Follow Prem Mandal For Free Pdf & Resources

SQL FOR DATA ANALYTICS

To Become a Successful Data Analysts & Data Scientist

@PREMMANDAL
BUSINESS & DATA ANALYST

swipe---▶

SQL : Basic to Advance

History of SQL:

SQL, short for Structured Query Language, has a history rooted in the 1970s when IBM developed the System R project. It aimed to create a database management system that could handle and manipulate structured data efficiently. As part of System R, IBM introduced a language called Structured English Query Language (SEQUEL), which later evolved into SQL The name "SEQUEL" was already taken by another company. The creators of the language didn't want to get sued, so they changed the name to ‘SQL’. SQL is a backronym for "Structured Query Language." The language was influenced by Edgar F. Codd's relational model, which introduced the concept of organizing data into tables with relationships. Over time, SQL became the standard language for interacting with relational databases and was standardized by ANSI and ISO. Today, SQL is widely adopted and utilized in various database systems, enabling organizations to manage and analyze structured data effectively.

Let’s first understand the Data modeling and database:

What is Data Modeling?

Data Modeling is a critical step in defining data structure, creating data models to describe associations and constraints for reuse. Data model is the conceptual design or plan for organizing data. It visually represents data with diagrams, symbols, or text to visualize relationships.
Enhancing data analytics, data modeling assures uniformity in nomenclature, rules, semantics, and security. Regardless of the application, the emphasis is on the arrangement and accessibility of the data.

Advantages of Data Modelling

The following are the essential advantages of Data Modelling:

- The data model helps us choose the right data sources to populate the model.
- The Data Model improves communication throughout the company.
- The data model aids in the ETL process's documentation of the data mapping.
- We can use data modeling to query the database's data and generate various reports based on the data. Data modeling supports data analysis through reports.

Data Modeling Terminology:

- **Entity:** Entities are the objects or items in the business environment for which we need to store data. Defines what the table is about. For example, in an e-commerce system, entities can be Customers, Orders, or Products.
- **Attribute:** Attributes provide a way to structure and organize the data within an entity. They represent the specific characteristics or properties of an entity. For instance, attributes of a Customer entity can include Name, Address, and Email.
- **Relationship:** Relationships define how entities are connected or associated with each other. They explain the interactions and dependencies between entities. For example, the relationship between Customers and Orders represents that a customer can place multiple orders.
- **Reference Table:** A reference table is used to resolve many-to-many relationships between entities. It transforms the relationship into one-to-many or many-to-one relationships. For instance, in a system with Customers and Products, a reference table called OrderDetails can be used to link customers with the products they have ordered.

- **Database Logical Design:** It refers to designing the database within a specific data model of a database management system. It involves creating the structure, relationships, and rules of the database at a conceptual level.
- **Logical Design:** Logical design involves the creation of keys, tables, rules, and constraints within the database. It focuses on the logical structure and organization of data without considering specific implementation details.
- **Database Physical Design:** It encompasses the decisions related to file organization, storage design, and indexing techniques for efficient data storage and retrieval within the database.
- **Physical Model:** The physical model is a representation of the database that considers implementation-specific details such as file formats, storage mechanisms, and index structures. It translates the logical design into an actual database implementation.
- **Schema:** A schema refers to the complete description or blueprint of the database. It defines the structure, relationships, constraints, and permissions associated with the database objects.
- **Logical Schema:** A logical schema represents the theoretical design of a database. It is typically created during the initial stages of database design, similar to drawing a structural diagram of a house. It helps visualize the relationships and organization of The database entities and attributes

Levels of Data Abstraction:

Data modeling typically involves several levels of abstraction, including:

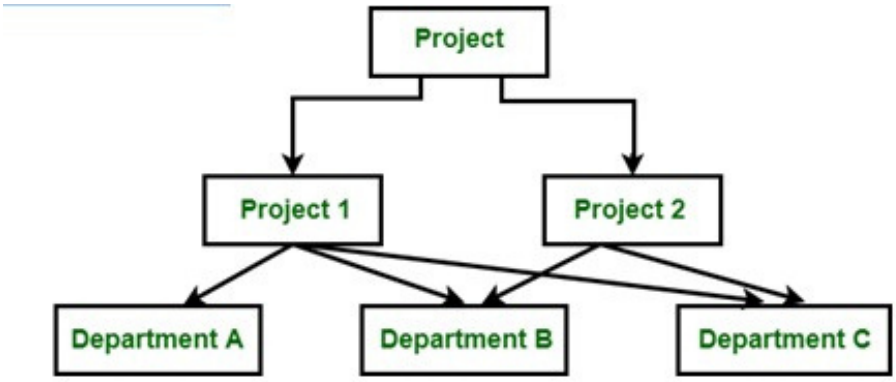
Conceptual level: This is the highest level of abstraction. It's about what data you need to store, and how it relates to each other. You can use diagrams or other visual representations to show this.
Example: You might decide that you need to store data about customers, products, and orders. You might also decide that there is a relationship between customers and orders, and between products and orders.
Logical level: This is the middle level of abstraction. It's about how the data will be stored and organized. You can use data modeling languages like SQL or ER diagrams to show this.
Example: You might decide to store the data in a relational database. You might create tables for customers, products, and orders, and define relationships between the tables.
Physical level: The physical level is the most basic or lowest abstraction. It concerns the particulars of how the data will be kept on disc. Data types, indexes, and other technical information fall under this category.
Example: You might decide to store the data in a specific database server, and use a specific data type for each column. You might also decide to create indexes to improve the performance of queries.

Important perspectives of a data model:

1.Network Model:

- The Network Model represents data as interconnected records with predefined relationships. It allows for many-to-many relationships and uses a graph-like structure. For example, in a company's database, employees can work on multiple projects, and each project can have multiple employees assigned to it. The Network Model connects employee records to project records through relationship pointers, enabling flexible relationships.

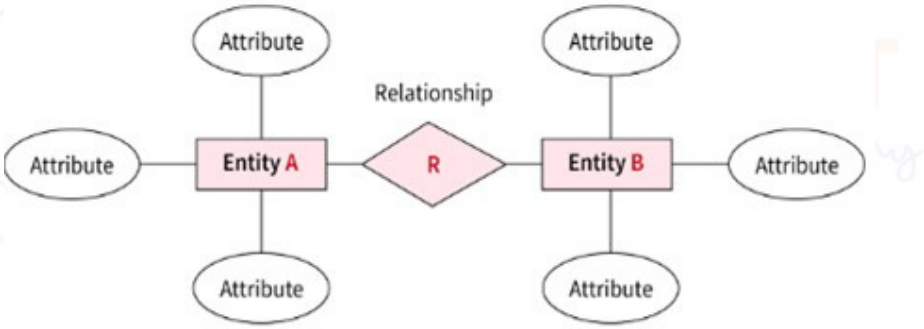
Check the below image:



2. Entity-Relationship (ER) Model:

• The ER Model represents data using entities, attributes, and relationships. Entities are real-world objects, attributes describe their properties, and relationships depict connections between entities. For instance, in a university database, entities could be Students and Courses, with attributes like student ID and course name. Relationships, such as "Student enrolls in Course," illustrate the associations between entities.

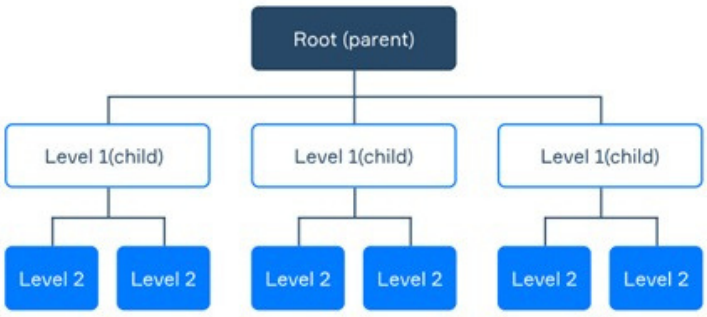
Check the below image:



3. Hierarchical Model:

• Data is arranged in a tree-like structure with parent-child relationships using the hierarchical model. There is one parent and several children per record. Consider the organizational hierarchy, where the CEO is at the top and is followed by the managers, employees, and department heads. These hierarchical links are graphically represented by the hierarchical model, allowing top-to-bottom or bottom-to-top navigation.

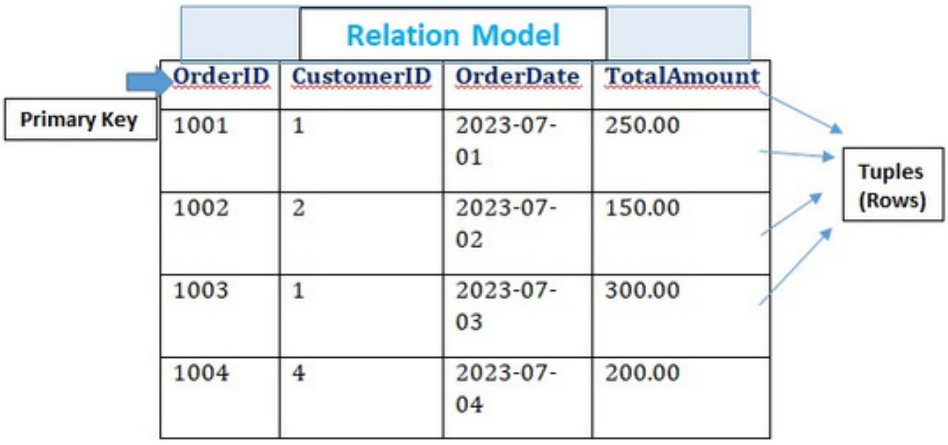
Check the below image:



4. Relational Model:

- The Relational Model organizes data into tables consisting of rows and columns. It create relationships between tables using primary and foreign keys. For example, in a customer and orders scenario, customer information is stored in one table, while order details are stored in another. The Relational Model connects the tables using a shared key, like a customer ID, to link relevant records.

