Mohammad Idrees Bhat

Tech Skills Trainer | AI/ML Consultant

Advanced SQL

Joins, subqueries, window functions.

Skills Covered

1. Complex SQL querying techniques.
2. Advanced joins, including self and recursive joins.
3. Writing and optimizing subqueries (simple and correlated).
4. Use of window functions for analytical queries.
5. Efficient data retrieval with partitioning and window functions.

Learning Outcomes

1. Students will be able to perform complex joins in SQL, including self-joins and recursive joins.

2. Students will write and use subqueries in various SQL clauses (SELECT, WHERE, FROM).

3. Students will understand and apply window functions like ROW_NUMBER(), RANK(), and LAG().

4. Students will develop skills to query and analyze large datasets efficiently using advanced SQL techniques.

5. Students will gain the ability to design and optimize complex SQL queries for real-world applications.

AGENDA

1. SQL Basics - SQL Joins.

2. Correlated subqueries in SELECT, WHERE, and FROM clauses.

3. Window functions, Cumulative sums and ranking

5. Hands-on examples using real-world data.

What's something about Nepal you find yourself explaining to your American friends?

1. SQL Basics - SQL Joins.

Introduction to Data and database management

A **database** is an organized collection of data, typically stored electronically. It allows efficient data management and retrieval, ensuring data is stored in a structured and easily accessible way.

Databases allow for efficient data storage, retrieval, and management, enabling users to work with large volumes of data effectively.

Ways to Work with Database Data and Tables

1. **Inserting Data**: Add new data into a table using SQL commands like `INSERT`.
2. **Retrieving Data**: Query and retrieve data using `SELECT` statements, applying filters and sorting.
3. **Updating Data**: Modify existing data with the `UPDATE` command.
4. **Deleting Data**: Remove data from tables using `DELETE`.
5. **Joining Tables**: Combine data from multiple tables using `JOIN` operations to gather comprehensive insights.

Relational Databases (RDBMS)

A **relational database** organizes data into tables (also known as relations) where each row represents a record, and each column represents an attribute of the data.

These databases use **SQL (Structured Query Language)** for managing and querying data and are based on the concept of relationships between tables.

Other Types of Databases

1. **NoSQL Databases**: These databases handle large amounts of unstructured or semi-structured data, like key-value stores (e.g., MongoDB, Cassandra).
2. **Object-Oriented Databases**: Stores data as objects, similar to object-oriented programming (e.g., db4o, ObjectDB).

3. **Graph Databases**: Focus on relationships between entities and represent them as nodes connected by edges (e.g., Neo4j).
4. **Column-Oriented Databases**: Store data in columns rather than rows for fast retrieval in analytical queries (e.g., Apache Cassandra, HBase).

Versions of SQL

1. **MySQL**: Open-source, widely used for web applications due to its speed and flexibility.
2. **PostgreSQL**: An open-source relational database known for advanced features and strict adherence to SQL standards.
3. **SQLite**: A lightweight, serverless database often embedded within applications for local data storage.
4. **SQL Server**: Microsoft's relational database with deep integration into Windows environments.
5. **Oracle SQL**: A robust, enterprise-grade SQL used in large-scale applications with powerful tools for transaction management and optimization.

Why PostgreSQL?

We are using **PostgreSQL** here for its balance of advanced features, reliability, and ease of use. (considering that we need to be beginner friendly too)

It's a powerful open-source relational database that adheres strictly to SQL standards, making it great for learning complex queries like **joins**, **subqueries**, and **window functions**.

Unlike other database systems like **Oracle** or **SQL Server**, which can be heavy to install and configure, PostgreSQL is lightweight and straightforward. It offers advanced capabilities without overwhelming students with unnecessary complexity, making it ideal for learning and later scaling to more complex systems.

Additionally, **SQLite** may be too limited for some advanced features (like certain window functions), while PostgreSQL handles them efficiently. Which is why we didn't choose to continue working with SQLite.

Databases and Tables

1. Create a Database

First, we need to create a database in PostgreSQL that we will use for all our queries throughout the class. For this example, let's create a fun and engaging **Movie Database** where we'll manage information about movies, actors, and directors.

```
CREATE DATABASE moviedb;
```

To start using the database, we need to connect to it:

```
\c moviedb;
```

2. Creating Tables

Next, we'll create some tables for the Movie Database:

- movies: Stores details about different movies.
- actors: Stores information about actors.
- directors: Stores information about directors.
- movie_cast: Links movies to actors.
- movie_directors: Links movies to directors.

