Data Analytics With SQL

Facilitator: Sandra Asagade

Table of Contents

In this course, you will learn:

What is SQL?

- **SQL**, or Structured Query Language, is a programming language used by many organizations to manage and manipulate large amounts of data.
- SQL is a fundamental tool for managing and working with data in a structured and efficient way. It is widely used in various applications and industries to handle data storage, retrieval, and analysis.
- It is designed to perform various tasks related to database management, including creating and modifying database structures, retrieving data from databases, inserting and updating data, and more.

SQL DBMS

A Database Management System (DBMS) is software that provides an interface for managing, organizing, and interacting with databases. **E.g.,** Microsoft SQL Server, MySQL, PostgreSQL, Oracle DB, etc.

Important Terms and Definitions

- **SQL (Structured Query Language):** SQL is a domain-specific programming language used for managing and manipulating relational databases.
- **NoSQL (Not Only SQL):** NoSQL databases are a category of databases that do not use traditional SQL-based relational models.
- **Relational databases:** Relational databases are structured systems with well-defined schemas, using tables to organize data into rows and columns.
- **Non-relational:** or NoSQL databases handle unstructured data, provide schema flexibility, and they may use various query languages and data models for applications with dynamic data needs, like social media platforms and content management systems
- **DBMS (Database Management System):** A DBMS is software that provides tools and interfaces to manage and interact with databases.

Important Terms and Definitions

- **Database:** A database is a structured collection of data organized for efficient storage and retrieval. It can contain multiple tables, views, and other objects.
- **Table:** In a relational database, a table is a structured set of data organized into rows and columns.
- **Queries:** Queries are SQL statements used to retrieve, modify, or manipulate data in a database.
- **Statements:** SQL statements are commands that perform specific tasks. These include SELECT (retrieve data), INSERT (add new data), UPDATE (modify existing data), DELETE (remove data), and others.
- **ER Diagram** (Entity-Relationship Diagram): ER diagrams are visual representations used to model the structure of a database. They define entities, their attributes, and the relationships between them.
- **Joins:** Joins are SQL operations used to combine data from two or more tables based on related columns. Common types of joins include INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN.

Importance of SQL in Data Analytics

- **Constraints:** Constraints in SQL are rules enforced on data columns to ensure the integrity, validity, and accuracy of the data within a database, such as PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, and CHECK
- **Scalability and Performance:** SQL databases are designed to handle large datasets efficiently, providing scalability for growing data needs and ensuring optimal performance during analysis tasks.
- **Database Design and Modeling:** SQL is crucial for designing and creating relational databases, establishing relationships between tables, and ensuring the integrity of data structures, which is foundational for effective Data Analytics.
- **Data Retrieval and Selection:** SQL allows analysts to extract and retrieve specific data from database
- **Data Organization:** SQL databases allow for the organization of data into tables, enabling easy categorization and retrieval of data.

SQL Commands

SQL commands are instructions used to communicate with a database to perform tasks such as querying, updating, and managing data. These commands are grouped into several categories based on their functionality:

- **DDL:** Commands to define and modify database structures.

  CREATE: Creates new tables, databases, indexes, or views.

  ALTER: Modifies existing database objects.

  DROP: Deletes existing database objects.

  TRUNCATE: Removes all rows from a table without deleting the table itself.

- **DML:** Commands to manipulate data within tables.

  INSERT: Adds new rows of data to a table.

  UPDATE: Modifies existing data within a table.

  DELETE: Removes rows of data from a table.

SQL Commands

- **DQL:** Commands to query the database for information.

  SELECT: Retrieves data from one or more tables.

- **DCL:** Commands to control access to data within the database.

  GRANT: Gives users access privileges to the database.

  REVOKE: Removes access privileges from users.

- **TCL:** Commands to manage transactions within the database.

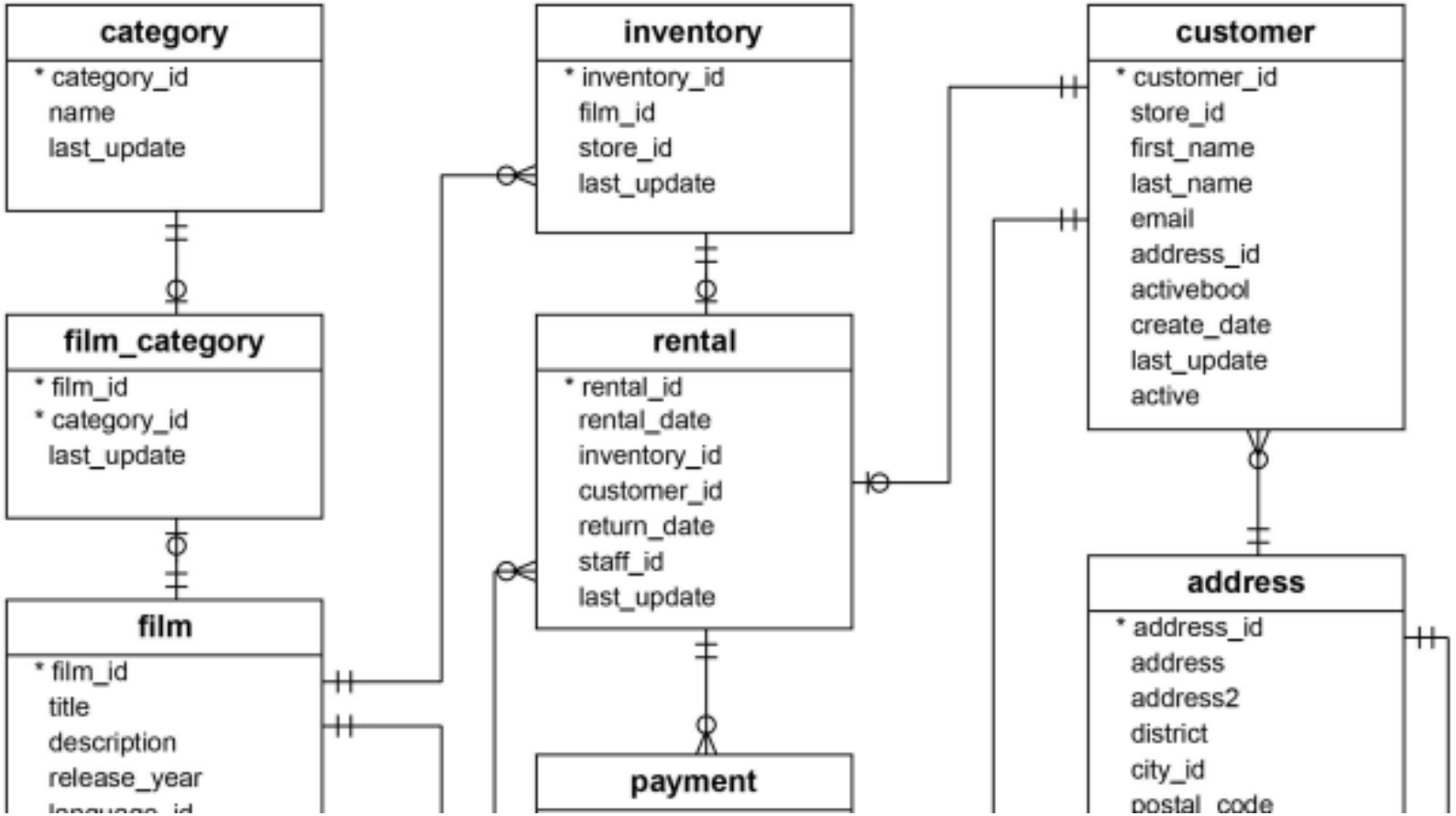
  COMMIT: Saves all changes made during the current transaction.