Check the below image:



OrderID	CustomerID	OrderDate	TotalAmount
1001	1	2023-07-01	250.00
1002	2	2023-07-02	150.00
1003	1	2023-07-03	300.00
1004	4	2023-07-04	200.00

What is Database?

A database is like a digital warehouse where data is stored, organized, and managed efficiently. Database is a physical or digital storage system that implements a specific data model. Database is the actual implementation of that design, where the data is stored and managed. It acts as a central hub for information, making it easier to access and analyze data.

Key Components of a Database:

- **Data:** Data is the raw information stored in a database, such as customer details, product information, or financial records. It can be in different formats like text, numbers, dates, or images.
- **Tables:** Tables are like virtual spreadsheets within a database. They have rows and columns, where each row represents a specific record or instance, and each column represents a particular piece of data. For example, a table for customers may have columns like ID, Name, Address, and Contact.
- **Relationships:** Relationships define how tables are connected within a database. They establish associations based on shared data elements. For instance, a customer's ID in one table can be linked to their orders in another table. This helps maintain data consistency and enables efficient data retrieval.
- **Queries:** Queries are like search commands that allow users to extract specific data from the database. Users can search, filter, and sort data based on the criteria they specify. For example, a query can be used to find all customers who made a purchase in the last month.

There are various types of databases used in different scenarios based on their design and functionality. Here are some common types of databases:

• **Relational Databases (RDBMS):** Relational databases separate data into rows and columns and tables, and they build associations between tables using keys. They are frequently used in business applications that manipulate data using Structured Query Language (SQL). Oracle, MySQL, and Microsoft SQL Server are a few examples.
• **NoSQL Databases:** NoSQL databases are Non-relational databases that offer flexible data models without using a fixed schema. They can manage enormous amounts of semi- or unstructured data. MongoDB, Cassandra, and Couchbase are a few examples.
• **Object-Oriented Databases:** Similar to object-oriented programming, OODBs store data as objects. They are helpful for programs that deal with intricate interactions and data structures. Examples are ObjectDB and db4o.
• **Hierarchical Databases:** Hierarchical databases organize data in a tree-like structure, where each record has a parent-child relationship. They are suitable for representing hierarchical relationships, such as organization structures. IMS (Information Management System) is an example of a hierarchical database.
• **Network Databases:** Network databases are similar to hierarchical databases but allow for more complex relationships. They use a network model where records can have multiple parent and child records. CODASYL DBMS (Conference on Data Systems Languages) is an example of a network database.
• **Graph Databases:** Graph databases store data in a graph structure with nodes and edges. They are designed to represent and process relationships between data points efficiently. They are commonly used for social networks, recommendation engines, and network analysis. Examples include Neo4j and Amazon Neptune.
• **In-Memory Databases:** In-memory databases store data primarily in memory, resulting in faster data access and processing compared to disk-based databases. They are suitable for applications that require high-speed data operations. Examples include Redis and Apache Ignite.
• **Time-Series Databases:** Time-series databases are optimized for storing and retrieving time-stamped data, such as sensor data, financial data, or log files. They provide efficient storage and retrieval of time-series data for analysis. Examples include InfluxDB and Prometheus.

Let’s Understand DBMS and RDBMS:

Although they are both software system used to manage databases, DBMS (Database Management System) and RDBMS (Relational Database Management System) have different qualities.

Why required DBMS?

A DBMS is required to efficiently manage the flow of data within an organization. It handles tasks such as inserting data into the database and retrieving data from it. The DBMS ensures the consistency and integrity of the data, as well as the speed at which data can be accessed.

Why required RDMS?

Similarly, an RDBMS is required when we want to manage data in a relational manner, using tables and relationships. It helps in reducing data duplication and maintaining the integrity of the database. RDBMS ensures that data is stored in a structured manner, allowing for efficient querying and retrieval.

Differences between DBMS and RDBMS:

DBMS	RDBMS
Applications using DBMS save data in files.	RDBMS applications store data in a tabular form.
No relationship between data.	Related data stored in the form of table.
Normalization is not present.	**Normalization** is present.
Distributed databases are not supported by DBMS.	RDBMS supports distributed database.
It works with small quantity of data.	It works with large amount of data.
Security is less	More security measures provided.
Low software and hardware necessities	Higher software and hardware necessities.
Examples: XML Window Registry, Forxpro, dbaseIIIplus etc.	Examples: PostgreSQL, MySQL, Oracle, Microsoft Access, SQL Server etc.

What is Normalization?

The process of normalization data in a database ensures data integrity by removing duplication. A database must be split up into various tables, and linkages must be established between them. Different levels of normalization exist, such as 1NF (First Normal Form), 2NF (Second Normal Form), 3NF (Third Normal Form), and BCNF (Boyce-Codd Normal Form)

1NF (First Normal Form):

In 1NF, each column in a table contains only atomic values, meaning it cannot be further divided. There should be no repeating groups or arrays of values within a single column. Each row in the table should be uniquely identifiable. Here's an example:

Original Table:

CustomerID	Name	Phone no.
1	Jia ria	8978847383
2	John Doe	7899748899, 8899278299
3	Smith tie	9877382892

1NF Table:

CustomerID	Name	Phone no.
1	Jia ria	8978847383
2	John Doe	7899748899
3	Smith tie	9877382892
4	John Doe	8899278299

2NF (Second Normal Form):

In 2NF, the table is already in 1NF, and each non-key column is dependent on the entire primary key. If there are partial dependencies, those columns should be moved to a separate table. Here's an example:

Original Table:

OrderID	ProductID	ProductName	Category	Price
1	1	Laptop	Electronics	1000
2	2	Smartphone	Electronics	800
3	3	Laptop	Electronics	900

2NF Tables:

Table 1: Products

ProductID	ProductName	Category
1	Laptop	Electronics
2	Smartphone	Electronics

Table 2: Orders

OrderID	ProductID	Price
1	1	1000
2	2	800
3	1	900

3NF (Third Normal Form):

In 3NF, the table is already in 2NF, and there are no transitive dependencies. Non-key columns should not depend on other non-key columns. If there are such dependencies, those columns should be moved to a separate table. Here's an example:

Original Table:

CustomerID	OrderID	ProductID	Price	CustomerName	CustomerEmail
1	1	1	1000	John Doe	john@example.com
2	2	2	800	Jane Smith	jane@example.com
3	3	1	900	John Doe	johndoe@example.com

3NF Tables:

Table 1: Customers

CustomerID	CustomerName	CustomerEmail
1	John Doe	john@example.com
2	Jane Smith	jane@example.com
3	John Doe	johndoe@example.com

Table 2: Product

ProductID	ProductName
1	Laptop
2	Smartphone

BCNF (Boyce-Codd Normal Form):
BCNF is an advanced form of normalization that addresses certain anomalies that can occur in 3NF. It ensures that there are no non-trivial functional dependencies of non-key attributes on a candidate key. Achieving BCNF involves decomposing tables further if necessary.

Binary Relationships:

A binary relationship exists when two different relationship are involved. Accordingly, every entity in the connection has a unique association with one entity in the other entity. For instance, a passport can only be issued to one individual, and a person is only allowed to have one passport at a time. An illustration of a one-to-one relationship might be this.

Cardinality:

Cardinality refers to the number of instances of an entity that can be associated with an instance of another entity. There are four types of cardinality:

One-to-One

Each instance of one entity can only be linked to one instance of the other, and vice versa, in a one-to-one relationship. It is commonly used to represent one-to-one things in the actual world, such a person and their passport, and is the strictest kind of relationship.

Example: A person can have only one passport, and a passport can be issued to only one person.

One-to-Many

A one-to-many relationship means that each instance of one entity can be associated with multiple instances of the other entity, but each instance of the other entity can only be associated with one instance of the first entity. This is a common type of relationship, and it is often used to represent hierarchical relationships in the real world, such as a parent and their children.

Example: A customer can place multiple orders, but each order can only be placed by one customer.

Many-to-One

An entity in A is associated to no more than one entity in B in this particular cardinality mapping. Or, we may say that any number (zero or more) of entities or things in A can be connected to a unit or thing in B.

Example: A single surgeon performs many operations in a specific institution. A many-to-one relationship is one of these relationships.

Many-to-Many

A many-to-many relationship means that each instance of one entity can be associated with multiple instances of the other entity, and each instance of the other entity can also be associated with multiple instances of the first entity. This is the most common type of relationship, and it is often used to represent relationships where the order of the entities does matter, such as a student and their courses.

Example: A student can take multiple courses, and each course can be taken by multiple students.

