IP addresses where transactions are originating from.**

**For example, any IP address beginning with '10.' is reserved for internal use.**

**We shouldn't see IP addresses like this accessing Reputable Company's service.**

**Find the ip\_addresses and emails listed with these transactions.**

SELECT ip\_address, email

FROM transaction\_data

WHERE ip\_address LIKE '10.%';

**Users are making fraudulent transactions using a temporary email address service. These services provide a**[**short-lived email**](https://en.wikipedia.org/wiki/Disposable_email_address)**that can be verified and then self-destructs.**

**Find the emails in transaction\_datawith 'temp\_email.com' as a domain.**

SELECT email

FROM transaction\_data

WHERE email LIKE '%temp\_email.com';

**The finance department is looking for a specific transaction. They know that the transaction occurred from an ip address starting with '120.' and their full name starts with 'John'.**

**Can you find the transaction?**

SELECT \*

FROM transaction\_data

WHERE ip\_address LIKE '120.%'

AND full\_name LIKE 'John%';

**Aggregates-**

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

Here is a quick preview of some important aggregates that we will cover in the next five exercises:

* 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
* ROUND(): round the values in the column

**Count**

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

COUNT() is a function that takes the name of a column as an argument and counts the number of non-empty values in that column.

This will count every row.

SELECT COUNT(\*)

FROM table\_name;

Use  WHERE clause in the previous query to count how many *free* apps are in the table.

SELECT COUNT(\*)

FROM fake\_apps

WHERE price = 0;

**Sum**

SQL makes it easy to add all values in a particular column using 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.

To find total downloads for all apps combined

SELECT SUM(downloads)

FROM fake\_apps;

**Max / Min**

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

MAX() takes the name of a column as an argument and returns the largest value in that column. Here, we returned the largest value in the downloads column.

MIN() works the same way but it does the exact opposite; it returns the smallest value.

To find how many downloads the most popular app has.

SELECT MAX(downloads)

FROM fake\_apps;

**Average**

SQL uses the AVG() function to quickly calculate the average value of a particular column.

To return the average number of downloads for an app in our database:

SELECT AVG(downloads)

FROM fake\_apps;

To find average price:

SELECT AVG(price)

FROM fake\_apps;

Round

By default, SQL tries to be as precise as possible without rounding. We can make the result table easier to read using the ROUND() function.

ROUND() function takes two arguments inside the parenthesis:

-a column name

-an integer

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

pass the column price and integer 0as arguments. SQL rounds the values in the column to 0 decimal places in the output.

SELECT name, ROUND(price, 0)

FROM fake\_apps;

SELECT ROUND(AVG(price), 2)

FROM fake\_apps;

**Group By**

Oftentimes, we will want to calculate an aggregate for data with certain characteristics.

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 WHEREstatements, but before ORDER BY or LIMIT.

To get app count by price:

SELECT price, COUNT(\*)

FROM fake\_apps

GROUP BY price;

To get app count by price and with over 20k downloads:

SELECT price, COUNT(\*)

FROM fake\_apps

WHERE downloads > 20000

GROUP BY price;

To get the total number of downloads by category:

SELECT category, SUM(downloads)

FROM fake\_apps

GROUP BY category;

To find out how many books you have by category, then by price in that category:

SELECT category,

price,

AVG(downloads)

FROM fake\_apps

GROUP BY 1, 2;

**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.

For instance, imagine that we want to see how many movies of different genres were produced each year, but we only care about years and genres with at least 10 movies.

We can't use WHERE here because we don't want to filter the rows; we want to filter groups.

This is where HAVING comes in.