  ROLLBACK: Undoes changes made during the current transaction.

  SAVEPOINT: Sets a point within a transaction to which you can later roll back.

Entity Relationship (ER) Diagram

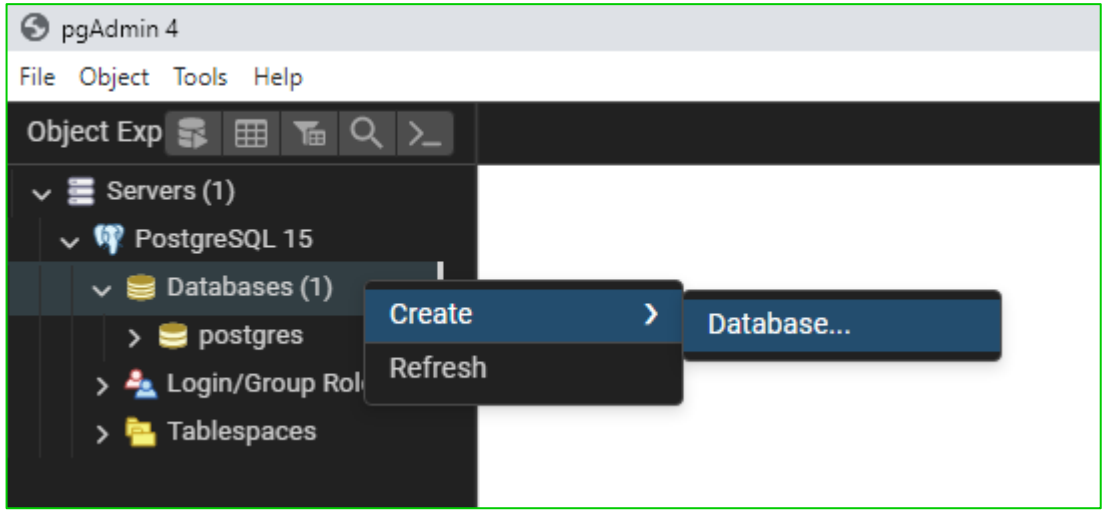
ER diagrams are visual representations used to model the structure of a database. They define entities, their attributes, and the relationships between them.



More to learn in this course

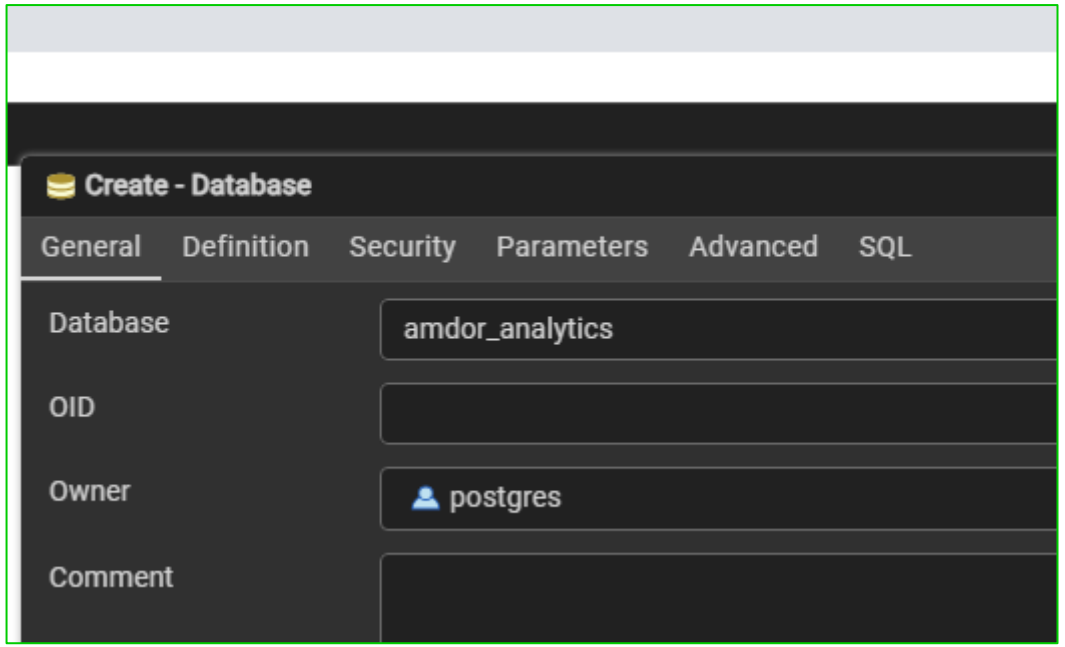
- Creating Databases and Tables, Modify Tables
- Importing data files (External Tables)
- Basic SQL Queries
- Restoring
- Aggregation
- Sorting & Filtering
- Date Formatting
- Update table
- Nested Queries
- JOINs
- Advanced SQL
- Views & Procedures
- Backup Databases
- Connecting Power BI to PostgreSQL database
- Case Study
- Capstone Project