Movies table

```
CREATE TABLE movies (
    movie_id SERIAL PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    release_year INT,
    genre VARCHAR(50)
);
```

a. Primary Key

A **primary key** is a column (or set of columns) in a table that uniquely identifies each row in that table. It ensures that no two rows have the same value for this column, maintaining data integrity. A primary key cannot have `NULL` values.

Example:

In the `movies` table, `movie_id` is the primary key.

Each movie has a unique `movie_id` to identify it.

Actor table

```
CREATE TABLE actors (
    actor_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    birthdate DATE
);
```

b. Foreign Key

A **foreign key** is a column (or set of columns) in one table that refers to the **primary key** in another table.

It establishes a relationship between two tables, enforcing referential integrity — meaning the foreign key value must correspond to an existing primary key value in the referenced table.

Example:

In the `movie_cast table`, `movie_id` and `actor_id` are foreign keys, referring to the `movie_id` in the `movies` table and the `actor_id` in the `actors` table.

This ensures that the movie and actor exist in their respective tables before we can link them in `movie_cast`.

It's like saying, "I can only use movie IDs that are already in the movies table."

movie_cast table

```
CREATE TABLE movie_cast (
    movie_id INT REFERENCES movies(movie_id),
    actor_id INT REFERENCES actors(actor_id),
    role VARCHAR(100),
    PRIMARY KEY (movie_id, actor_id)
);
```

Primary Key ensures that each record in a table is unique and can be retrieved efficiently.

Foreign Key establishes relationships between tables, ensuring that data is consistently linked and enforcing referential integrity.

Directors table

```
CREATE TABLE directors (
    director_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    birthdate DATE
);
```

movie_directors table

```
CREATE TABLE movie_directors (
    movie_id INT REFERENCES movies(movie_id),
    director_id INT REFERENCES directors(director_id),
    PRIMARY KEY (movie_id, director_id)
);
```

Basic SQL Queries - Inserting, Updating, Deleting

1. Inserting Data into a Table

To add data into a table, we use the `INSERT INTO` statement.

This statement allows us to add a new record (row) to the table. You can either specify the values for all columns or only for specific ones.

Syntax for Insert:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Example: Insert data into the `movies` table

```
INSERT INTO movies (title, release_year, genre)
VALUES ('Inception', 2010, 'Sci-Fi'),
       ('The Dark Knight', 2008, 'Action'),
       ('The Matrix', 1999, 'Action');
```

Similar method can be used for all the other tables:

```
INSERT INTO actors (first_name, last_name, birthdate)
VALUES ('Leonardo', 'DiCaprio', '1974-11-11'),
       ('Christian', 'Bale', '1974-01-30'),
       ('Keanu', 'Reeves', '1964-09-02');

INSERT INTO directors (first_name, last_name, birthdate)
VALUES ('Christopher', 'Nolan', '1970-07-30'),
       ('Wachowski', 'Brothers', '1967-06-21');

-- Establishing the roles of each actor
--    Leonardo DiCaprio played the role of Cobb in the movie Inception.
--    Christian Bale played the role of Bruce Wayne in The Dark Knight.
--    Keanu Reeves played Neo in The Matrix.

INSERT INTO movie_cast (movie_id, actor_id, role)
VALUES (1, 1, 'Cobb'),
       (2, 2, 'Bruce Wayne'),
       (3, 3, 'Neo');

INSERT INTO movie_directors (movie_id, director_id)
VALUES (1, 1), -- Inception by Christopher Nolan
       (2, 1), -- The Dark Knight by Christopher Nolan
       (3, 2); -- The Matrix by Wachowski Brothers
```

2. Updating Data in a Table

To modify existing records, we use the `UPDATE` statement. This allows us to change the values of one or more columns for a specific row or set of rows in a table.

Syntax for Update:

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;
```

Example: Let's update the `movies` table. Suppose we need to change the release year of **The Matrix** from 1999 to 2000.

