Lesson 1

Definition:
Databases are systems that allow users to store and organize data.
They are useful when dealing with large amount of data

Databases have a wide variety of users

-Analysts
    -marketing
    -business
    -Sales
-Technical
    -Data Scientist
    -Software Engineers
    -Web Developers

Section 5:
pgAdmin is a great tool and will save you a lot of time for these types of tasks

Create a Database with pgAdmin
Restore a Database
Delete a Database

SQL syntax fundamentals

SELECT statement

- One of the most common tasks is to query data from tables by using the SELECT statement.
- It has many clauses that you can combine to form a powerful query.

Let's start with a basic form of the SELECT statement to query data from a table.

Syntax:

SELECT column1, column2,... **FROM** table_name

1. First, you specify a list of columns in the table from which you want to query data in the SELECT clause. You use a comma between each column in case you want to query data from multiple columns.
2. If you want to query data from all columns, you can use an asterisk (*) as the shorthand for all columns. (Not good practice, slow down application. Mention specific column name)

3. Second, you indicate the table name after the FROM keyword

Challenge 1:

```
SELECT first_name, last_name, email FROM customer;
```

SELECT with DISTINCT (SELECT DISTINCT Statement)

- In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.
- The DISTINCT keyword can be used to return only distinct (different) values.

Syntax:

```
SELECT DISTINCT column1, column2 FROM table_name ;
```

Challenge 2:

```
SELECT DISTINCT rating FROM film;
```

SELECT with WHERE (SELECT WHERE Statement)

- What if you want to query just particular rows from a table?
- In this case, you need to use the WHERE clause in the SELECT statement.
- The WHERE clause appears right after the FROM clause of the SELECT statement.
- The conditions are used to filter the rows returned from the SELECT statement.
- PostgreSQL provides you with various standard operators to construct the conditions.

Syntax :

```
SELECT column_1, column_2, …., column_N FROM table_name WHERE conditions;
```

EX_1:

- If you want to get all customers whose first name are Jamie, you can use the WHERE clause with the equal (=) operator as follows:
- SELECT last_name, first_name FROM customer WHERE first_name='Jamie';
- SELECT last_name, first_name FROM customer WHERE first_name='Jamie' AND last_name='Rice';

EX_2:

- If you want to know who paid the rental with amount is either less than 1 USD or greater than 8 USD, you can use the following query with OR operator:
- SELECT customer_id, amount, payment_date FROM payment WHERE amount <= 1 OR amount >= 8;

Challenge 3.1:

```
SELECT email FROM customer
WHERE first_name='Nancy' AND last_name='Thomas';
```

Answer: "nancy.thomas@sakilacustomer.org"

Challenge 3.2:
SELECT description FROM film
WHERE title='Outlaw Hanky';
Answer: "A Thoughtful Story of a Astronaut And a Composer who must Conquer a Dog in The Sahara Desert"

Challenge 3.3:
SELECT phone FROM address
WHERE address='259 Ipoh Drive';
Answer: "419009857119"

COUNT function statement

The COUNT function returns the number of input rows that match a specific condition of a query

The COUNT(*) function returns the number of rows returned by a SELECT clause
When you apply the COUNT(*) to the entire table, PostgreSQL scans table sequentially

You can also specify a specific column count for readability
Syntax: SELECT COUNT(column) FROM table

Similar to the COUNT(*) function, the COUNT(column) function returns the number of rows returned by a SELECT clause. (However, it doesn't consider NULL values in the column.)

FInally, we can use COUNT with DISTINCT, for example:
Syntax: SELECT COUNT(DISTINCT column) FROM table;
Example:
SELECT COUNT (*) FROM payment;
SELECT COUNT(DISTINCT (amount)) FROM payment;

LIMIT STATEMENT

- LIMIT allows you to limit the number of rows you get back after a query
- Useful when wanting to get all columns but not all rows
- Goes at the end of a query

Ex:
SELECT * FROM customer
Limit 5; #Only returns the first five rows of the query

ORDER BY STATEMENT

- When you query data from a table, PostgreSQL returns the rows in the order that they were inserted into the table.
- In order to sort the result set, you use the ORDER BY clause in the SELECT statement.