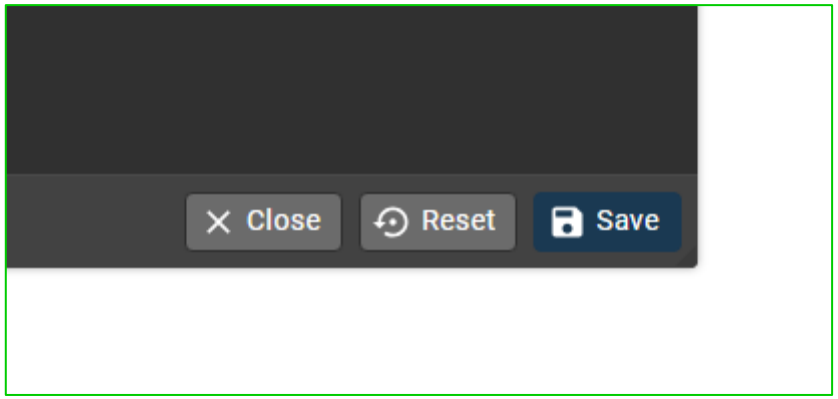
How to Create a Database in PgAdmin



1. Right Click on the Default Databases > Create > Database.

2. Name the Database, and Click the Save button.





How to Create a Table in PgAdmin

1.

- Expand the **Schemas** in your database
- Right-Click on Tables > Create > Table
- Under General settings, name your table
- Under Column settings, add relevant columns to your table and include their respective data types
- Under Constraints settings, specify constraints present in your table

2.

- Right-Click on Tables > Query Tool
- Write the correct syntax to create table using PostgreSQL

  CREATE TABLE table_name(

  column_name(s) data type);

- Now lets create a table for students in C24-04

```
DROP TABLE IF EXISTS student;

CREATE TABLE students(
id VARCHAR(5),
first_name VARCHAR(25),
last_name VARCHAR(25),
gender VARCHAR(7),
location VARCHAR(15));
```

How to view table results using the DQL SELECT FROM WHERE

- The first step to writing queries, is understanding the task at hand
- Know the columns you are to work with, which will be included in the SELECT command
- To select a single column

  Syntax: *SELECT first_name FROM students;*
- To select multiple columns

  Syntax: *SELECT first_name, gender, location FROM students;*
- To select all columns

  Syntax: *SELECT * FROM students;*

  ***** is used when you want to return all columns from your table
- To alias a selected column(s)

  Syntax: *SELECT first_name AS name, gender, location FROM students;*

  AS is used together with the selected column to give a different name to the column in the result set without changing how it is in the original table

How to Modify (Edit) a Table

ALTER TABLE??

Modifying a table in PostgreSQL involves using the **ALTER TABLE** command to perform various changes such as adding, dropping, or renaming columns, changing data types, adding constraints, and more.

Example Scenario 1: Suppose we forgot to add student's phone numbers, and we'd like to do that without having to delete and create the table again.

Syntax: *ALTER TABLE students ADD COLUMN phone_number VARCHAR(14);*

Example Scenario 2: Suppose we'd like to change the data type of student's id column to integer

Syntax: *ALTER TABLE students ALTER COLUMN id SET DATA TYPE INT USING id::int ;*

Example Scenario 3: Suppose we want to drop the phone_number column

Syntax: *ALTER TABLE students DROP COLUMN phone_number;*

Example Scenario 4: suppose we want to change the name of the id column to student_id

Syntax: *ALTER TABLE students RENAME COLUMN id TO student_id;*

How to Modify (Edit) a Table

Example Scenario 5: suppose we want to make student_id the primary key of the students table

***Syntax: ALTER TABLE** students **ADD CONSTRAINT** students_pkey **PRIMARY KEY** (student_id);*

Example Scenario 6: suppose we have a separate table for student's grades [table name: grade] with the columns: grade_id, course_name, student_id, grade. Assume the table was created with only one constraint (grade_id as the primary key), and we'd like to create a relationship between the grade and students table using the student_id as the foreign key.

Syntax: first we need to create the grade table;

```
CREATE TABLE grade(
grade_id INT PRIMARY KEY, course_name VARCHAR(10), student_id INT, grade VARCHAR(1)
);
-- setting student_id as a foreign key in the grade table
ALTER TABLE grade ADD CONSTRAINT grade_student_id_fkey FOREIGN KEY (student_id) REFERENCES students (student_id);
```

How to Import external data file (csv)

Suppose we have a csv (comma separated values) file named 2011Sales.csv, and because it is a growing data, we'd like to store it in our database [amdor_analytics].