Introduction to SQL:

The computer programming language SQL (Structured Query Language) was developed especially for managing and changing relational databases. It provides commands and statements to connect to databases, retrieve and modify data, construct database structures, and perform numerous data tasks. More details are provided below:

Definition and Purpose of SQL:

According to its definition and intended application, SQL is a declarative language utilized for relational database management. By constructing queries, it enables users to interact with databases to access, alter, and manage structured data. No matter what database management system is used underneath, SQL provides a standardized and efficient method for working with databases.

Why SQL?

Due to its adaptability and efficiency in maintaining relational databases, SQL is frequently used in data science and analytics. The main justifications for SQL's high value are as follows:

- The core activities of inserting, updating, and deleting data in relational databases are made available to data professionals via SQ. It gives a simple and effective method for changing data.
- SQL gives customers the ability to get particular data from relational database management systems. Users can provide criteria and conditions to retrieve the desired information by creating SQL queries.
- SQL is useful for expressing the structure of stored data. Users can define the structure, data types, and relationships of database tables as well as add, change, and delete them.
- SQL gives users the ability to handle databases and their tables efficiently. In order to increase the functionality and automation of database operations, it facilitates the construction of views, stored procedures, and functions.
- SQL gives users the ability to define and edit data that is kept in a relational database Data constraints can be specified by users, preserving the integrity and consistency of the data.
- Data Security and Permissions: SQL has tools for granting access to and imposing restrictions on table fields, views, and stored procedures. Giving users the proper access rights promotes data security.

SQL constraints

- Rules for the data in a table can be specified using SQL constraints.
- The kinds of data that can be entered into a table are restricted by constraints. This guarantees the reliability and accuracy of the data in the table. The action is stopped if there is a violation between the constraint and the data action.
- Column-level or table-level constraints are both possible. Table level restrictions apply to the entire table, while column level constraints just affect the specified column.

In SQL, the following restrictions are frequently applied:

NOT NULL: A column cannot have a NULL value by using the NOT NULL flag.
UNIQUE: A unique value makes sure that each value in a column is distinct.
PRIMARY KEY: A NOT NULL and UNIQUE combination. Identifies each table row in a unique way.
FOREIGN KEY: Prevent acts that would break linkages between tables.
CHECK - Verifies if the values in a column meet a certain requirement.
DEFAULT: If no value is specified, DEFAULT sets a default value for the column.
CREATE INDEX - Used to easily create and access data from the database.

NOT NULL constraint

- A column may by default contain NULL values.
- A column must not accept NULL values according to the NOT NULL constraint.
- This forces a field to always have a value, thus you cannot add a value to this field while adding a new record or updating an existing record.
  Syntax:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    price int);
```

Check Constraint:

- The value range that can be entered into a column is restricted by the CHECK constraint.
- Only specific values will be permitted for a column if you define a CHECK constraint on it.
- A table's CHECK constraint can be used to restrict the values in specific columns based on the values of other columns in the same row.

Example: Create check constraint:

```
CREATE TABLE employee (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CHECK (Age>=18));
```

SQL DEFAULT CONSTRAINT:

- A column's default value is set using the DEFAULT constraint.
- If no alternative value is supplied, the default value will be appended to all new records.
- Example: Create Default.

```sql
CREATE TABLE EMPLOYEE (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'LONDON');
```

By utilizing operations like GETDATE, the DEFAULT constraint can also be utilized to insert system data ()

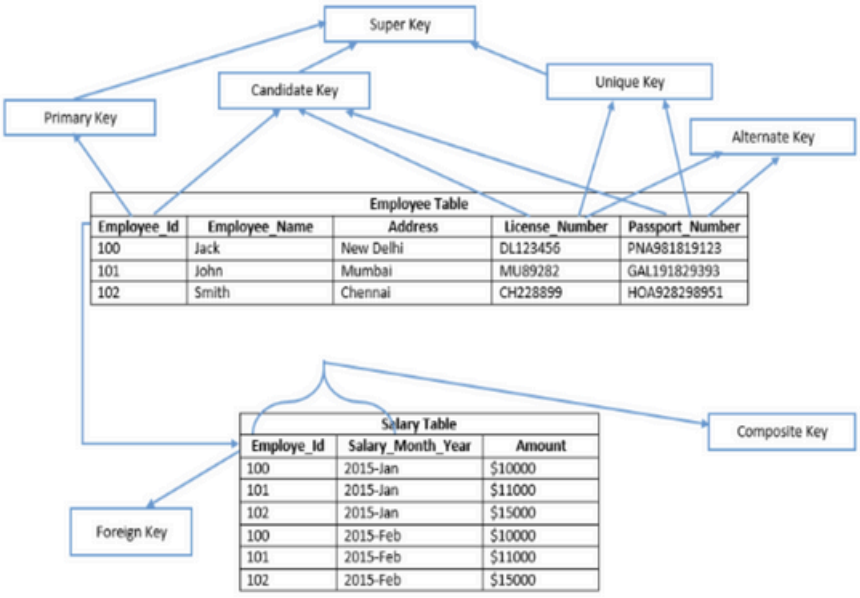
Various key types

We'll talk about keys and the many types that exist in SQL Server. Let's define keys to get this topic started.

What is the Key?

In RDBMS systems, keys are fields that take part in the following operations on tables:
To establish connections between two tables.

- To keep a table's individuality.
- To maintain accurate and consistent data in the database.
- Possibly speed up data retrieval by enabling indexes on column (s).



Types of keys

The following list includes the many key types that SQL Server supports:

- Candidate Key
- Primary Key
- Unique Key
- Alternate Key
- Composite Key
- Super Key
- Foreign Key

Candidate Key:

A candidate key is a table's primary key that has the potential to be chosen. There may be several candidate keys in a table, but only one can be chosen to serve as the main key.
Example: The candidate keys are License Number, Employee Id, , and Passport Number.

Primary Key

- The table's primary key was chosen as a candidate key to uniquely identify each record. Primary keys maintain unique values throughout the column and do not permit null values. Employee Id is the primary key of the Employee table in the example above. In SQL Server, a heap table's main key automatically builds a clustered index (a table which does not have a clustered index is known as a heap table). A table's nonclustered primary key can also be defined by explicitly specifying the kind of index.
- A table can have only one primary key, in SQL Server, the primary key can be defined using SQL commands below:
- CRETE TABLE statement (at the time of table creation) – In this case, system defines the name of primary key
- ALTER TABLE statement (using a primary key constraint) – User defines the name of the primary key

Example: Employee_Id is a primary key of Employee table.

Unique Key

- Similar to a primary key, a unique key prevents duplicate data from being stored in a column In comparison to the primary key, it differs in the following ways:
- One null value may be present in the column.
- On heap tables, it by default creates a nonclustered index.

Alternate Key

- The alternate key is a potential primary key for the table that has not yet been chosen.
- For instance, other keys include License Number and Passport Number.

Composite Key

Each row in a table is uniquely identified by a composite key, sometimes referred to as a compound key or concatenated key. A composite key's individual columns might not be able to identify a record in a certain way. It may be a candidate key or a primary key.

Example: To uniquely identify each row in the salary database, Employee Id and Salary Month Year are merged. Each entry cannot be individually identified by the Employee Id or Salary Month Year columns alone. The Employee Id and Salary Month Year columns in the Salary database can be used to build a composite primary key.

Super Key

- A super key is a group of columns from which the table's other columns derive their functional dependence. Each row in a table is given a unique identification by a collection of columns. Additional columns that are not strictly necessary to identify each row uniquely may be included in the super key. The minimal super keys, or subset of super keys, are the primary key and candidate keys.

SQL syntax:

SQL syntax is the set of rules that govern how SQL statements are written. It is a language that is used to interact with relational databases. SQL statements can be used to create, read, update, and delete data in a database.

The basic syntax of a SQL statement is:

- Keyword [parameter_1, parameter_2, ...]

The keyword is the name of the SQL statement. The parameters are the values that are passed to the statement.

For example, the following is a SELECT statement that selects the Name and Department ID columns from the Students table:

SELECT Name, Department ID

FROM Students;

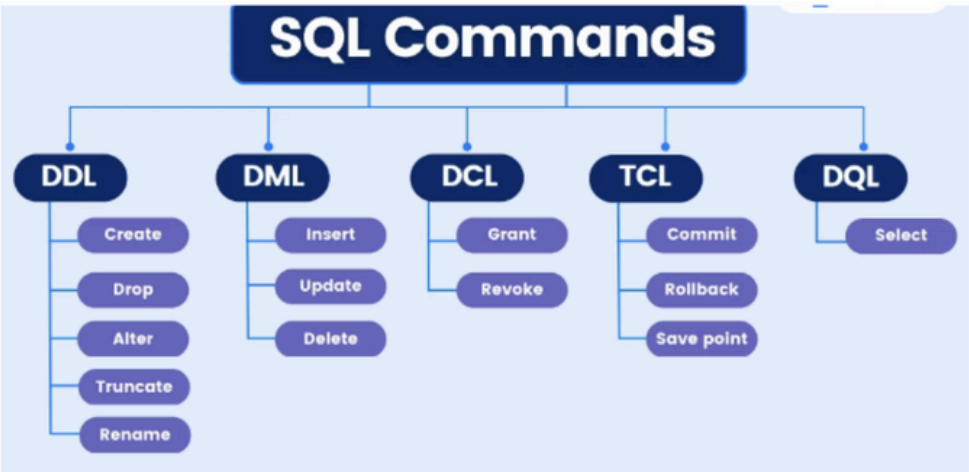
The keyword is SELECT, and the parameters are Name and Department ID.

SQL Commands

- The set of guidelines governing the writing of SQL statements is known as SQL syntax. The commands CREATE, DROP, INSERT, and others are employed in the SQL language to do the necessary activities.
- A table receives instructions from SQL statements. It is used to perform some Tasks on the database. Additionally, it is utilized to carry out particular duties, functions, and data inquiries. Table creation, data addition, table deletion, table modification, and user permission setting are just a few of the activities that SQL is capable of.

The five main categories into which these SQL commands fall are as follows:

- Data Definition Language (DDL)
- Data Query Language (DQL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- Transaction Control Language (TCL)



DDL (Data Definition Language)

The SQL statements that can be used to specify the database schema make up DDL, or Data Definition Language. It is used to create and modify the structure of database objects in the database and only works with descriptions of the database schema. Although data cannot be created, modified, or deleted with DDL, database structures can. In most cases, a typical user shouldn't use these commands; instead, they should use an application to access the database.

DDL commands list:

CREATE: Create the database or its objects using the CREATE (like database, table, index, function, views, store procedure, and triggers).

SYNTAX:

```
CREATE DATABASE database_name

CREATE TABLE table_name (
  column1 datatype,...);
```

DROP: Use the DROP command to remove objects from the database.
SYNTAX:

```
-- For database
DROP DATABASE database_name;
-- For table
DROP TABLE table_name;
```

ALTER: This is used to change the database's structure.
SYNTAX

```
ALTER TABLE table_name
ADD column_name datatype;

ALTER TABLE table_name
DROP COLUMN column_name;

ALTER TABLE table_name
MODIFY column_name datatype;
```

TRUNCATE: This function is used to eliminate every record from a table, along with any spaces set aside for the records.

SYNTAX:

```
TRUNCATE TABLE table_name;
```

RENAME: This command is used to rename an existing database object.
SYNTAX

```
-- rename table
RENAME table_name TO new_table_name;
-- renaming a column
ALTER TABLE table_name
RENAME COLUMN column_name TO new_column_name;
```

DQL (Data Query Language)

DQL is a sublanguage of SQL that is used to query data from a database.
It is a declarative language, which means that you tell the database what you want, not how to get it.
DQL statements are made up of keywords, operators, and values.
Some common DQL keywords include SELECT, FROM, WHERE, and ORDER BY.
SYNTAX

```
select *
from customer_orders
where customer_id = 100;
```

DML (Data Manipulation Language)

The majority of SQL statements are part of the DML which is used to manipulate data that is present in databases. It is part of the SQL statement in charge of managing database and data access. Essentially, DML statements and DCL statements belong together.

DML commands list:

INSERT: Data is inserted into a table using the INSERT command.
SYNTAX

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

UPDATE: A table's existing data is updated using UPDATE.
SYNTAX:

```
UPDATE table_name
SET column1 = new_value1, ...
WHERE condition;
```

DELETE: Delete records from a database table using the DELETE command.
SYNTAX:

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

LOCK: Concurrency under table management.
SYNTAX:

```
LOCK TABLE table_name IN lock_mode;
```

CALL: Call a Java or PL/SQL subprogram.
SYNTAX:

```
CALL subprogram_name
```

EXPLAIN PLAN: It outlines the data access route.

DCL (Data Control Language)

DCL comprises commands such as GRANT and REVOKE which largely deal with the rights, permissions, and other controls of the database system.

List of DCL commands:

GRANT: This command gives users access privileges to the database.
Syntax:

```
GRANT SELECT, UPDATE ON MY TABLE TO SOME USER, ANOTHER USER;
```

REVOKE: This command withdraws the user's access privileges supplied by using the GRANT command.

Syntax:

```
REVOKE SELECT, UPDATE ON MY TABLE FROM USER1, USER2;
```

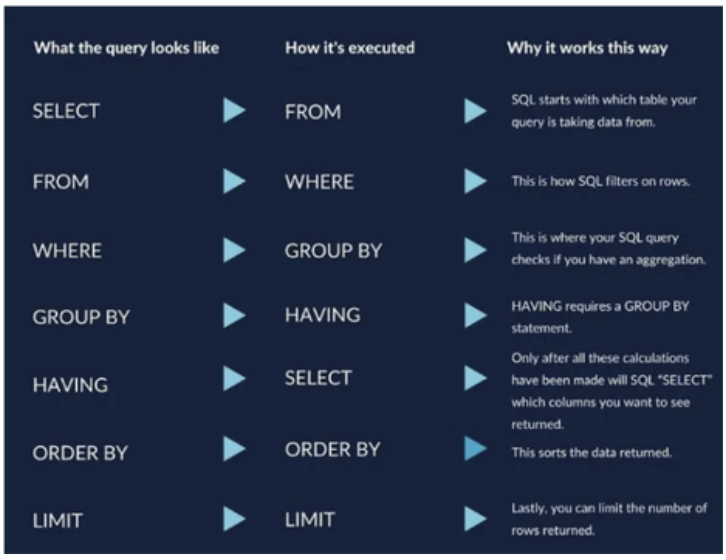
TCL (Transaction Control Language)

A group of tasks are combined into a single execution unit using transactions. Each transaction starts with a particular task and is completed once every activity in the group has been properly completed. The transaction fails if any of the tasks are unsuccessful. Therefore, there are only two outcomes for a transaction: success or failure. Here, you may learn more about transactions. As a result, the following TCL commands are used to manage how a transaction is carried out:

TCL commands list:

BEGIN: Opens a transaction with BEGIN
Syntax:
COMMIT: Completes a Transaction.
Syntax:
COMMIT;
Rollback: When an error occurs, a transaction is rolled back.
Syntax:
ROLLBACK;
SAVEPOINT: Creates a transactional save point.
Syntax:
The SAVEPOINT NAME;

SQL query Execution Order



SQL Statements:

The specific queries or operations entered into the database using the SQL language are known as SQL statements. They are employed to access, modify, or administer data stored in database tables. Keywords, expressions, clauses, and operators make up SQL statements.

SELECT STATEMENT: MySQL

To choose data from a database, use the SELECT statement.
The information received is kept in a result table known as the result-set.

For example, in the code below, we're choosing a row from a table called employees for a field called name.

Input

```
SELECT FullName
FROM employees;
```

Output

FullName
Filter...
John Snow
Walter White
Kuldeep Rana

SELECT *

All of the columns in the table we are searching will be returned when SELECT is used with an asterisk (*).

Input

```
SELECT *
FROM employees;
```

Output

EmpId	FullName	ManagerId	DateOfJoining	City
Filter...	Sort FullName	Filter...	Filter...	Filter...
121	John Snow	321	2019-01-31	Toronto
321	Walter White	986	2020-01-30	California
421	Kuldeep Rana	876	2021-11-27	New Delhi

SELECT DISTINCT

If there are duplicate records, SELECT DISTINCT will only return one of each; otherwise, it will provide data that is distinct.
The code below would only retrieve rows from the Employees table that had unique names.

Input

```
SELECT DISTINCT(fullname)
FROM employees;
```

Output

fullname
Filter...
John Snow
Walter White
Kuldeep Rana

SELECT INTO

The data supplied by SELECT INTO is copy from one table to another.

```
SELECT * INTO customer_orders
FROM employees;
```

Aliases (AS):

Aliases are frequently used to improve the readability of column names.
An alias only exists while that query is running.
By using the AS keyword, an alias is produced.
For instance, we're renaming the name column in the code below to fullname: AS renames a column or table with an alias that we can choose.

Input

```
SELECT FullName as FN
FROM employees;
```

Output

FN
Filter...
John Snow
Walter White
Kuldeep Rana
John Snow

Alias for Tables

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName='Around the Horn' AND c.CustomerID=o.CustomerID;
```

From

FROM identifies the table from which we are obtaining our data.

```
SELECT *
FROM EMPLOYEES
```

SQL WHERE Clause

Records can be filtered using the WHERE clause.
It is used to exclusively extract records that meet a certain requirement.
The WHERE clause is a part of an SQL SELECT statement that tells the database which rows to return. It uses a variety of operators to compare values in the database to values that you specify.

Input

```
SELECT *
FROM EMPLOYEES
WHERE EMPID=321
```

Output

EmpId	FullName	ManagerId	DateOfJoining	City
Filter...	Filter...	Filter...	Filter...	Filter...
321	Walter White	986	2020-01-30	California

NOTE: Single quotes must be used around text values in SQL. However, quotations should not be used around numeric fields:

Operators in the WHERE Clause:

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	less than or equal to
<>, !=	Not equivalent. Note: This operator may be written as ! In some SQL versions.
between	Within a specific range
like	Look for patterns
in	to designate a column's potential values in numerous ways

Where Clause- Tricky Questions:

1.Can you use aggregate functions in the WHERE clause?
No, you cannot use aggregate functions directly in the WHERE clause. Instead, you typically use them in the HAVING clause to filter the results of aggregate queries.
2.What is the difference between WHERE and HAVING clauses?
The WHERE clause is used to filter rows before they are grouped or aggregated, while the HAVING clause is used to filter grouped or aggregated results. The HAVING clause can use aggregate functions, unlike the WHERE clause.
3.Can you use multiple conditions in the WHERE clause?
Yes, you can combine multiple conditions using logical operators such as AND and OR. For example:

```
WHERE condition1 AND condition2
WHERE condition1 OR condition2
```

What happens if you use a column alias in the WHERE clause?
The WHERE clause is evaluated before the SELECT clause, so you cannot use column aliases defined in the SELECT clause directly in the WHERE clause. However, you can use the original column name or wrap the query in a subquery.
Remember, the WHERE clause is a powerful tool for filtering and retrieving specific data

from a table based on conditions. It allows you to narrow down the result set and perform complex queries to meet your specific requirements.