- The ORDER BY clause allows you to sort the rows returned from the SELECT statement in ascending or descending order based on criteria specified.
- Specify the column that you want to sort in the ORDER BY clause. If you sort the result set by multiple columns, use a comma to separate between two columns.
- Use ASC to sort the result set in ascending order and DESC to sort the result set in descending order.
- If you leave it blank, the ORDER BY clause will use ASC by default.
- Syntax: **SELECT** column_1, column_2 **FROM** table_name **ORDER BY** column_1 **ASC / DESC**;
- Example: **SELECT** first_name, last_name **FROM** customer **ORDER BY** first_name **ASC,** last_name **DESC**;
- Example; **SELECT** first_name **FROM** customer **ORDER BY** last_name;

Challenge ORDER BY:

```
SELECT customer_id FROM payment
ORDER BY amount DESC LIMIT 10;
```

Challenge:

```
SELECT title FROM film
ORDER BY film_id ASC Limit 5;
```

BETWEEN STATEMENT

- We use the BETWEEN operator to match a value against a range of values. For example: **value BETWEEN low AND high;**
- If the value is greater than or equal to the low value and less than or equal to the high value, the expression returns true, or vice versa.
- We can rewrite the BETWEEN operator by using the greater than or equal (>=) or less than or equal (<=) operators as the following statement:
- **Value >= low and value <= high;**
- If we want to check if a value is out of range, we use the NOT BETWEEN operator as follows:
- **Value NOT BETWEEN low AND high;**
- **Value < low OR value > high;**
- Example: SELECT customer_id, amount FROM payment WHERE amount NOT BETWEEN 6 AND 9;
- Example: SELECT amount, payment_date FROM payment WHERE payment_date BETWEEN ‘2007-02-07’ AND ‘2007-02-15’;

IN STATEMENT

- You use the IN operator with the WHERE clause to check if a value matches any value in a list of values.
- The syntax of the IN operator is as follows:
- **Value IN(value1, value2, …)**
- The expression returns true if the value matches any value in the list i.e., value1, value2, etc. The list of values is not limited to a list of numbers or strings but also a result set of a SELECT statement as shown in the following query:
- **Value IN** (**SELECT** value **FROM** tbl_name)
- Just like with BETWEEN, you can use NOT to adjust an IN statement (NOT IN)
- Example:
  SELECT customer_id, rental_id, return_date **FROM** rental **WHERE** customer_id **IN(**1,2**) ORDER BY** return_date **DESC**;

LIKE STATEMENT

- The query returns rows whose values in the first name column begin with Jen and may be followed by any sequence of characters. This technique is called pattern matching.
- **Syntax: SELECT** first_name, last_name **FROM** customer **WHERE** first_name **LIKE** 'Jen%';
- You construct a pattern by combining a string with wildcard characters and use the LIKE or NOT LIKE operator to find the matches.
  - Percent (%) for matching any sequence of characters.
  - Underscore (_) for matching any single character.

- **Example: SELECT** first_name, last_name **FROM** customer **WHERE** first_name **LIKE** '**Jen%**'; (any first name begins with 'Jen')
- **Example: SELECT** first_name, last_name **FROM** customer **WHERE** first_name **LIKE** '**%y**'; (any first name ends with 'y')
- **Example: SELECT** first_name, last_name **FROM** customer **WHERE** first_name **LIKE** '**%er%**'; (any first name that contains 'er' somewhere)
- **Example: SELECT** first_name, last_name **FROM** customer **WHERE** first_name **LIKE** '_her%'; (any first name that the underscore placeholder can be anything + 'her' + anything)