```
UPDATE movies
SET release_year = 2000
WHERE title = 'The Matrix';
```

`WHERE` keyword is extremely important in SQL, especially when performing operations like **UPDATE** and **DELETE**.

`WHERE` filters the rows that the SQL command affects.

Without `WHERE`, the operation will affect **all rows** in the table, which is usually not desirable.

In the example earlier if we do not use `WHERE`, that statement will update the release_year for every record in the movies table to 2023.

3. **Deleting Data from a Table**

To remove records from a table, we use the `DELETE` statement. This statement removes one or more rows from the table based on a condition.

Syntax for Delete:

```
DELETE FROM table_name
WHERE condition;
```

Example: Let's say we want to delete the movie The Dark Knight from the `movies` table

```
DELETE FROM movies
WHERE title = 'The Dark Knight';
```

4. **Modifying Table Structure (Adding Keys) [Advanced/Optional]**

If you forgot to add a primary key or foreign key when creating a table, you can modify the table structure using the `ALTER TABLE` statement.

Syntax for Alter (Primary key):

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name PRIMARY KEY (column_name);
```

Example: Suppose we forgot to add a primary key to the actors table on the actor_id column. We can add it using the following query:

```
ALTER TABLE actors
ADD CONSTRAINT pk_actor_id PRIMARY KEY (actor_id);
```

Syntax for Alter (Foreign key):

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name FOREIGN KEY (column_name) REFERENCES
referenced_table(referenced_column);
```

Example: Let's say the movie_cast table doesn't have the foreign key constraints linking movie_id and actor_id to the respective tables. You can add them as follows:

```
ALTER TABLE movie_cast
ADD CONSTRAINT fk_movie FOREIGN KEY (movie_id) REFERENCES
movies(movie_id),
ADD CONSTRAINT fk_actor FOREIGN KEY (actor_id) REFERENCES
actors(actor_id);
```

5. **Aggregate Functions in SQL [Advanced/Optional]**

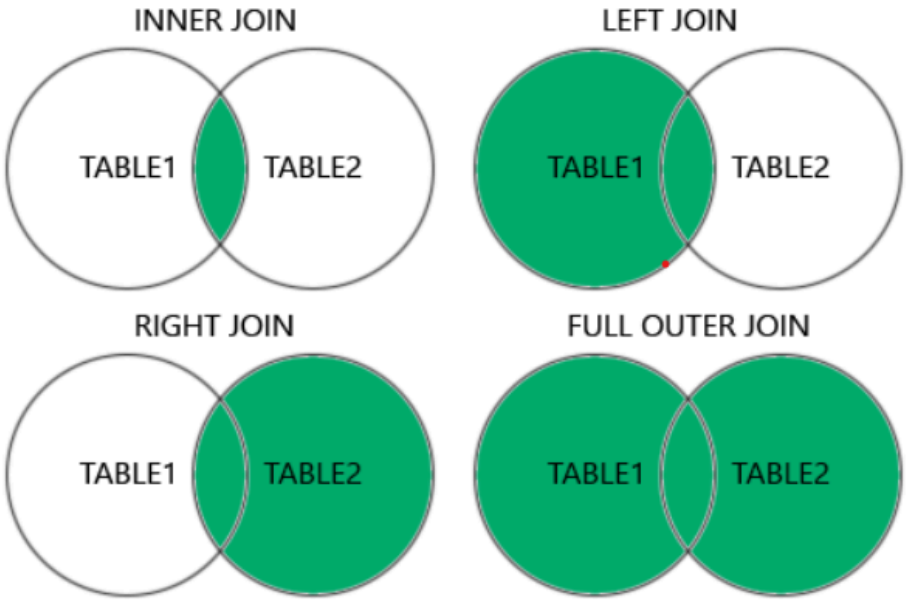
Aggregate functions are used to perform calculations on a set of values and return a single result.

These are extremely useful when working with large datasets to summarize, group, or aggregate data. Examples are Sum, average, minimum, maximum, group by, having,

Syntax:

```
SELECT COUNT(*) FROM movies;

SELECT SUM(revenue) FROM movies;
--Adds up all the values in a numeric column.

SELECT AVG(rating) FROM movies;
-- Returns the average of a numeric column.

SELECT MIN(release_year) FROM movies;

SELECT MAX(release_year) FROM movies;

SELECT release_year, AVG(rating)
FROM movies
GROUP BY release_year;
--groups the movies by their release year and calculates the average
rating for each year

SELECT release_year, AVG(rating)
FROM movies
GROUP BY release_year
HAVING AVG(rating) > 7;
-- will return only the years where the average movie rating is greater
than 7
```

JOINS



Joins are used in SQL to combine data from two or more tables based on a related column between them.

Joins allow us to retrieve comprehensive information by linking records from different tables.

Types of Joins

There are several types of joins, each useful for different situations. Let's cover the most common ones:

1. INNER JOIN

- **Purpose**: Returns only the rows where there is a match in both tables.
- **How it works**: The INNER JOIN finds records that have matching values in both tables. If there is no match, those records are excluded from the result.
- **Example**:

```
SELECT actors.actor_name, movies.title
FROM actors
INNER JOIN movie_cast ON actors.actor_id = movie_cast.actor_id
INNER JOIN movies ON movie_cast.movie_id = movies.movie_id;

-- returns a list of actors and the movies they appeared in. It
only includes actors who have appeared in movies.
```

2. LEFT JOIN (or LEFT OUTER JOIN)

- **Purpose**: Returns all the rows from the left table and the matched rows from the right table. If there is no match, the result is NULL on the side of the right table.
- **How it works**: The LEFT JOIN ensures that all records from the left table are returned, even if there is no matching record in the right table.
- **Example:**

```
SELECT actors.actor_name, movies.title
FROM actors
LEFT JOIN movie_cast ON actors.actor_id = movie_cast.actor_id
LEFT JOIN movies ON movie_cast.movie_id = movies.movie_id;
--returns a list of actors and the movies they appeared in. If an actor
has not appeared in any movie, the movie title will be NULL.
```

3. RIGHT JOIN (or RIGHT OUTER JOIN)

- **Purpose**: Returns all the rows from the right table and the matched rows from the left table. If there is no match, the result is NULL on the side of the left table.
- **How it works**: The RIGHT JOIN ensures that all records from the right table are returned, even if there is no matching record in the left table.

- **Example:**

```
SELECT actors.actor_name, movies.title
FROM actors
RIGHT JOIN movie_cast ON actors.actor_id = movie_cast.actor_id
RIGHT JOIN movies ON movie_cast.movie_id = movies.movie_id;
--This query returns a list of movies and their actors. If a movie has
no actors associated with it, the actor name will be NULL.
```

4. FULL OUTER JOIN

- **Purpose**: Returns all records when there is a match in either the left or right table. It returns `NULL` for non-matching rows from both tables.
- **How it works**: The FULL OUTER JOIN returns all rows from both tables, with `NULL` in places where there is no match.
- **Example:**

```
SELECT actors.actor_name, movies.title
FROM actors
FULL OUTER JOIN movie_cast ON actors.actor_id = movie_cast.actor_id
FULL OUTER JOIN movies ON movie_cast.movie_id = movies.movie_id;
--This query returns all actors and movies, regardless of whether they
match. If an actor has no movies, the movie title will be NULL, and if
a movie has no actors, the actor name will be NULL.
```

(ADVANCED/OPTIONAL)

5. Use of Aliases in Joins

You can simplify table references using aliases for better readability.

- **Example:**

```
SELECT a.actor_name, m.title
FROM actors a
INNER JOIN movie_cast mc ON a.actor_id = mc.actor_id
INNER JOIN movies m ON mc.movie_id = m.movie_id;

--Here, a, mc, and m are aliases for the actors, movie_cast, and movies
tables, respectively. This makes the query shorter and easier to read.
```

6. Joins with Multiple Tables

Joins can also be performed between multiple tables.

Example:

```
SELECT actors.actor_name, movies.title, movie_cast.role
FROM actors
JOIN movie_cast ON actors.actor_id = movie_cast.actor_id
JOIN movies ON movie_cast.movie_id = movies.movie_id;
--will return a list of actors, the movies they appeared in, and their
```

```
role in those movies by combining three tables (actors, movie_cast, and
movies)
```

7. SELF JOIN

Purpose: A self join is a regular join but the table is joined with itself.

How it works: It is useful when we want to relate rows in the same table.

- **Example:**

```
SELECT a.actor_name AS actor1, b.actor_name AS actor2
FROM actors a
INNER JOIN actors b ON a.actor_id <> b.actor_id
WHERE a.actor_id = 1;
--returns the names of all actors (except the one with actor_id = 1)
that are not the same actor, for comparison or analysis.
```

2. Correlated subqueries in SELECT, WHERE, and FROM clauses.

A subquery (also known as a nested query or inner query) is a query that is embedded inside another query, usually within the WHERE, FROM, or SELECT clauses.

A correlated subquery is a subquery that depends on the outer query.

Unlike a regular subquery, which can be executed independently, a correlated subquery is evaluated once for each row processed by the outer query.

Sub query Example:

```
SELECT employee_name
FROM employees
WHERE department_id = (SELECT department_id FROM departments WHERE
department_name = 'Sales');
```

Correlated Subquery Example

The inner query cannot run independently. It relies on values from the outer query (e.department_id) for each row.