Steps:

- Create a table for the file
- Add relevant columns, data types and constraints
- Run the syntax
- Import the file into the created table using the **COPY** command;

  Syntax: *COPY table_name FROM 'filepath.csv' DELIMITER ',' CSV HEADER;*

Basic SQL Queries

Lets get some insights from the sales data;

- Write a query to return the total number of orders

  Syntax: *SELECT * FROM sales2011;*

- Write a query to return the total number of customers

  Syntax: *SELECT COUNT(DISTINCT(customer_id) FROM sales2011;*

- Write a query to return the total revenue generated from sales

  Syntax: *SELECT SUM(sales) FROM sales2011;*

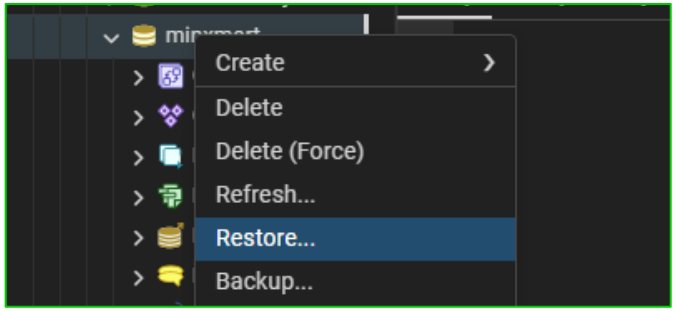
- Write a query to return the maximum shipping cost

  Syntax: *SELECT MAX(shipping_cost) FROM sales2011;*

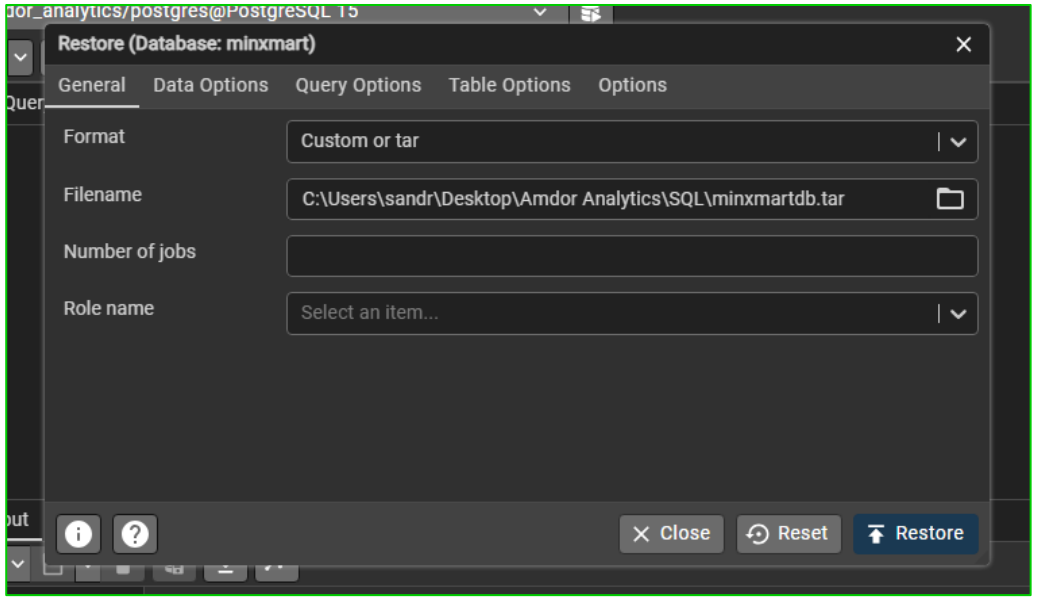
- Write a query to return the total number of orders from the consumer segment

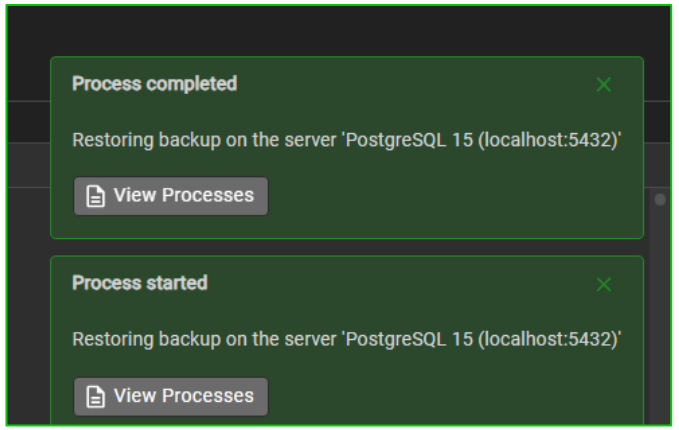
  Syntax: *SELECT * FROM sales2011 WHERE segment = 'Consumer';*

How to Restore a database in PgAdmin



- Create database first (*see slide 8 for steps*)
- Right-Click on the created database > Restore
- Under general settings, select Custom or tar
- Select or browse the database filename/file path
- When browsing the filename, ensure to change file type, from Custom files to All files
- Click on the restore button to restore
- You will see Process Started, if successful, the Process will be completed
- We will be restoring the database of minxmart, a retail company located in the US





Aggregation in SQL

Aggregations in SQL are functions that perform a calculation on a set of values and return a single value.

Here are some of the most commonly used aggregation functions:

- **COUNT():** Returns the number of rows that match a specified condition.
- **SUM():** Returns the total sum of a numeric column.
- **AVG():** Returns the average value of a numeric column.
- **MIN():** Returns the smallest value in a set of values.
- **MAX():** Returns the largest value in a set of values.
- **DISTINCT:** Used within aggregate functions to return the sum of distinct (unique) values.

Aggregation in SQL

Example Scenario 1: Suppose we want to return the total number of rows in the sales_order table

Syntax: *SELECT COUNT(*)*

FROM sales_order;

Example Scenario 2: Suppose we want to return the total number of orders that minxmart received.