- **Example: SELECT** first_name, last_name **FROM** customer **WHERE** first_name **NOT LIKE** ‘Jen%’;
- **Example: SELECT** first_name, last_name **FROM** customer **WHERE** first_name **ILIKE** ‘BaR%’; (ILIKE means we wants anything string requested to be case insensitive)

General Challenge:
Q1: How many payment transactions were greater than $5.00?
A1: SELECT COUNT(amount) FROM payment
WHERE amount BETWEEN '5.99' AND '9.99';

Official ANS: SELECT COUNT(amount) FROM payment WHERE amount >5;

Q2: How many actors have a first name that starts with the letter P?
A2: SELECT COUNT(first_name) FROM actor
WHERE first_name LIKE 'P%';

Official ANS: SELECT COUNT(*) FROM actor WHERE first_name LIKE ‘P%’;

Q3: How many unique districts are our customers from?
A3: SELECT COUNT (DISTINCT (district)) FROM address;

Official ANS: Same as above

Q4: Retrieve the list of names for those distinct districts from the previous question.
A4: SELECT DISTINCT district FROM address;

Q5: How many films have a rating of R and a replacement cost between $5 and $15?
A5: SELECT COUNT(*) FROM film
WHERE rating='R' AND replacement_cost BETWEEN '5' AND '15';

Official ANS: SELECT COUNT(*) FROM film
WHERE rating='R' AND replacement_cost BETWEEN 5 AND 15;

Q6: How many films have the word Truman somewhere in the title?
A6: SELECT COUNT(*) FROM film
WHERE title LIKE 'Truman%';

Official ANS: SELECT COUNT(*) FROM film
WHERE title LIKE '%Truman%';

MIN MAX AVG SUM Aggregate Functions

Example:
SELECT AVG(amount**) FROM** payment;
SELECT ROUND (AVG(amount**), 2) FROM** payment;
SELECT MIN(amount**) FROM** payment;
SELECT ROUND(SUM(amount**), 1) FROM** payment;

GROUP BY STATEMENT

- The **GROUP BY** clause divides the rows and returned from the SELECT statement into groups.
- For each group, you can apply an aggregate function, for example:
  - Calculating the sum of items
  - Count the number of items in the groups.
- GROUP BY Syntax: **SELECT** column_1, **aggregate_function(**column_2**) FROM** table_name **GROUP BY** column_1;
- Example: **SELECT** customer_id **FROM** payment **GROUP BY** customer_id; (A lot of same customers with multiple payments)
- Example: **SELECT** staff_id, **COUNT(**payment_id**) FROM** payment **GROUP BY** staff_id;
  - Select staff_id column instances and count the transaction times from each staff_id when grouping them by staff_id
- Example: **SELECT** rating, **AVG(**rental_rate**) FROM** film **GROUP BY** rating;

GROUP BY Challenge:
Q1: We have two staff members with Staff IDs 1 and 2. We want to give a bonus to the staff member that handled the most payments.
How many payments did each staff member handle? And how much was the total amount processed by each staff member?

A1: For how many? For how much total amount of payments each staff?

SELECT count(staff_id) FROM payment
WHERE staff_id ='1'; ----> 7292 payments
SELECT sum(amount) FROM payment
WHERE staff_id ='1'; -----> total amount of payment: 30252.12

SELECT count(staff_id) FROM payment
WHERE staff_id ='2'; -----> 7304 payments
SELECT sum(amount) FROM payment
WHERE staff_id ='2'; ------> total amount of payment: 31059.92

Improved answers:
SELECT staff_id, sum(amount) FROM payment
GROUP BY staff_id;

SELECT staff_id, count(staff_id) FROM payment
GROUP BY staff_id;

Q2: Corporate headquarters is auditing our store! They want to know the average replacement cost of movies by rating.
For example, R rated movies have an average replacement cost of 20.23 USD.