And, or, and not operators in SQL:

- Operators like AND, OR, and NOT can be used with the WHERE clause.
- To filter records based on multiple criteria, use the AND , OR operators:
- If every condition that is divided by AND is TRUE, the AND operator displays a record.
- If any of the terms divided by OR is TRUE, the OR operator outputs a record.
- If the condition(s) is/are NOT TRUE, the NOT operator displays a record.

AND - INPUT:

```
SELECT EMPID, FULLNAME
FROM EMPLOYEES
WHERE EMPID=121 AND FULLNAME="John Snow":
```

Output

EMPID	FULLNAME
Filter...	Filter...
121	John Snow
121	John Snow

OR – OPERATOR

Input

```
SELECT EMPID,
    FULLNAME
FROM EMPLOYEES
WHERE EMPID = 121
    OR FullName = "Walter White";
```

Output

EMPID	FULLNAME
Filter...	Filter...
121	John Snow
321	Walter White
121	John Snow

NOT OPERATOR

Input

```
SELECT EMPID, FULLNAME
FROM employees
WHERE NOT EMPID=121
```

Output

EMPID	FULLNAME
Filter...	Filter...
321	Walter White
421	Kuldeep Rana

SQL ORDER BY Keyword

The result set can be sorted in either ascending or descending order using the ORDER BY keyword.
Records are typically sorted using the ORDER BY keyword in ascending order. Use the DESC keyword to sort the records in descending order.

Example: Ascending Order by

Input

```
SELECT EMPID, FULLNAME
FROM employees
ORDER BY FullName
```

Example: Descending Order by

Input

```
SELECT EMPID, FULLNAME
FROM employees
ORDER BY FullName DESC
```

Output

EMPID	FULLNAME
Filter...	Filter...
321	Walter White
421	Kuldeep Rana
121	John Snow
121	John Snow

Example of ORDER BY Several Columns

In this example as you can see SQL Statement selected all column from athlete_event3 table and order by with multiple column.

Input

```
SELECT *
FROM athlete_events3
ORDER BY  Season, City DESC
```

Output

	Season	City	Sport
	Filter...	Filter...	Filter...
	Summer	Tokyo	Boxing
	Summer	Tokyo	Shooting
	Summer	Tokyo	Hockey
	Summer	Tokyo	Athletics
	Summer	Tokyo	Athletics

INSERT INTO STATEMENT

To add new records to a table, use the INSERT INTO statement.

INSERT INTO Syntax

There are two methods to format the INSERT INTO statement:

1. Specify the values to be inserted together with the column names.
2. You do not need to provide the column names in the SQL query if you are adding values to every column of the table. However, make sure the values are arranged in the same order as the table's column headings. In this case, the syntax for INSERT INTO would be as follows:

Input

```
INSERT INTO EMPLOYEES(EMPID, FULLNAME, MANAGERID, DATEOFJOINING, CITY)
VALUES(333,"JAY RAO",999,2019-03-20,"LONDON")
```

Output

EmpId	FullName	ManagerId	DateOfJoining
121	John Snow	321	2019-01-31
321	Walter White	986	2020-01-30
421	Kuldeep Rana	876	2021-11-27
121	John Snow	321	1987
333	JAY RAO	999	1996

Additionally, you can insert data entry to certain columns.

IS NULL OPERATOR

Testing for empty values is done with the IS NULL operator (NULL values).
To check for non-empty values, use the IS NOT NULL operator (NOT NULL values).
Example: The SQL statement below lists every employee whose "city" fiel
has a NULL value:

```
SELECT FullName, EmpId, city
FROM EMPLOYEES
where city is null;
```

SQL DELETE Statement

Existing records in a table can be deleted using the DELETE statement.
Reminder: Take care while eliminating records from a table! In the DELETE statement, pay attention to the WHERE clause. Which record(s) should be removed is specified by the WHERE clause. All records in the table will be erased if the WHERE clause is left off.
Example: Check below , delete statement deleteing city toronto from employees table

```
DELETE
FROM EMPLOYEES
where city ='Toronto';
```

Output: Toronto city deleted.

FullName	ManagerId	DateOfJoining	City
Walter White	986	2020-01-30	California
Kuldeep Rana	876	2021-11-27	New Delhi
JAY RAO	999	1996	LONDON

LIMIT CLAUSE:

The number of rows to return is specified using the LIMIT clause.
On huge tables with tens of thousands of records, the LIMIT clause is helpful.
Performance may be affected if many records are returned.
The SELECT TOP clause is not supported by all database management systems.
In contrast to Oracle, which employs FETCH FIRST n ROWS ONLY and ROWNUM,
MySQL supports the LIMIT clause to restrict the number of records that are selected.

EXAMPLE: from employees table only 2 records display

Input

```
SELECT *
FROM EMPLOYEES
LIMIT 2;
```

Output

EmpId	FullName	ManagerId	DateOfJoining
Filter...	Filter...	Filter...	Filter...
321	Walter White	986	2020-01-30
421	Kuldeep Rana	876	2021-11-27

SQL UPDATE Statement

The existing records in a table can be changed using the UPDATE command.
When updating records in a table, take caution! In the UPDATE statement, pay attention to the WHERE clause. The record(s) to be modified are specified by the WHERE clause.
The table's records will all be updated if the WHERE clause is left off!

UPDATE A multiple Of Records

How many records are updated is determined by the WHERE clause.
Note: When updating records, exercise caution. ALL records will be updated if the WHERE clause is not included!
Example: Update records fullname rana where city is London

```
UPDATE employees
SET FULLNAME='RANA'
WHERE CITY='London';
```

Output

FullName	ManagerId	DateOfJoining	City
Filter...	Filter...	Filter...	Filter...
Walter White	986	2020-01-30	California
Kuldeep Rana	876	2021-11-27	New Delhi
RANA	999	1996	LONDON

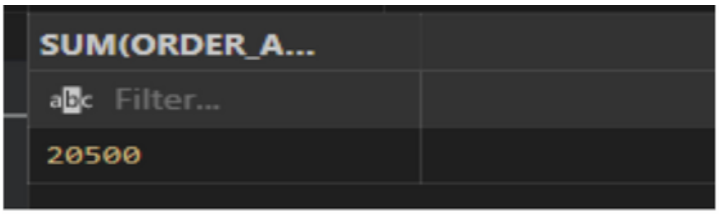
SQL aggregation function:

A number of aggregation functions exist in SQL that can be used to conduct calculations on a collection of rows and return a single value. Here are a few aggregation functions that are frequently used:

SUM()= Determines the total of a column or expression using SUM().
for instance:

```
SELECT SUM(ORDER_AMOUNT)
FROM CUSTOMER_ORDERS;
```

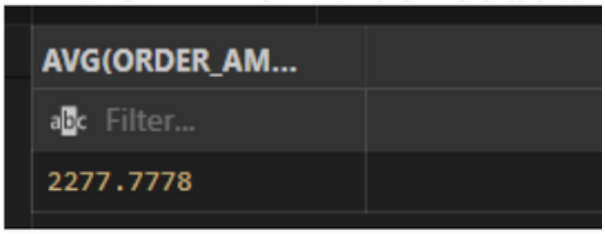
Output



AVG()= Determines the average of a column or phrase using AVG().
for instance:

```
SELECT AVG(ORDER_AMOUNT)
FROM CUSTOMER_ORDERS;
```

Output

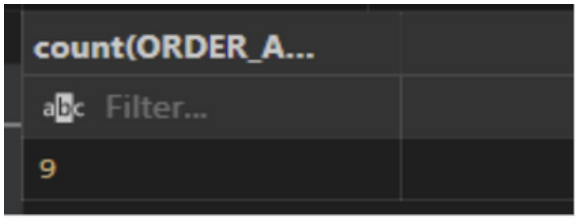


COUNT() = Returns the number of rows in a table or the number of rows that meet a condition using the COUNT() function.

Example:

```
SELECT COUNT(ORDER_AMOUNT)
FROM CUSTOMER_ORDERS;
```

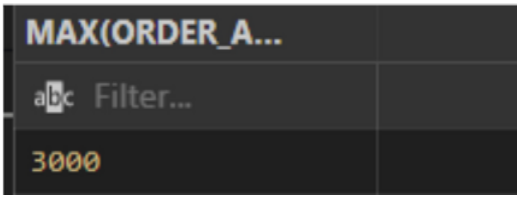
Output



MAX()= The MAX() function displays the highest value in a column or expression.
Example:

```
SELECT MAX(ORDER_AMOUNT)
FROM CUSTOMER_ORDERS;
```

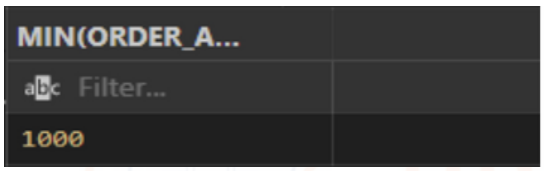
Output



MIN()= Function display the lowest value in the column.

```
SELECT MIN(ORDER_AMOUNT)
FROM CUSTOMER_ORDERS;
```

Output



Like Operator:

To look for a specific pattern in a column, use the LIKE operator in a WHERE clause.
Two wildcards are frequently combined with the LIKE operator:
The percent sign (%) denotes a character or many characters.
The underscore character (_) stands for a single character.