```
SELECT employee_name
FROM employees e
WHERE salary > (
    SELECT AVG(salary)
    FROM employees e2
    WHERE e2.department_id = e.department_id
);
```

Use Cases and examples

- Finding Records Based on a Condition Related to Other Records:

For example, finding actors whose movies have a higher box office collection than a specific value.

- Complex Filtering:

Correlated subqueries allow for more complex filtering criteria when comparing rows between two tables.

Example 1: Find all actors who acted in movies that have a box office collection greater than $500 million.

```
SELECT actor_name
FROM actors a
WHERE EXISTS (
    SELECT 1
    FROM movies m
    WHERE m.movie_id = a.movie_id
    AND m.box_office > 500000000
);
```

The subquery checks each movie that the actor is linked to and compares the box office collection.

For each actor (outer query), it runs the subquery, checking if the box office collection of their movies exceeds $500 million.

Example 2: Get actors who appeared in movies where the release year is greater than the average release year of all movies.

```
SELECT actor_name
FROM actors a
WHERE EXISTS (
    SELECT 1
    FROM movies m
    WHERE m.movie_id = a.movie_id
    AND m.release_year > (SELECT AVG(release_year) FROM movies)
);
```

3. Window functions for advanced analytics.

Window functions allow you to perform calculations across a set of table rows that are somehow related to the current row.

Unlike aggregate functions, which return a single result for an entire query, window functions return a value for each row, based on a set of rows that are related to the current one.

Key Concepts

Window: The set of rows that the window function operates on. It is defined using the OVER() clause, which allows specifying how rows should be grouped (e.g., by partitioning and ordering).

Partitioning: Grouping rows into subsets (partitions) where window functions are applied independently to each subset.

Ordering: Sorting the rows within each partition, which is useful for functions like ROW_NUMBER() and LEAD().

Common Window Functions

1. ROW_NUMBER()

Assigns a unique sequential integer to rows within a partition of a result set, starting from 1 for the first row in each partition.

```
SELECT actor_name, ROW_NUMBER() OVER (ORDER BY actor_name) AS rank
FROM actors;
```

Use case: You might use ROW_NUMBER() to rank actors alphabetically or in another meaningful order.

Example: Assign a unique rank to each actor, ordered by their salary:

```
SELECT actor_name, salary,
ROW_NUMBER() OVER (ORDER BY salary DESC) AS salary_rank
FROM actors;
```

2. RANK()

Rank is a way of ordering or assigning a position to items (or rows) based on a specific value or criterion.

In SQL, you often rank rows based on a column's values using window functions like `RANK()`.

Similar to `ROW_NUMBER()`, but it gives the same rank to tied rows and skips the subsequent ranks.

```
SELECT actor_name, RANK() OVER (ORDER BY actor_name) AS rank
FROM actors;
```

Use case: Useful when you want to rank actors or employees but keep the ranking gap if there are ties.

Actor Name	Salary
John	50000
Alice	50000
Bob	40000
Charlie	30000

If we apply `RANK()` based on salary, the result would be:

Actor Name	Salary	Rank
John	50000	1
Alice	50000	1
Bob	40000	3
Charlie	30000	4

Actor Name Salary John 50000 Alice 50000 Bob 40000 Charlie 30000

After applying Rank():

Actor Name Salary Rank John 50000 1 Alice 50000 1 Bob 40000 3 Charlie 30000 4

Notice that both John and Alice have the same salary, so they are both given rank 1.

However, the next rank given to Bob is 3, skipping 2 because there was a tie for rank 1.

Example: Assign a rank to each actor, ordered by their salary:

```
SELECT actor_name, salary,
RANK() OVER (ORDER BY salary DESC) AS salary_rank
FROM actors;
```

3. DENSE_RANK()

Like `RANK()`, but it does not skip the next rank for tied rows. It ensures that the ranks are continuous.

```
SELECT actor_name, DENSE_RANK() OVER (ORDER BY actor_name) AS rank
FROM actors;
```

Use case: Useful when you want to rank actors or employees but keep the ranking gap if there are ties.

Example: Assign a rank to each actor, ordered by their salary (No skipped ranks)

```
SELECT actor_name, salary,
DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_salary_rank
FROM actors;
```

4. SUM() (Cumulative Sum)

Sums values over a window (partition) and returns the cumulative total.