Answer:
SELECT rating, AVG(replacement_cost) FROM film
GROUP BY rating;

Q3: We want to send coupons to the 5 customers who have spent the most amount of money. Get me the customer ids of the top 5 spenders.

Answer:
SELECT customer_id, sum(amount) from payment
GROUP BY customer_id ORDER BY sum(amount) DESC LIMIT 5;

HAVING STATEMENT
We often use the HAVING clause in conjunction with the GROUP BY clause to filter group rows that do not satisfy a specified condition.

HAVING Syntax

SELECT column_1, aggregate_function(column_2) **FROM** table_name **GROUP BY** column_1 **HAVING** condition;

The HAVING clause sets the condition for group rows created by the GROUP BY clause after the GROUP BY clause applies while the WHERE clause sets the condition for the individual rows before GROUP BY clause applies. This is the main difference between the HAVINg and WHERE clauses.

Example:
SELECT customer_id, SUM(amount) **FROM** payment
GROUP BY customer_id
HAVING SUM(amount) >200;

SELECT store_id, COUNT(customer_id) **FROM** customer
GROUP BY store_id
HAVING COUNT(customer_id) >300;

SELECT rating, rental_rate FROM film
WHERE rating IN ('R', 'G','PG')
GROUP BY rating
HAVING AVG(rental_rate) < 3;

HAVING Challenge
We want to know what customers are eligible for our platinum credit card. The requirements are that the customer has at least a total of 40 transaction payments. What customers (by customer_id) are eligible for the credit card?

SELECT customer_id, COUNT(amount) **FROM** payment
GROUP BY customer_id
HAVING COUNT(amount)>= 40;

When grouped by rating, what movie ratings have an average rental duration of more than 5 days?

SELECT rating, avg(rental_duration) FROM film
GROUP BY rating
HAVING avg(rental_duration) > 5;

Assessment Test 1

1. Return the customer IDs of customers who have spent at least 110 USD with the staff member who has an ID of 2.

   Answer:
   SELECT customer_id, sum(amount) **FROM** payment
   GROUP BY customer_id
   HAVING sum(amount)>= 110 **AND** staff_id = 2;

2. How many films begin with the letter J?

   Answer:
   SELECT count(title) **FROM** film
   WHERE title LIKE 'J%';

   20 films

3. What customer has the highest customer ID number whose name starts with an ‘E’ and has an address ID lower than 500?

   Answer: Eddie Tomlin

   SELECT first_name, last_name, customer_id from customer
   WHERE first_name LIKE 'E%' AND customer_id < 500;

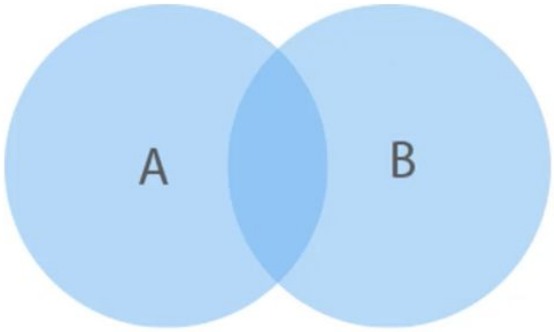
JOIN Statement

- So far, you have learned how to select data from a table, choosing which columns and rows you want, and how to sort the result set in a particular order.
- It is time to move to one of the most important concepts in the database called joining that allows you to relate data in one table to the data in other tables.
- There are several kinds of Joins including INNER JOIN, OUTER JOIN and Self -join.

Inner Join:

(Inner Join produces only the set of records that match in both Table A and Table B.)

- Suppose you want to get data from two tables named A and B.