Syntax: *SELECT COUNT(order_number)*

FROM sales_order;

Example Scenario 3: Suppose we want to return the total revenue generated.

Syntax: *SELECT SUM(unit_price * order_quantity) AS revenue*

FROM sales_order;

Example Scenario 4: Suppose we want to return the distinct number of products ordered.

Syntax: *SELECT COUNT(DISTINCT product_id)*

FROM sales_order;

Aggregation in SQL

Example Scenario 5: Suppose we want to return the most recent order date.

Syntax: *SELECT MAX(order_date)*

FROM sales_order;

Example Scenario 6: Suppose we want to return the earliest order date.

Syntax: *SELECT MIN(order_date)*

FROM sales_order;

Example Scenario 7: Suppose we want to return the overall average price of all products.

Syntax: *SELECT AVG(unit_price)*

FROM sales_order;

Aggregation + Group By

aggregation functions with the **GROUP BY** clause to get aggregated values for groups of rows.

Example Scenario 1: Suppose we want to return the total number of orders handled by each salesteam

Syntax: *SELECT salesteam_id, COUNT(order_number) AS orders*

FROM sales_order

GROUP BY salesteam_id;

Example Scenario 2: Suppose we want to return the total units/quantity ordered for each product

Syntax: *SELECT product_id, SUM(order_quantity)*

FROM sales_order

GROUP BY product_id;

Example Scenario 3: Suppose we want to return the total number of states in each of the regions

Syntax: SELECT region, COUNT(state_code) AS states_count

FROM region

GROUP BY region;

Sorting in SQL

Sorting in SQL is performed using the **ORDER BY** clause, which is used to arrange the result set of a query in either ascending or descending order based on one or more columns. This helps in organizing the data in a meaningful way for analysis or presentation.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

Example Scenario 1: Suppose we want to return the order number, customer id, product id, and quantity columns, but we would want the result to be ordered by the latest order number first

Syntax:

```
SELECT order_number, customer_id, product_id, order_quantity
FROM sales_order
ORDER BY order_number DESC;
```

Sorting in SQL

Example Scenario 2: Suppose we want to return all columns in the customer table, with the result ordered by the customer names [A-Z]

Syntax: *SELECT * FROM customer ORDER BY customer_name ASC;*

Example Scenario 3: Suppose we want to return the total number of orders handled by each salesperson, with the result ordered by their numbers (highest first)

Syntax: *SELECT salesteam_id, COUNT(order_number) AS orders*

FROM sales_order

GROUP BY salesteam_id

ORDER BY orders DESC;

Sorting in SQL

Example Scenario 4: Suppose we want to return the order number, customer id, product id, quantity, and revenue columns, but we would want the result to be ordered by the highest revenue first

Syntax:

```
SELECT order_number, customer_id, product_id, order_quantity, SUM(unit_price * order_quantity) AS revenue
FROM sales_order
GROUP BY order_number, customer_id, product_id
ORDER BY revenue DESC;
```

Sorting + Limit

The **LIMIT** clause in SQL is used to restrict the number of rows returned by a query. When using LIMIT, it's common to also use the **ORDER BY** clause to ensure the rows are returned in a specific order before applying the limit.

Example Scenario 1: Suppose we want to return the top 5 most ordered products

Syntax:

```
SELECT product_id, COUNT(*) AS orders
FROM sales_order
GROUP BY product_id
ORDER BY orders DESC
LIMIT 5;
```

Sorting + Limit

Example Scenario 2: Suppose we want to return the least/bottom 5 ordered products

Syntax:

SELECT product_id, COUNT(*) AS orders

FROM sales_order

GROUP BY product_id

ORDER BY orders ASC

LIMIT 5;

Example Scenario 3: Suppose we want to return the id of the salesperson who handled the most orders

Syntax:

SELECT salesteam_id, COUNT(*) AS orders

FROM sales_order

GROUP BY salesteam_id

ORDER BY orders DESC

LIMIT 1;

Filtering in SQL

Filtering in PostgreSQL is accomplished using the **WHERE** clause, which allows you to specify conditions that rows must meet to be included in the result set. Various operators and keywords can be used within the WHERE clause to create more complex and precise filters.

Operators

Operators are used to specify conditions in the WHERE clause.

1. **Comparison Operators: =, !=, >, <, >=, <=**

Example Scenario 1: Suppose we want to return only orders from customer with customer id 22

Syntax: *SELECT * FROM sales_order WHERE customer_id = 22;*

Example Scenario 2: Suppose we want to return the total number or orders made via the Wholesale sales channel

Syntax: *SELECT COUNT(*) FROM sales_order WHERE sales_channel = 'Wholesale';*

Example Scenario 3: Suppose we want to return all orders except those via the In-Store sales channel

Syntax: *SELECT * FROM sales_order WHERE sales_channel !='In-Store';*

Filtering in SQL

1. **Comparison Operators: =, !=, >, <, >=, <=**

Example Scenario 4: Suppose we want to return orders having above 5 in quantity

Syntax: *SELECT * FROM sales_order WHERE order_quantity > 5;*

Example Scenario 5: Suppose we want to return the total number of orders having their quantities less than or equal to 4

Syntax: *SELECT COUNT(*) FROM sales_order WHERE order_quantity <= 4;*

Example Scenario 6: Suppose we want to return the total revenue generated only if the revenue from orders is greater than 5000

Syntax: *SELECT SUM(unit_price * order_quantity) AS revenue FROM sales_order WHERE revenue > 5000;*

Filtering in SQL

2. Logical Operators: AND, OR, NOT

Example Scenario 1: Suppose we want to return orders via the Distributor sales channel with discount above 3%