```
SELECT actor_name, salary, SUM(salary) OVER (ORDER BY salary) AS
cumulative_salary
FROM actors;
```

Use case: Calculating running totals or cumulative metrics, such as cumulative sales or cumulative salary.

Example: Find the running total of actor salaries, ordered by salary

```
SELECT actor_name, salary,
SUM(salary) OVER (ORDER BY salary) AS cumulative_salary
FROM actors;
```

Example: Find the running total of total salary paid to all actors in the database:

```
SELECT SUM(salary) AS total_salary
FROM actors;
```

5. LEAD()

Returns the value of a column from the next row in the result set, within the same partition.

```
SELECT actor_name, salary, LEAD(salary, 1) OVER (ORDER BY salary) AS
next_salary
FROM actors;
```

Use case: Use `LEAD()` to compare current row data with the next row's data, such as comparing an employee's salary to the next employee.

Example: Retrieve the next actor's name and next salary for each actor

```
SELECT actor_name, salary,
LEAD(actor_name) OVER (ORDER BY salary DESC) AS next_actor_name,
LEAD(salary) OVER (ORDER BY salary DESC) AS next_actor_salary
FROM actors;
```

6. LAG()

Similar to `LEAD()`, but it returns the value of a column from the previous row.

```
SELECT actor_name, salary, LAG(salary, 1) OVER (ORDER BY salary) AS
prev_salary
FROM actors;
```

Use case: Useful for comparing a row's data with the previous row, such as comparing sales or salary changes between periods.

Example: Retrieve the previous actor's name and previous salary for each actor:

```sql
SELECT actor_name, salary,
LAG(actor_name) OVER (ORDER BY salary DESC) AS previous_actor_name,
LAG(salary) OVER (ORDER BY salary DESC) AS previous_actor_salary
FROM actors;
```

4. Hands-on examples using real-world datasets.

Live Exercise

Now it's your turn!

Task 0: Super Simple Movie Database

In this easy and fun exercise, you'll work with a very basic movie database to help you revise the ultimate basics of SQL.

This activity will cover creating tables, inserting data, and writing simple queries.

Step 1: Create Tables You will create two small tables:

- movies: Contains information about movies.
- directors: Contains information about directors.

```sql
CREATE TABLE movies (
    movie_id SERIAL PRIMARY KEY,
    movie_name VARCHAR(100),
    release_year INT
);

CREATE TABLE directors (
    director_id SERIAL PRIMARY KEY,
    director_name VARCHAR(100),
    movie_id INT REFERENCES movies(movie_id)
);
```

Step 2: Insert Data Add some fun movie and director data into these tables.

```sql
INSERT INTO movies (movie_name, release_year)
VALUES
('Inception', 2010),
('Interstellar', 2014),
('The Dark Knight', 2008);

INSERT INTO directors (director_name, movie_id)
VALUES
('Christopher Nolan', 1),
('Christopher Nolan', 2),
('Christopher Nolan', 3);
```

Step 3: Query Data Now, run some super simple queries to revise the basics.

Practice querrying now:

Query 1: Retrieve All Movies

Goal: Display all the movies in the database.

```
SELECT * FROM movies;
```

Query 2: Find Movies Released After 2010

Goal: Use the WHERE clause to filter movies based on the release year.

```
SELECT movie_name, release_year
FROM movies
WHERE release_year > 2010;
```

Query 3: Find the Director of a Movie

Goal: Use a JOIN to find the director of a specific movie.

```
SELECT m.movie_name, d.director_name
FROM movies m
JOIN directors d ON m.movie_id = d.movie_id
WHERE m.movie_name = 'Inception';
```

Query 4: Count the Number of Movies

Goal: Use an aggregate function (COUNT()) to count how many movies are in the database.

```
SELECT COUNT(*) AS total_movies FROM movies;
```

Task 1: Restaurant Orders & Customers Data Analysis

In this exercise, you'll manage and analyze data from a fictional restaurant's orders and customers. Your goal is to query the database to get meaningful insights such as customer behavior and order details. This task focuses on basic SQL querying with an emphasis on joins.

Step 1: Create Tables You'll create two simple tables:

customers: Contains customer information. orders: Contains details of food orders made by customers.

Step 2: Insert Data into Tables Populate the tables with some fun, fictional data for customers and their food orders.

- Use these sample values:

```
VALUES
('Alice', 'alice@mail.com'),
('Bob', 'bob@mail.com'),
('Charlie', 'charlie@mail.com'),
('Daisy', 'daisy@mail.com');
```