- The B table has the fka field that relates to the primary key of the A table.
- To get data from both tables, you use the INNER JOIN clause in the SELECT statement as follows:
    - EX: **SELECT** A.pka, A.c1, B.pkb, B.c2 **FROM** A **INNER JOIN** B **ON** A.pka=B.fka;
    - First, you specify the column in both tables from which you want to select data in the SELECT clause
    - Second, you specify the main table i.e., A in the FROM clause.
    - Third, you specify the table that the main table joins to i.e., B in the INNER JOIN clause. In addition, you put a join condition after the ON keyword i.e, A.pka = B.fka
- For each row in the A table, PostgreSQL scans the B table to check if there is any row that matches the condition
- i.e., A.pka = B.fka
- If it finds a match, it combines columns of both rows into one row and add the combined row to the returned result set.
- Sometimes A and B tables have the same column name so we have to refer to the column as table_name.column_name to avoid ambiguity.
- In case the name of the table is long, you can use a table alias e.g., tbl and refer to the column as tbl.column_name.
- The Inner join clause returns rows in A table that have the corresponding row in the B table.
- 

PostgreSQL INNER JOIN
- **SELECT** customer.customer_id, first_name, last_name, email, amount, payment_date **FROM** customer **INNER JOIN** payment **ON** payment.customer_id = customer.customer_id **ORDER BY** customer.customer_id;
- **SELECT** customer.customer_id, first_name, last_name, email, amount, payment_date **FROM** customer **INNER JOIN** payment **ON** payment.customer_id = customer.customer_id **WHERE** first_name LIKE 'A%';
- **SELECT** title, name **AS** movie_language **FROM** film **JOIN** language **AS** lan **ON** lan.language.id = film.language_id;

- **SELECT** title, COUNT(title) **AS** copies_at_store1**FROM** inventory **INNER JOIN** film **ON** inventory.film_id = film.film_id **WHERE** store_id = 1 **GROUP BY** title;
- **SELECT** title, name movie_language **FROM** film **JOIN** language lan **ON** lan.language_id = film.language_id; (abbreviate “AS”, “INNER”, “AS”)

As Statement

- “As” allows us to rename columns or table selections with an alias
- Examples:
    - **SELECT** payment_id **AS** my_payment_column **FROM** payment;
    - **SELECT** customer_id, SUM(amount) **AS** total_spent **FROM** payment **GROUP BY** customer_id;

FULL OUTER JOIN:

- Full outer join produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null.

Original Tables

```
id name       id  name
-- ----       --  ----
1  Pirate     1   Rutabaga
2  Monkey     2   Pirate
3  Ninja      3   Darth Vader
4  Spaghetti  4   Ninja
```

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name

id    name       id    name
--    ----       --    ----
1     Pirate     2     Pirate
2     Monkey     null  null
3     Ninja      4     Ninja
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```

LEFT OUTER JOIN:

- Left outer join produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null.


```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name

id  name       id    name
--  ----       --    ----
1   Pirate     2     Pirate
2   Monkey     null  null
3   Ninja      4     Ninja
4   Spaghetti  null  null
```

LEFT/ RIGHT OUTER JOIN with WHERE

- To produce the set of records only in Table A, but not in Table B, we perform the same left outer join, then exclude the records we dont want from the right side via a where clause.
- **SELECT** * **FROM** TableA **LEFT/RIGHT OUTER JOIN** TableB **on** TableA.name = TableB.name **WHERE** TableB.id **IS** null

FULL OUTER JOIN with WHERE

- To produce the set of records unique to Table A and Table B, we perform the same full outer join, then exclude the records we dont want from both sides via a Where clause.