Please take note that MS Access substitutes an asterisk (*) for the percent sign (%) and a question mark (?) for the underscore ( ).
Additionally, you can combine the underscore and the % sign!

Operator	Discription
WHERE FULLNAME LIKE 'a%'	Searches for any values beginning with "a,"
WHERE FULLNAME LIKE '%a'	Searches for values ending in "a."
WHERE FULLNAME LIKE '%or%'	identifies any values that contain the word "or" anywhere.
WHERE FULLNAME LIKE '_r%'	in the second position is done by using the WHERE
WHERE FULLNAME LIKE 'a_%'	Finds any values that begin with "a" and have at least two characters in length using the WHERE
WHERE FULLNAME LIKE 'a__%'	Search any values that start with "a" and are at least 3 characters in length.
WHERE FULLNAME LIKE 'a%o'	Search any values that start with "a" and ends with "o".

Example: Find fullname starts with w from employees table

```
SELECT *
FROM EMPLOYEES
WHERE FULLNAME LIKE 'W%';
```

Output

EmpId	FullName	ManagerId	DateOfJoining
Filter...	Filter...	Filter...	Filter...
321	Walter White	986	2020-01-30

IN SQL Operator

You can define several values in a WHERE clause by using the IN operator.
The numerous OR conditions are abbreviated as the IN operator.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

OR

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

Example: Select all records where city in california and new delhi.

```
SELECT *
FROM employees
WHERE CITY IN ('CALIFORNIA','NEW DELHI')
```

Output

FullName	ManagerId	DateOfJoining	City
Filter...	Filter...	Filter...	Filter...
Walter White	986	2020-01-30	California
Kuldeep Rana	876	2021-11-27	New Delhi

BETWEEN OPERATOR:

The BETWEEN operator chooses values from a predetermined range. The values could be text, integers, or dates.
The BETWEEN operator includes both the begin and finish variables.

BETWEEN EXAMPLE: select order amount between 2100 to 2900 from customer orders

```
SELECT *
FROM customer_orders
WHERE ORDER_AMOUNT BETWEEN 2100 AND 2900;
```

order_id	customer_id	order_date	order_amount
Filter...	Filter...	Filter...	Filter...
2	200	2022-01-01T00:00:00.000Z	2500
3	300	2022-01-01T00:00:00.000Z	2100
5	400	2022-01-02T00:00:00.000Z	2200
6	500	2022-01-02T00:00:00.000Z	2700

Not between:

Example: select order amount which not in range 2100 to 2900 from customer orders table

```
SELECT *
FROM customer_orders
WHERE ORDER_AMOUNT NOT BETWEEN 2100 AND 2900
```

Output

order_id	customer_id	order_date	order_amount
Filter...	Filter...	Filter...	Filter...
1	100	2022-01-01T00:00:00.000Z	2000
4	100	2022-01-02T00:00:00.000Z	2000
7	100	2022-01-03T00:00:00.000Z	3000
8	400	2022-01-03T00:00:00.000Z	1000
9	600	2022-01-03T00:00:00.000Z	3000

BETWEEN with IN :

Example: select all records where order amount range between 2100 and 2900 and also order id 1,4,8

```
SELECT *
FROM customer_orders
WHERE ORDER_AMOUNT NOT BETWEEN 2100 AND 2900
AND ORDER_ID IN(1,4,8)
```

Output

order_id	customer_id	order_date	order_amount
Filter...	Filter...	Filter...	Filter...
1	100	2022-01-01T00:00:00.000Z	2000
4	100	2022-01-02T00:00:00.000Z	2000
8	400	2022-01-03T00:00:00.000Z	1000

Between- Tricky Questions:

Does the BETWEEN operator include the boundary values?
Yes, the BETWEEN operator includes the boundary values. It is an inclusive range operator, so values equal to value1 or value2 will be included in the result.
Can the BETWEEN operator be used with non-numeric data types?
Yes, the BETWEEN operator can be used with non-numeric data types such as dates, strings, or timestamps. It compares the values based on their inherent order or alphabetic sequence.
What happens if value1 is greater than value2 in the BETWEEN operator?
The BETWEEN operator still functions correctly even if value1 is greater than value2. It will return rows where the column value falls within the specified range, regardless of the order of value1 and value2.
Can you use the NOT operator with the BETWEEN operator? Yes, you can use the NOT operator to negate the result of the BETWEEN operator. For example:

```
WHERE column_name NOT BETWEEN value1 AND value2
```

Can you use the BETWEEN operator with NULL values?
No, the BETWEEN operator cannot be used with NULL values because NULL represents an unknown value. Instead, you can use the IS NULL or IS NOT NULL operators to check for NULL values. **Can you use the BETWEEN operator with datetime values and timestamps?** Yes, the BETWEEN operator can be used with datetime values and timestamps to filter rows within a specific date or time range. Remember, the BETWEEN operator provides a convenient way to filter rows based on a range of values. It is widely used for various data types and allows for inclusive range comparisons. Understanding its behavior and nuances will help you accurately retrieve the desired data from your database.

SQL joins

Data is kept in several tables that are connected to one another in relational databases like SQL Server, Oracle, MySQL, and others via a common key value. As a result, it is frequently necessary to combine data from two or more tables into one results table. The SQL JOIN clause in SQL Server makes this simple to do. A JOIN clause is used to combine rows from those tables based on a shared column between two or more of the tables. They allow you to retrieve data from multiple tables in a single query, based on common data points. Here's a detailed explanation of SQL joins:

Types of JOINS:

Inner join/ Equijoin	Full outer join	Natural join
Left join	Cross join/ Cartesian Join	
Right join	Self join	

Inner join/ Equijoin:

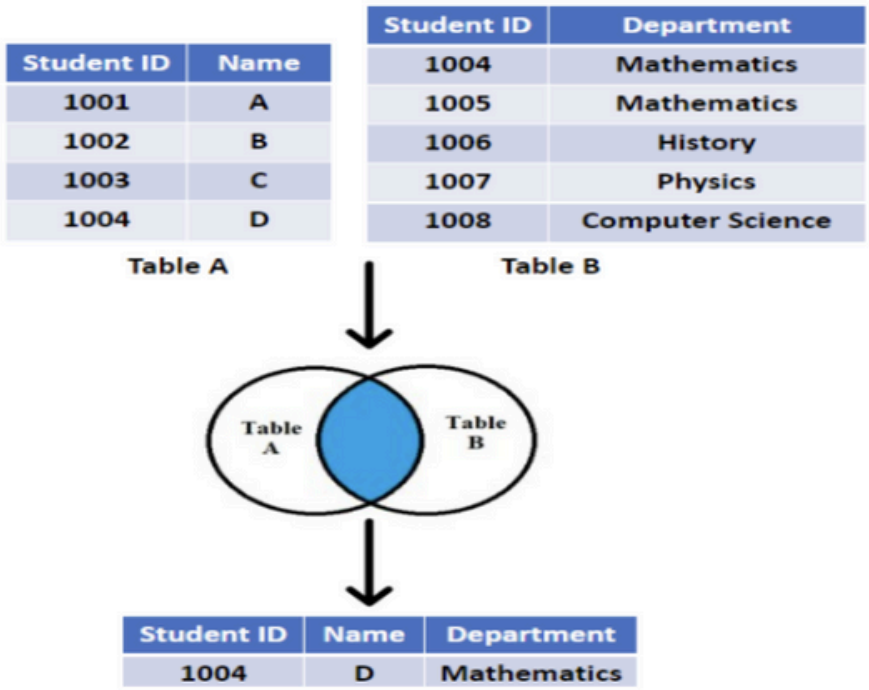
If the criteria is met, the INNER JOIN keyword selects all rows from both tables. By merging all rows from both tables that meet the requirement—that is, have the same value for the shared field—this keyword will provide a result set. Inner joins are a useful way to combine data from two tables where there is a relationship between the two tables.

Here are some of the benefits of using inner joins:

They can be used to get a more complete picture of a set of data.
They can be used to identify customers who have placed an order.
They can be used to join tables that have different schemas.

Here are some of the limitations of using inner joins:

They can return less data than other types of joins.
They can be inefficient if there are a lot of rows in the two tables that do not have matching values.
The join condition is typically an equality comparison between the related columns in the two tables.
You can use table aliases to simplify the syntax and improve readability.
You can join more than two tables by extending the join operation using additional INNER JOIN clauses.
Check the below image.