```
VALUES
(1, '2024-10-10', 'Pizza', 12.99),
(2, '2024-10-10', 'Burger', 9.50),
(1, '2024-10-11', 'Salad', 7.50),
(3, '2024-10-11', 'Pasta', 11.00),
(4, '2024-10-12', 'Sandwich', 6.99),
(2, '2024-10-12', 'Fries', 4.00);
```

Step 3: Query Basic Information Start by retrieving basic information from the orders table to get familiar with the data.

- Retrieve all the order details including customer names.

Step 4: Find Customers Who Ordered Specific Items Let's find out which customers ordered a "Pizza."

- Use a JOIN to get the names of customers who ordered Pizza.

Step 5: Calculate Total Spending per Customer Find out how much each customer spent at the restaurant in total.

- Use SUM() to calculate the total amount spent by each customer.

Step 6: Identify the Most Frequent Customer Let's see who has placed the most orders.

- Use COUNT() and GROUP BY to find out which customer placed the highest number of orders.

Step 7: Query Recent Orders Find out all the orders placed in the last two days (from the current date).

- Use WHERE to filter orders based on the date.

Step 8: Customers with Multiple Orders on the Same Day Check if any customers placed more than one order on the same day.

- Use HAVING to find customers with multiple orders on a single day.

Task 2: Sales Data Analysis

This exercise simulates a scenario you might encounter in data warehousing and data analytics. You will work with sales data to analyze product sales, revenue, and customer behavior—tasks similar to what analysts perform in companies.

Instructions

Step 1: Create Tables You'll create two tables for this exercise:

products: Contains information about products, such as product name and price. sales: Contains information about sales transactions, including product sold, quantity, and the total revenue.

Step 2: Insert Data into Tables Populate the tables with products and sales data.

Step 3: Calculate Total Sales and Revenue Calculate the total quantity sold and the total revenue for each product using SUM().

- Use SUM() to calculate the total revenue and total sales for each product.

Step 4: Ranking Products by Sales Performance Rank the products based on their total sales and revenue using RANK().

- Use RANK() to rank products by sales performance.

Step 5: Cumulative Revenue Calculate the cumulative total revenue across products to see how revenue adds up.

- Use a window function to calculate cumulative sums for revenue.

Step 6: Compare Sales with Lead and Lag Compare sales and revenue between consecutive products using LEAD() and LAG().

- Use LEAD() and LAG() to analyze differences in total revenue between consecutive products.

Some Additional Resources:

- Installation tutorial for postgreSQL -> https://www.youtube.com/watch?v=4qH-7w5LZsA

Live Exercise Solutions

Task 1: Restaurant Orders & Customers Data Analysis

In this exercise, you'll manage and analyze data from a fictional restaurant's orders and customers. Your goal is to query the database to get meaningful insights such as

customer behavior and order details. This task focuses on basic SQL querying with an emphasis on joins.

Step 1: Create Tables You'll create two simple tables:

customers: Contains customer information. orders: Contains details of food orders made by customers.

```
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    customer_name VARCHAR(100),
    contact_info VARCHAR(100)
);

CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INT REFERENCES customers(customer_id),
    order_date DATE,
    food_item VARCHAR(100),
    amount DECIMAL(10, 2)
);
```

Step 2: Insert Data into Tables Populate the tables with some fun, fictional data for customers and their food orders.

```
INSERT INTO customers (customer_name, contact_info)
VALUES
('Alice', 'alice@mail.com'),
('Bob', 'bob@mail.com'),
('Charlie', 'charlie@mail.com'),
('Daisy', 'daisy@mail.com');

INSERT INTO orders (customer_id, order_date, food_item, amount)
VALUES
(1, '2024-10-10', 'Pizza', 12.99),
(2, '2024-10-10', 'Burger', 9.50),
(1, '2024-10-11', 'Salad', 7.50),
(3, '2024-10-11', 'Pasta', 11.00),
(4, '2024-10-12', 'Sandwich', 6.99),
(2, '2024-10-12', 'Fries', 4.00);
```

Step 3: Query Basic Information Start by retrieving basic information from the orders table to get familiar with the data.

- Retrieve all the order details including customer names.

```
SELECT o.order_id, c.customer_name, o.food_item, o.amount
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;
```

Step 4: Find Customers Who Ordered Specific Items Let’s find out which customers ordered a "Pizza."

- Use a JOIN to get the names of customers who ordered Pizza.