Syntax: *SELECT * FROM sales_order WHERE sales_channel = 'Distributor' AND discount_applied > 0.03;*

Example Scenario 2: Suppose we want to return only orders with product id 12 and order date is 31st May, 2018

Syntax: *SELECT * FROM sales_order WHERE product_id = 12 AND order_date = '2018-05-31';*

Example Scenario 3: Suppose we want to return stores that are either in Alabama or the City Type is Town

Syntax: *SELECT * FROM location WHERE state= 'Alabama' OR type = 'Town';*

Filtering in SQL

3. IN/NOT IN

Example Scenario 1: Suppose we want to return orders from three customers with id 13, 17 and 20

Syntax: *SELECT * FROM sales_order WHERE customer_id IN (13, 17, 20);*

Example Scenario 2: Suppose we want to return orders from all sales channel excluding Wholesale, In-Store, and Online

Syntax: *SELECT * FROM sales_order WHERE sales_channel NOT IN (*'Wholesale', 'In-Store', 'Online'*);*

4. BETWEEN

Example Scenario 1: Suppose we want to return orders having products with unit price between 500 and 1000

Syntax: *SELECT * FROM sales_order WHERE unit_price BETWEEN 500 AND 1000;*

Example Scenario 2: Suppose we want to return orders made between 1st June, 2018 and 30th June, 2018

Syntax: *SELECT * FROM sales_order WHERE order_date BETWEEN '2018-06-01' AND '2018-06-30';*

5. LIKE/LIKE With Wildcards – ‘%’ and ‘_’

Wildcards are used with the LIKE and ILIKE operators to search for patterns in text.

LIKE: Case-sensitive pattern matching.

ILIKE: Case-insensitive pattern matching.

Example Scenario 1: Suppose we want to return the id of a customer whose name is Eminence Corp

Syntax: *SELECT * FROM customer WHERE customer_name LIKE ‘Eminence Corp’;*

Example Scenario 2: Suppose we want to return customers having Group in their name

Syntax: *SELECT * FROM customer WHERE customer_name LIKE ‘%Group%’;*

Example Scenario 3: Suppose we want to return the information of cities that begin with letter A

Syntax: *SELECT * FROM location WHERE city_name LIKE ‘A%’;*

Filtering in SQL

Example Scenario 4: Suppose we want to return the information of cities that end with the letter e

Syntax: *SELECT * FROM location WHERE city_name LIKE '%e';*

Example Scenario 5: Suppose we want to return the information of cities having the letters en anywhere in their name

Syntax: *SELECT * FROM location WHERE city_name ILIKE '%en%';*

Example Scenario 6: Suppose we want to return the information of 5-letters word cities

Syntax: *SELECT * FROM location WHERE city_name LIKE '_____';*

Example Scenario 7: Suppose we want to return the information of cities having 7 letters beginning with C

Syntax: *SELECT * FROM location WHERE city_name LIKE 'C_____';*

Date Formatting

You can format dates using the **TO_CHAR** function, which allows you to convert dates into various string formats

Common Date Format Patterns

YYYY: Year in 4 digits

YY: Year in 2 digits

MM: Month (01-12)

MON: Abbreviated month name (e.g., JAN)

MONTH: Full month name (e.g., JANUARY)

DD: Day of the month (01-31)

Day: Full day name (e.g., Sunday)

Dy: Abbreviated day name (e.g., Sun

Date Formatting

Example Scenario 1: Suppose we want to return only the year values from the order date column

Syntax: *SELECT TO_CHAR(order_date, 'YYYY') AS year FROM sales_order;*

Example Scenario 2: Suppose we want to return only the month values from the order date column

Syntax: *SELECT TO_CHAR(order_date, 'Mon') AS month FROM sales_order;*

Example Scenario 3: Suppose we want to return revenue generated per month

Syntax: *SELECT TO_CHAR(order_date, 'Mon') AS month,*

*SUM(unit_price * order_quantity) AS revenue*

FROM sales_order GROUP BY month ORDER BY revenue DESC;

Example Scenario 4: Suppose we want to return total number of orders by day of the week

Syntax: *SELECT TO_CHAR(order_date, 'Day') AS day_of_week ,*

COUNT(order_number) AS total_orders

FROM sales_order GROUP BY day_of_week ORDER BY total_orders DESC;

Date Formatting

Example Scenario 5: Suppose we want to return the order summary by month and day of the week

Syntax: *SELECT TO_CHAR(order_date, 'Mon') AS month,*

TO_CHAR(order_date, 'Day') AS day_of_week ,

COUNT(order_number) AS total_orders,

*SUM(unit_price * order_quantity) AS revenue*

FROM sales_order GROUP BY month, day_of_week ORDER BY revenue DESC;

How to Update Information in a Table

the **UPDATE** statement is used to modify existing records in a table. This command allows you to update one or more columns of one or more rows, depending on the condition specified in the **WHERE** clause.

Example Scenario 1: Suppose we want to create a revenue column in the sales order table

Syntax: *ALTER TABLE sales_order ADD COLUMN revenue DECIMAL;*

UPDATE sales_order

*SET revenue = unit_price * order_quantity;*

Example Scenario 2: Suppose we want to change the sales channel from In-Store to Retail

Syntax: *UPDATE sales_order*

SET sales_channel = 'Retail'

WHERE sales_channel = 'In-Store';

How to Update Information in a Table

Example Scenario 3: Suppose we want to set the discount of product with id 22 to 15% (0.15)

Syntax: *UPDATE sales_order*

SET discount_applied = 0.15

WHERE product_id = 22;

Example Scenario 4: Suppose we want to reduce the unit price of some products by 350 if the unit cost is greater than 1000

Syntax: *UPDATE sales_order*

SET unit_price = unit_price – 350

WHERE unit_cost > 1000;

Nested Queries

Nested queries, also known as **subqueries**, are queries embedded within another SQL query. They can be used to filter results, calculate aggregates, update data based on conditions, and perform various other operations, enhancing the flexibility and expressiveness of SQL.