Original Tables

id	name	id	name
1	Pirate	1	Rutabaga
2	Monkey	2	Pirate
3	Ninja	3	Darth Vader
4	Spaghetti	4	Ninja

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
OR TableB.id IS null
```

id	name	id	name
2	Monkey	null	null
4	Spaghetti	null	null
null	null	1	Rutabaga
null	null	3	Darth Vader

UNION:

- The UNION operator combines result sets of two or more SELECT statements into a single result set.
- The following illustrates the syntax of the union operator that combines result sets from two queries:
- Ex: **SELECT** column_ 1, column 2 **FROM** tbl_name_1 **UNION SELECT** column_1, column_2 **FROM** tbl_name_2;
- The UNION operator removes all duplicate rows unless the UNION ALL is used.
- The UNION operator may place the rows in the first query before, after or between the rows in the result set of the second query.
- To sort the rows in the combined result set by a specified column, you use the ORDER BY clause.
- We often use the UNION operator to combine data from similar tables that are not perfectly normalized
- Those tables are often found in the reporting or data warehouse system.
- Using UNION will delete duplicate row. Using UNION ALL will keep them.

ADVANCED

TimeStamp:

- SQL allows us to use the timestamp data type to retain time information.

Extract Function:

- The PostgreSQL extract function extracts parts from a date.
    - Extract (unit from date)

- We can extract many types of time-based information

https://www.postgresql.org/docs/9.1/functions-datetime.html

Exs:

- SELECT customer_id, extract(day from payment_date) AS day FROM payment;
- SELECT SUM(amount) AS total, extract (month from payment_date) As month From payment GROUP BY month ORDER BY total DESC LIMIT 1;

Mathematical Functions

- **SQL comes with a lot of mathematical operators built-in that are very useful for numeric column types**
- **https://www.postgresql.org/docs/9.5/functions-math.html**

String Functions

- **Ex:** SELECT first_name || ' ' || last_name AS full_name FROM customer;
- SELECT first_name, char_length(first_name) FROM customer;
- SELECT lower(first_name) FROM customer;

Subquery

- A subquery allows us to use multiple SELECT statements, where we basically have a query within a query
- Suppose we want to find the films whose rental rate is higher than the average rental rate.
- We can do it in two steps;
    - Find the average rental rate by using the SELECT statement and average function (avg)
    - Use the result of the first query in the second SELECT statement to find the films that we want.
    - SELECT AVG(rental_rate) FROM film;
    - SELECT title, rental_rate FROM film WHERE rental_rate > 2.98;
- The code is not so elegant, which requires two steps.
- We want a way to pass the result of the first query to the second query in one query.
- The solution is to use a subquery.
- A subquery is a query nested inside another query
- To construct a subquery, we put the second query in brackets and use it in the WHERE clause as an expression.
    - **IMPROVED:** SELECT film_id, title, rental_rate FROM film WHERE rental_rate > (SELECT AVG(rental_rate) FROM film);
    -

- SELECT film_id, title FROM film WHERE film_id IN (SELECT inventory.film_id FROM rental INNER JOIN inventory ON inventory.inventory_id = rental.inventory_id WHERE return_date BETWEEN '2005-05-29' AND '2005-05-30');

SELF JOIN:

- A special case that you join table to itself, which is known as self-join
- You use self join when you want to combine rows with other rows in the same table.
- To perform the self join operation, you must use a table alias to help SQL distinguish the left table from the right table of the same table.
- **SELECT** e1.employee_name **FROM** employee **AS** e1, employee **AS** e2 **WHERE** e1.employee_location = e2.employee_location **AND** e2.employee_name =”joe”;
- This query will return the names Joe and Jack since Jack is the only other person who lives in New York like Joe.
- Generally, queries that refer to the same table can be greatly simplified by re-writing the queries as self joins.
- There is definitely a performance benefit for this as well.
- **SELECT** a.first_name, a.last_name, b.first_name, b.last_name **FROM** customer **AS** a, customer **AS** b **WHERE** a.first_name = b.last_name;
- **SELECT** a.customer_id, a.first_name, a.last_name, b.customer_id, b.first_name, b.last_name **FROM** customer **AS** a **LEFT JOIN** customer **AS** b **ON** a.first_name = b.last_name **ORDER BY** a.customer_id;
- INTERVIEW QUESTIONS HEAVILY HAPPENED HERE!! (Manager employee Self Join)
-