```
SELECT c.customer_name, o.food_item
FROM orders o
```

```
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.food_item = 'Pizza';
```

Step 5: Calculate Total Spending per Customer Find out how much each customer spent at the restaurant in total.

- Use SUM() to calculate the total amount spent by each customer.

```
SELECT c.customer_name, SUM(o.amount) AS total_spent
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
GROUP BY c.customer_name;
```

Step 6: Identify the Most Frequent Customer Let's see who has placed the most orders.

- Use COUNT() and GROUP BY to find out which customer placed the highest number of orders.

```
SELECT c.customer_name, COUNT(o.order_id) AS order_count
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
GROUP BY c.customer_name
ORDER BY order_count DESC;
```

Step 7: Query Recent Orders Find out all the orders placed in the last two days (from the current date).

- Use WHERE to filter orders based on the date.

```
SELECT o.order_id, c.customer_name, o.food_item, o.order_date
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date >= '2024-10-11';
```

Step 8: Customers with Multiple Orders on the Same Day Check if any customers placed more than one order on the same day.

- Use HAVING to find customers with multiple orders on a single day.

```
SELECT c.customer_name, o.order_date, COUNT(o.order_id) AS order_count
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
GROUP BY c.customer_name, o.order_date
HAVING COUNT(o.order_id) > 1;
```

Task 2: Sales Data Analysis

This exercise simulates a scenario you might encounter in data warehousing and data analytics. You will work with sales data to analyze product sales, revenue, and customer behavior—tasks similar to what analysts perform in companies.

Instructions

Step 1: Create Tables You'll create two tables for this exercise: products: Contains information about products, such as product name and price. sales: Contains information about sales transactions, including product sold, quantity, and the total revenue.

```sql
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100),
    price DECIMAL(10, 2)
);

CREATE TABLE sales (
    sale_id SERIAL PRIMARY KEY,
    product_id INT REFERENCES products(product_id),
    quantity_sold INT,
    sale_date DATE,
    total_revenue DECIMAL(10, 2)
);
```

Step 2: Insert Data into Tables Populate the tables with products and sales data.

```sql
INSERT INTO products (product_name, price)
VALUES
('Laptop', 1000.00),
('Smartphone', 500.00),
('Tablet', 300.00);

INSERT INTO sales (product_id, quantity_sold, sale_date, total_revenue)
VALUES
(1, 5, '2024-10-10', 5000.00),
(2, 10, '2024-10-11', 5000.00),
(3, 3, '2024-10-12', 900.00),
(1, 2, '2024-10-13', 2000.00);
```

Step 3: Calculate Total Sales and Revenue Calculate the total quantity sold and the total revenue for each product using SUM().

- Use SUM() to calculate the total revenue and total sales for each product.

```sql
SELECT p.product_name,
SUM(s.quantity_sold) AS total_quantity_sold,
SUM(s.total_revenue) AS total_revenue
FROM sales s
JOIN products p ON s.product_id = p.product_id
GROUP BY p.product_name;
```

Step 4: Ranking Products by Sales Performance Rank the products based on their total sales and revenue using RANK().

- Use RANK() to rank products by sales performance.

```sql
SELECT p.product_name,
SUM(s.total_revenue) AS total_revenue,
RANK() OVER (ORDER BY SUM(s.total_revenue) DESC) AS revenue_rank
FROM sales s
JOIN products p ON s.product_id = p.product_id
GROUP BY p.product_name;
```

Step 5: Cumulative Revenue Calculate the cumulative total revenue across products to see how revenue adds up.

- Use a window function to calculate cumulative sums for revenue.

```sql
SELECT p.product_name, s.total_revenue,
SUM(s.total_revenue) OVER (ORDER BY s.total_revenue DESC) AS
cumulative_revenue
FROM sales s
JOIN products p ON s.product_id = p.product_id
ORDER BY cumulative_revenue DESC;
```

Step 6: Compare Sales with Lead and Lag Compare sales and revenue between consecutive products using LEAD() and LAG().

- Use LEAD() and LAG() to analyze differences in total revenue between consecutive products.

```sql
SELECT p.product_name, s.total_revenue,
LAG(s.total_revenue) OVER (ORDER BY s.total_revenue DESC) AS
previous_revenue,
LEAD(s.total_revenue) OVER (ORDER BY s.total_revenue DESC) AS
next_revenue
FROM sales s
JOIN products p ON s.product_id = p.product_id;
```

Programming Interveiw Questions

1. topic:
    - question

Mohammad Idrees Bhat

Tech Skills Trainer | AI/ML Consultant