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

SELECT year,

genre,

COUNT(name)

FROM movies

GROUP BY 1, 2

HAVING COUNT(name) > 10;

**Aggregates challenges-**

**Calculate the total number of companies in the table.**

SELECT COUNT(\*)

FROM startups;

**We want to know the total value of all companies in this table.**

**Calculate this by getting the SUM() of the valuation column.**

SELECT SUM(valuation)

FROM startups;

**What is the highest amount raised by a startup?**

**Return the maximum amount of money raised.**

SELECT MAX(raised)

FROM startups;

**Edit the query so that it returns the maximum amount of money raised, during 'Seed' stage.**

SELECT MAX(raised)

FROM startups

WHERE stage = 'Seed';

**In what year was the oldest company on the list founded?**

SELECT MIN(founded)

FROM startups;

**Return the average valuation.**

SELECT AVG(valuation)

FROM startups;

**Return the average valuation, in each category.**

SELECT category, AVG(valuation)

FROM startups

GROUP BY category;

**Return the average valuation, in each category.**

**Round the averages to two decimal places**.

SELECT category, ROUND(AVG(valuation), 2)

FROM startups

GROUP BY category;

**Return the average valuation, in each category.**

**Round the averages to two decimal places.**

**Lastly, order the list from highest averages to lowest.**

SELECT category, ROUND(AVG(valuation), 2)

FROM startups

GROUP BY 1

ORDER BY 2 DESC;

**First, return the name of each category with the total number of companies that belong to it.**

SELECT category, COUNT(\*)

FROM startups

GROUP BY category;

**Next, filter the result to only include categories that have more than three companies in them. What are the most competitive markets?**

SELECT category, COUNT(\*)

FROM startups

GROUP BY category

HAVING COUNT(\*) > 3

ORDER BY 2 DESC;

**What is the average size of a startup in each location?**

SELECT location, AVG(employees)

FROM startups

GROUP BY location;

**What is the average size of a startup in each location, with average sizes above 500?**

SELECT location, AVG(employees)

FROM startups

GROUP BY location

HAVING AVG(employees) > 500;

**Working with multiple tables-**

**Combining tables manually is time-consuming. Luckily, SQL gives us an easy sequence for this: it's called a JOIN.**

SELECT \*

FROM orders

JOIN subscriptions

ON orders.subscription\_id =

subscriptions.subscription\_id;

SELECT \*

FROM orders

JOIN subscriptions

ON orders.subscription\_id =

subscriptions.subscription\_id

WHERE subscriptions.description =

'Fashion Magazine';

**Inner Joins-**

SELECT COUNT(\*)

FROM newspaper;

SELECT COUNT(\*)

FROM online;

Can be done like this:

SELECT COUNT(\*)

FROM newspaper

JOIN online

ON newspaper.id = online.id;

# Left Joins -

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

SELECT \*

FROM table1

LEFT JOIN table2

ON table1.c2 = table2.c2;

1. The first line selects all columns from both tables.
2. The second line selects table1 (the "left" table).
3. The third line performs a LEFT JOIN on table2 (the "right" table).
4. The fourth line tells SQL how to perform the join (by looking for matching values in column c2).

SELECT \*

FROM newspaper

LEFT JOIN online

ON newspaper.id = online.id;

SELECT \*

FROM newspaper

LEFT JOIN online

ON newspaper.id = online.id

WHERE online.id IS NULL;

**Primary Key vs Foreign Key**

The PRIMARY KEY uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only one primary key, which may consist of single or multiple fields.

A FOREIGN KEY is a key used to link two tables together.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

SELECT \*

FROM classes

JOIN students

ON classes.id = students.class\_id;

Primary Key = class\_id, foreign key = students.class\_id

**Cross Join –**

The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with CROSS JOIN.This kind of result is called as Cartesian Product.

If WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.

An alternative way of achieving the same result is to use column names separated by commas after SELECT and mentioning the table names involved, after a FROM clause.

SELECT \*

FROM newspaper

CROSS JOIN months;

SELECT \*

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

AND end\_month >= month;

SELECT month,

COUNT(\*)

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

AND end\_month >= month

GROUP BY month;

**Union-**

The UNION operator is used to combine the result-set of two or more SELECT statements.

Each SELECT statement within UNION must have the same number of columns

The columns must also have similar data types

The columns in each SELECT statement must also be in the same order

SELECT \*

FROM newspaper

UNION

SELECT \*

FROM online;

**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

We can then go on to do whatever we want with this temporary table (such as join the temporary table with another table)

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.

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 customers.customer\_id = previous\_query.customer\_id;

**Join Review**

In this lesson, we learned about relationships between tables in relational databases and how to query information from multiple tables using SQL.

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.