Example: First create table T1, and T2.

```
CREATE TABLE t1 (id INT);
CREATE TABLE t2 (id2 INT);
```

Insert values:

```
INSERT INTO t1 (id) VALUES (1), (1), (1), (1), (2), (3), (NULL);
INSERT INTO t2 (id2) VALUES (1), (1), (1), (1), (2), (3), (4), (5), (NULL);
```

Table- T1

id
1
1
1
1
2
3
NULL

Table- T2

id2
1
1
1
1
2
3
4
5
NULL

Input

```
SELECT * FROM t1
INNER JOIN t2 ON t1.id = t2.id2;
```

Output

id	id2
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
2	2
3	3

Right join:

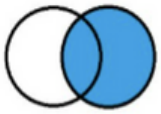
This join gives back all the rows from the table on the right side of the join as well as any matching rows from the table on the left. The result-set will include null for any rows for which there is no matching row on the left. RIGHT OUTER JOIN is another name for RIGHT JOIN. Similar to LEFT JOIN is RIGHT JOIN.

In simple language,

All rows from the right table are returned by the right join.
The result will exclude any rows that are present in the left table but not the right table.
Limitations: Complex queries: When using RIGHT JOINs in complex queries involving multiple tables, it can be challenging to maintain clarity and understand the relationship between tables. Careful consideration of table order and join conditions is necessary to produce accurate results.
Code readability: RIGHT JOINs, especially in more complex queries, can make the SQL code less readable and harder to interpret.
Check the below image.

Student ID	Name
1001	A
1002	B
1003	C
1004	D

Student ID	Department
1004	Mathematics
1005	Mathematics
1006	History
1007	Physics
1008	Computer Science



Student ID	Name	Department
1004	D	Mathematics
1005	NULL	Mathematics
1006	NULL	History
1007	NULL	Physics
1008	NULL	Computer Science

Example

```
SELECT * FROM t1
RIGHT JOIN t2 ON t1.id = t2.id2;
```

Output

id	id2
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
2	2
3	3
NULL	4
NULL	5

Left join:

The rows that match for the table on the right side of the join are returned along with all of the rows from the table on the left side of the join. The result-set will include null for all rows for which there is no matching row on the right side. LEFT OUTER JOIN is another name for LEFT JOIN. Left joins are a useful way to get all the data from one table, even if there is no matching data in another table.

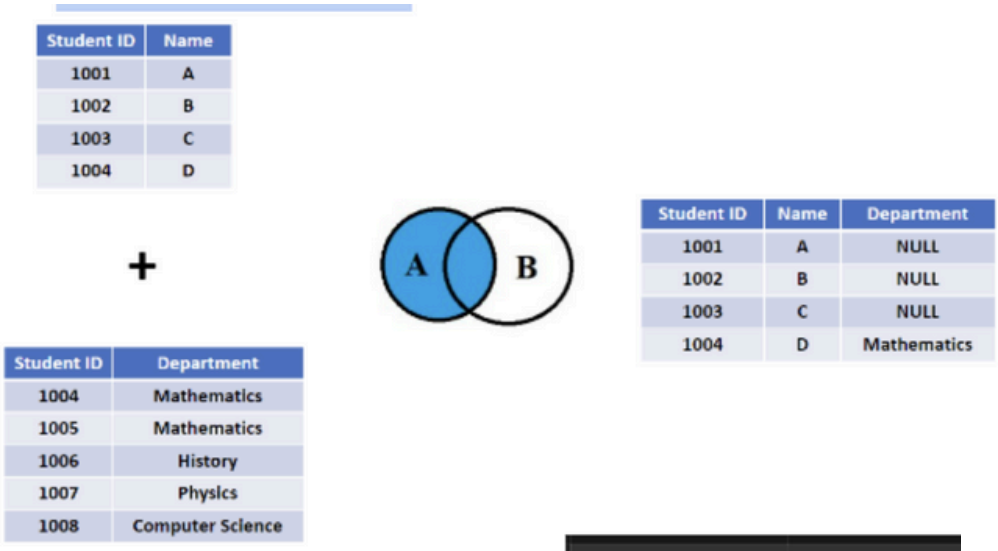
Here are some of the benefits of using left joins:

They can be used to get a complete overview of a set of data.
They can be used to identify customers who have not yet placed an order.
They can be used to join tables that have different schemas.

Here are some of the limitations of using left joins:

They can return a lot of data, which can make it difficult to analyze.
They can be inefficient if there are a lot of rows in the right table that do not have a matching row in the left table.

Check the below image.



Example: Left join

```
SELECT * FROM t1
LEFT JOIN t2 ON t1.id = t2.id2;
```

id	id2
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	1
2	2
3	3
NULL	NULL

Full Outer Join:

By combining the results of both LEFT JOIN and RIGHT JOIN, FULL JOIN produces the result-set. The rows from both tables are all included in the result-set. The result-set will contain NULL values for the rows where there was no match. Full outer joins are a useful way to get all the data from two tables,his can be helpful for getting a complete overview of a set of data.

Here are some of the benefits of using full outer joins:

They can be used to get a complete overview of a set of data.
They can be used to identify customers who have not yet placed an order.
They can be used to identify orders that have not yet been assigned to a customer.

Here are some of the limitations of using full outer joins:

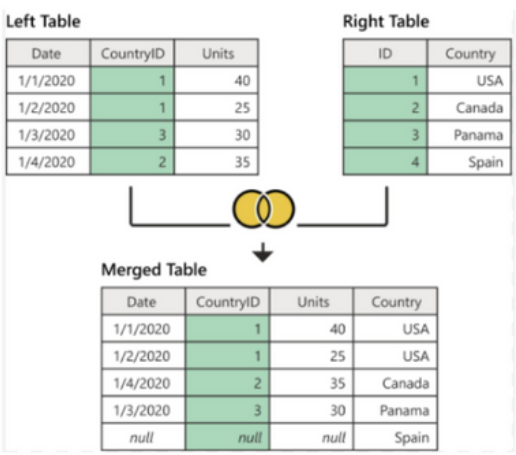
They can return a lot of data, which can make it difficult to analyze.
They can be inefficient if there are a lot of rows in the two tables that do not have matching values.

Benefits of CROSS JOIN:

Cartesian product: A CROSS JOIN generates a Cartesian product of the two tables involved, combining every row from the first table with every row from the second table. This can be useful in scenarios where you need to explore all possible combinations between two sets of data.
Data expansion: CROSS JOIN can be used to create additional rows, especially when working with dimension tables or reference tables that have a small number of rows. This expansion can be beneficial for creating test data or generating comprehensive reports.

Limitations of CROSS JOIN:

Result set size: The result set of a CROSS JOIN can grow rapidly, especially when joining large tables or multiple tables. The Cartesian product generates a number of rows equal to the product of the number of rows in the joined tables, which can lead to a significant increase in the dataset's size. **Performance impact:** Due to the potential for a large result set, a CROSS JOIN operation can have a performance impact on the database. It can consume significant resources and take longer to execute compared to other types of joins. Check the below image:



Date	CountryID	Units
1/1/2020	1	40
1/2/2020	1	25
1/3/2020	3	30
1/4/2020	2	35

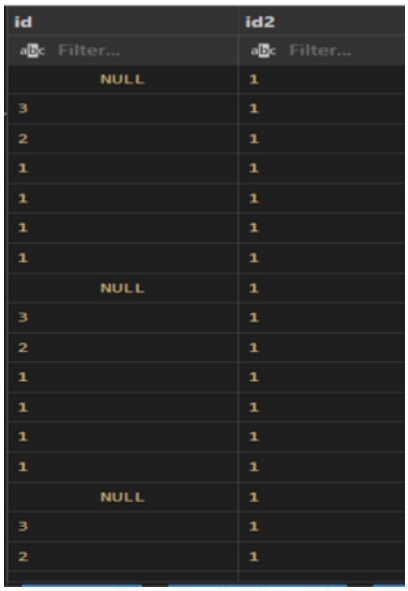
ID	Country
1	USA
2	Canada
3	Panama
4	Spain

Date	CountryID	Units	Country
1/1/2020	1	40	USA
1/2/2020	1	25	USA
1/4/2020	2	35	Canada
1/3/2020	3	30	Panama
null	*null*	*null*	Spain

Input

```
SELECT * FROM t1
Cross JOIN t2;
```

Output



id	id2
Filter...	Filter...
NULL	1
3	1
2	1
1	1
1	1
1	1
1	1
NULL	1
3	1
2	1
1	1
1	1
1	1
1	1
NULL	1
3	1
2	1

Natural Join:

The NATURAL JOIN is a type of join operation in SQL that combines tables based on columns with the same name and data type. It automatically matches columns between tables and returns only the matching rows. Here's a detailed explanation of the NATURAL JOIN, and the difference between NATURAL JOIN and CROSS JOIN:
Synatx:

```
SELECT * FROM t1
NATURAL JOIN t2;
```

Benefits of NATURAL JOIN:
Simplicity:NATURAL JOIN simplifies the join operation by automatically matching columns with the same name and data type between the joined tables. It eliminates the need to explicitly specify the join conditions, reducing the complexity and potential for errors in the query.
Readability: NATURAL JOIN can enhance code readability as it reflects the inherent relationship between tables based on column names. It improves query comprehension and makes the code more intuitive and self-explanatory.