Example Scenario 1: Suppose we want to get the total amount spent by a particular customer, but we only know their name

Syntax: *SELECT customer_id, SUM(unit_price * order_quantity) as revenue*

FROM sales_order

WHERE customer_id = (SELECT customer_id

FROM customer

WHERE customer_name LIKE 'Eminence Corp')

GROUP BY customer_id;

Nested Queries

Nested queries, also known as **subqueries**, are queries embedded within another SQL query. They can be used to filter results, calculate aggregates, update data based on conditions, and perform various other operations, enhancing the flexibility and expressiveness of SQL.

Example Scenario 1: Suppose we want to get the total amount spent by a particular customer, but we only know their name

Syntax: *SELECT customer_id, SUM(unit_price * order_quantity) as revenue*

FROM sales_order

WHERE customer_id = (SELECT customer_id

FROM customer

WHERE customer_name LIKE 'Eminence Corp')

GROUP BY customer_id;

Nested Queries

Example Scenario 2: Suppose we want to return order details for products with a unit price greater than the average unit price of all products

Syntax: *SELECT order_number, sales_channel,*

order_date, product_id, unit_price

FROM sales_order

WHERE unit_price > (SELECT AVG(unit_price) FROM sales_order);

JOINs in SQL

JOINs are used to combine rows from two or more tables based on a related column between them. They are essential for querying data across multiple tables efficiently.

Types of JOINs

- **JOIN (or INNER JOIN):** Combines rows from two tables only when there are matching values in a common column. Think of it as finding common friends between two lists.
- **LEFT JOIN (or LEFT OUTER JOIN):** Returns all rows from the left table and the matched rows from the right table. If there's no match, NULLs are shown for columns from the right table.
- **RIGHT JOIN (or RIGHT OUTER JOIN):** Returns all rows from the right table and the matched rows from the left table. If there's no match, NULLs are shown for columns from the left table.
- **FULL JOIN (or FULL OUTER JOIN):** Returns rows when there is a match in one of the tables. If there's no match, NULLs are shown for missing matches from either table. Think of it as combining all items from both lists, filling in blanks where matches don’t exist.

JOINs in SQL

Example Scenario 1: Suppose we want to return order details of customers who ordered via the Distributor sales channel

Syntax: *SELECT c.customer_name, so.order_number, so.product_id, so.order_quantity,*

so.discount_applied, so.unit_price, so.revenue

FROM sales_order so

INNER JOIN customer c ON c.customer_id = so.customer_id

WHERE so.sales_channel = 'Distributor';

Example Scenario 2: Suppose we want to return the total number of orders from each city

Syntax: *SELECT l.city_name, COUNT(so.order_number) AS total_orders*

FROM sales_order so

INNER JOIN location l ON l.store_id = so.store_id

GROUP BY l.city_name;

JOINs in SQL

Example Scenario 3: Suppose we want to return the total number of stores in each region

Syntax: *SELECT r.region, COUNT(l.store_id) AS total_stores*

FROM location l

INNER JOIN region r ON r.state_code = l.state_code

GROUP BY r.region;

Example Scenario 4: Suppose we want to return order details of customers who ordered via the Distributor sales channel, including the name of products ordered, and city of store where order took place

Syntax: *SELECT c.customer_name, so.order_number, l.city_name, p.product_name, so.order_quantity,*

so.discount_applied, so.unit_price, so.revenue

FROM sales_order so

INNER JOIN customer c ON c.customer_id = so.customer_id

INNER JOIN product p ON p.product_id = so.product_id

INNER JOIN location l ON l.store_id = so.store_id

WHERE so.sales_channel = 'Distributor';

JOINs in SQL

Example Scenario 5: Suppose we want to return the top 10 customers by total spending (revenue)

Syntax: *SELECT c.customer_name, SUM(so.unit_price * so.order_quantity) as revenue*

FROM customer c

LEFT JOIN sales_order so ON so.customer_id = c.customer_id

GROUP BY c.customer_name

ORDER BY revenue DESC LIMIT 10;

Example Scenario 6: Suppose we want to return the least/bottom 10 products by quantity ordered

Syntax: *SELECT p.product_name, SUM(so.order_quantity) order_quantity*

FROM sales_order so

RIGHT JOIN product p ON p.product_id = so.product_id

GROUP BY p.product_name

ORDER BY order_quantity ASC LIMIT 10;

Advanced SQL – Case When

CASE WHEN is used to create conditional logic within SQL queries. It allows you to perform different actions or return different values based on specified conditions.

Example Scenario 1: Suppose we want to categorize the price of products into high, medium and low

Syntax: *SELECT p.product_name, so.unit_price,*

```
CASE
    WHEN unit_price >= 1500 THEN 'High'
    WHEN unit_price >= 500 THEN 'Medium'
    ELSE 'Low'
END AS price_category
FROM product p
LEFT JOIN sales_order so ON p.product_id = so.product_id
GROUP BY p.product_name, so.unit_price, price_category
```

Advanced SQL – Case When

Example Scenario 2: Suppose we want to count the number of orders in each price category

Syntax: *SELECT*

COUNT(CASE WHEN unit_price > 1500 THEN 1 END) AS high,

COUNT(CASE WHEN unit_price > 500 AND unit_price <= 1500 THEN 1 END) AS medium,

COUNT(CASE WHEN unit_price <= 500 THEN 1 END) AS low

FROM sales_order

Advanced SQL – CTEs

Common Table Expressions **(CTEs)** are a way to define temporary result sets. CTEs improve query readability and organization, especially for complex queries. They are defined using the WITH clause.

Example Scenario 1: Suppose we want to know the number of orders in each price category

Syntax: *WITH price_category_cte AS (SELECT order_number, product_id, unit_price,*

CASE WHEN unit_price >= 1500 THEN 'High'

WHEN unit_price >= 500 THEN 'Medium'

ELSE 'Low'

END AS price_category

FROM sales_order

GROUP BY order_number, product_id, unit_price, price_category)

SELECT price_category, COUNT(order_number) AS total_orders

FROM price_category_cte

GROUP BY price_category;

Advanced SQL – Views

A **view** is a virtual table that is based on the result set of an SQL query. It provides a way to simplify complex queries, and enhance security by restricting access to specific data. Views do not store data themselves; they dynamically retrieve data from the underlying tables whenever they are queried.

Example Scenario 1: Suppose we want to create a view for orders made in August

Syntax: *CREATE VIEW august_orders AS*

*SELECT * FROM sales_order WHERE order_date BETWEEN '2018-08-01' AND '2018-08-31';*

Example Scenario 2: Suppose we want to create a view for the top 10 most ordered products

Syntax: *CREATE VIEW top10_most_ordered_product AS*

SELECT p.product_id, p.product_name, COUNT(so.order_number) AS orders

FROM sales_order so

INNER JOIN product p ON p.product_id = so.product_id

GROUP BY p.product_id, p.product_name

ORDER BY orders DESC

LIMIT 10;

Advanced SQL – Procedures

A **procedure** is a stored program that can execute a series of SQL statements and procedural logic. Procedures allow you to simplify and automate complex database operations using SQL and procedural logic.

Example Scenario 1: Suppose we want to create a procedure to add new products to the product table

Syntax: *CREATE OR REPLACE PROCEDURE insert_new_product(n_product_id INT, n_product_name VARCHAR)*

```
LANGUAGE plpgsql
AS $$
BEGIN
INSERT INTO product(product_id, product_name)
VALUES(n_product_id, n_product_name);
END;
$$;
CALL insert_new_product(48, 'can');
```

Advanced SQL – Procedures

Example Scenario 2: Suppose we want to create a procedure that updates the unit price of an order based on its order number

Syntax: *CREATE OR REPLACE PROCEDURE update_unit_price(order_num VARCHAR, new_price DECIMAL)*

```
LANGUAGE plpgsql
AS $$
BEGIN
UPDATE sales_order
SET unit_price = new_price
WHERE order_number = order_num;
END;
$$;
CALL update_unit_price('SO - 000211', 2500.00);
```

Advanced SQL – Functions

A **function** is a stored program that can return a single value or a result set. Functions simplify SQL queries that can be reused throughout your database applications.

Example Scenario 1: Suppose we want to create a function to calculate the total amount spent on an order

Syntax: *CREATE FUNCTION total_sales_orderN(f_order_number VARCHAR)*

RETURNS DECIMAL

LANGUAGE plpgsql

AS $$

DECLARE total DECIMAL;

BEGIN

*SELECT SUM(unit_price * order_quantity) INTO total*

FROM sales_order

WHERE order_number = f_order_number;

RETURN total;

END;

$$;

SELECT get_order_total('SO - 000211');

Advanced SQL – Functions

Example Scenario 2: Suppose we want to create a function to calculate the total amount spent on an order

Syntax: *CREATE FUNCTION total_sales_customer(f_customer_id INT)*

```
RETURNS DECIMAL
LANGUAGE plpgsql
AS $$
DECLARE total_sales NUMERIC;
BEGIN
    SELECT SUM(order_quantity * unit_price) INTO total_sales
    FROM sales_order
    WHERE customer_id = f_customer_id;
 RETURN total_sales;
END;
$$;
```

SELECT *total_sales_customer*(3)

Advanced SQL – Functions

Example Scenario 3: Suppose we want to create a function to return order details of a product when their product id is entered

Syntax: *CREATE FUNCTION get_product_details(f_product_id INT)*

RETURNS TABLE(order_number VARCHAR, sales_channel VARCHAR, product_name VARCHAR, order_quantity INT)

LANGUAGE plpgsql

AS $$

BEGIN

RETURN QUERY

SELECT so.order_number, so.sales_channel, p.product_name, so.order_quantity

FROM sales_order so

INNER JOIN product p ON p.product_id = so.product_id

WHERE p.product_id = f_product_id;

END;

$$;

SELECT * FROM *get_product_details*(10);

Advanced SQL – Functions

Example Scenario 4: Suppose we want to create a function to return all orders based on the sales channel entered.

Syntax: *CREATE OR REPLACE FUNCTION order_details_by_channel(f_sales_channel VARCHAR)*

RETURNS TABLE(order_number VARCHAR,sales_channel VARCHAR,order_date DATE,customer_id INT,

product_id INT,revenue DECIMAL)

LANGUAGE plpgsql

AS $$

DECLARE total_sales NUMERIC;

BEGIN

RETURN QUERY

SELECT so.order_number, so.sales_channel, so.order_date, so.customer_id, so.product_id, so.revenue

FROM sales_order so

WHERE so.sales_channel = f_sales_channel;

END;

$$;

SELECT * FROM order_details_by_channel('Online');

Advanced SQL – Functions

Example Scenario 5: Suppose we want to create a function to return all customers with a specified letter(s)/words anywhere in their name.

Syntax: *CREATE OR REPLACE FUNCTION search_customers(word VARCHAR)*

RETURNS TABLE(customer_id INT, customer_name VARCHAR)

LANGUAGE plpgsql

AS $$

BEGIN

RETURN QUERY

SELECT c.customer_id, c.customer_name

FROM customer c

WHERE c.customer_name ILIKE '%' || word || '%';

END;

$$;

SELECT * FROM search_customers('corp');

With SQL, you can communicate with data, and data communicates with you.