Limitations of NATURAL JOIN:

Ambiguous column names: If the joined tables contain columns with the same name but different meanings, using NATURAL JOIN can lead to confusion or produce incorrect results. It relies solely on column names and data types, disregarding any semantic differences.
Lack of flexibility: NATURAL JOIN provides limited control over the join conditions. It may not suit complex scenarios where more specific or custom join conditions are required.
Performance implications: NATURAL JOIN might impact performance, especially when joining large tables or in cases where indexing is not optimized for the matching columns.

Difference between CROSS JOIN and NATURAL JOIN:

CROSS JOIN: A CROSS JOIN, or Cartesian join, combines every row from the first table with every row from the second table, resulting in a Cartesian product. It does not consider any column matching or relationships between tables. It generates a large result set that includes all possible combinations between the joined tables.
NATURAL JOIN: A NATURAL JOIN combines tables based on columns with the same name and data type. It automatically matches the columns between tables and returns only the matching rows. It considers the column names as the join condition, without explicitly specifying it.

Benefits of Self Joins:

Comparing related data: Self joins allow you to compare and analyze related data within the same table. For example, you can compare sales data for different time periods or evaluate hierarchical relationships within organizational data. **Establishing relationships:** Self joins enable you to establish relationships between rows within a table. This is common in scenarios where data has a hierarchical structure, such as an employee table with a self-referencing manager column. **Simplifying complex queries:** Self joins simplify complex queries by enabling you to consolidate related information in a single result set. This can help in performing calculations, aggregations, or generating reports based on self-related data.

Limitations of Self Joins:

Performance impact: Self joins can have a performance impact, especially on large tables or when the join condition is not optimized. It is important to ensure proper indexing on the relevant columns for improved performance.
Code complexity: Self joins can make the SQL code more complex and potentially harder to read and understand. It is crucial to provide clear aliases and document the purpose and logic of the self join for easier comprehension and maintenance.

Tricky questions related to joins:

What is a self join, and when would you use it?
A self join is a join operation where a table is joined with itself. It is useful when you want to compare rows within the same table or establish relationships between different rows in the table, such as hierarchical data or comparing related records.

Can you perform a JOIN operation without specifying the JOIN condition?
No, specifying the join condition is essential for performing a join operation in SQL. It defines the relationship between the tables being joined and determines how the rows are matched.
How does a LEFT JOIN differ from a RIGHT JOIN?
A LEFT JOIN returns all rows from the left table and the matching rows from the right table. A RIGHT JOIN, on the other hand, returns all rows from the right table and the matching rows from the left table.
What is the result of joining a table with itself using a CROSS JOIN?
Joining a table with itself using a CROSS JOIN, also known as a Cartesian join, results in the Cartesian product of the table. It combines every row from the table with every other row, resulting in a potentially large result set.
Can you JOIN more than two tables in a single SQL query?
Yes, it is possible to join more than two tables in a single SQL query. This is often required when you need to combine data from multiple tables to retrieve the desired information.
How can you simulate a FULL OUTER JOIN in a database that does not support it?
In databases that do not support a FULL OUTER JOIN, a FULL OUTER JOIN can be simulated by combining a LEFT JOIN and a RIGHT JOIN using the UNION operator.
What is the difference between a natural join and an equijoin?
A natural join is a join operation that automatically matches columns with the same name and data type between the joined tables. An equijoin, on the other hand, is a join operation that explicitly specifies the equality condition between columns in the joined tables.
Can you use the WHERE clause to perform a JOIN operation?
While it is possible to use the WHERE clause to filter rows in a join operation, it is generall recommended to use the JOIN condition in the ON clause for specifying the relationship between the tables being joined.
How can you exclude rows that match in a JOIN operation?
To exclude rows that match in a JOIN operation, you can use an OUTER JOIN (LEFT JOIN or RIGHT JOIN) and check for NULL values in the columns of the non-matching table. Rows with NULL values indicate that they exist in one table but not in the other.
Can you join tables with different data types in SQL?
Yes, it is possible to join tables with different data types in SQL as long as the join condition is based on compatible data types. SQL will automatically perform necessary data type conversions if possible.
How can you optimize the performance of a JOIN operation?
To optimize the performance of a JOIN operation, you can ensure that the join columns are properly indexed, use appropriate join types (e.g., INNER JOIN instead of CROSS JOIN), and apply relevant filtering conditions to reduce the result set size.
Is it possible to JOIN tables based on non-matching columns?
Yes, it is possible to join tables based on non-matching columns by using conditions other than equality in the JOIN clause, such as using greater than or less than operators.
What are the implications of using a Cartesian product (CROSS JOIN)?
The Cartesian product can result in a large result set, especially if the joined tables have many rows. It can consume significant resources, impact query performance, and lead to unintended data duplication if not used carefully.
Can you JOIN tables that have different column names?
Yes, it is possible to join tables that have different column names. In such cases, you can specify the column names explicitly in the join condition using aliases or by using the ON clause to specify the relationship between the columns with different names.

GROUP BY Statement:

In database management systems like SQL, the group by clause is a strong tool fo categorizing rows of data based on one or more columns. It enables data aggregation and summarization, delivering insightful data and facilitating effective analysis. The COUNT(), MAX(), MIN(), SUM(), and AVG() aggregate functions are frequently used with the GROUP BY statement to group the result set by one or more columns. Although this feature has many advantages, it also has some drawbacks and particular applications.

Benefits of Group By:

- Data Summarization: Grouping enormous datasets according to specified criteria enables us to summarize them. It makes it possible to compute several summary statistics for each group, including counts, averages, sums, minimums, maximums, and more. Understanding the broad traits and patterns in the data is made easier by this summary.
- Simplified Data Analysis: Group by streamlines the analysis process by combining relevant data. It aids in the discovery of patterns, trends, and connections in the data. A clearer image is created by grouping data according to pertinent criteria, which also facilitates efficient decision-making.

Restrictions for Group By:

- Data Loss: Only aggregated results are shown when using the Group by clause; the original detailed information is lost. This could be a drawback if specific data points or individual records must be scrutinized for more in-depth research. In some circumstances, complex computations or unique aggregations might be necessary, which the Group by clause might not directly handle.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Example: sum up price for each city wise from aemf2 table.

```
SELECT SUM(PRICE), CITY
FROM aemf2
GROUP BY CITY;
```

Output:

SUM(PRICE)	CITY
Filter...	Filter...
1192074.613931599	Amsterdam
801208.9611635196	Athens
832204.2497717799	Barcelona
607546.0409463913	Berlin
709937.4911909413	Budapest
1372806.988745241	Lisbon
2625250.0233064746	Paris
1854073.1302258035	Rome
854477.2456391889	Vienna

HAVING clause

Since the WHERE keyword cannot be used with aggregate functions, the HAVING clause was added to SQL
Utilizing aggregate function-based conditions, the HAVING clause in SQL is used to restrict the results of a query. To filter grouped data, it is frequently combined with the GROUP BY clause. After grouping and aggregation, the HAVING clause enables you to apply criteria to the grouped data. In most cases, the HAVING clause is used in conjunction with the GROUP BY clause.

The HAVING Clause's advantages include:

- Grouped Data Filtering: The HAVING clause's capacity to filter grouped data depending on conditions is its main advantage. You can use it to specify complicated conditions that incorporate aggregate functions and column values to limit the result set to only the groups that match particular requirements. Flexible Aggregate Filtering: The HAVING
- clause allows for flexible aggregate filtering of data. It enables you to specify constraints on an aggregate function's output.

  The having clause has some restrictions.

  Order of Evaluation: The GROUP BY and aggregate procedures are assessed before the HAVING clause. Because they are not yet available during the evaluation of the HAVING condition, column aliases and aggregate functions established in the SELECT clause cannot be used in the HAVING clause. Performance Impact: When working with huge datasets, the HAVING clause may have an effect on performance.

  Syntax:

-

```
SELECT column1, column2,...
FROM table
GROUP BY column1, column2,...
HAVING condition;
```

Example: count the person capacity who have grater then 2 with each city from aemf2 table.

Output:

count(person_ca...	city
Filter...	Filter...
2080	Amsterdam
5280	Athens
2833	Barcelona
2484	Berlin
4022	Budapest
5763	Lisbon
6688	Paris
9027	Rome
3537	Vienna

Difference between Group by and having:

Purpose:
GROUP BY: The GROUP BY clause is used to group rows together based on one or more columns. It creates distinct groups of data based on the specified columns.
HAVING: The HAVING clause is used to filter the grouped data after the grouping and aggregation have taken place. It applies conditions to the result of aggregate functions.
Usage:
GROUP BY: The GROUP BY clause is used in conjunction with aggregate functions to perform calculations and summarizations on the grouped data.
HAVING: The HAVING clause is used to filter the grouped data based on conditions involving aggregate functions. It narrows down the result set to only those groups that satisfy the specified conditions.
Placement:
GROUP BY: The GROUP BY clause is placed after the FROM and WHERE clauses but before the ORDER BY clause (if used) in a SQL query.
HAVING: The HAVING clause is placed after the GROUP BY clause in a SQL query. It follows the GROUP BY clause and precedes the ORDER BY clause (if used).
Evaluation:
GROUP BY: The GROUP BY clause operates on the original rows of data and creates groups based on the specified columns. It does not filter or remove any rows from the original dataset.
HAVING: The HAVING clause operates on the grouped data after the GROUP BY and aggregation steps. It applies conditions to the aggregated results and filters out groups that do not satisfy the specified conditions.

Tricky Interview Questions with Answers:

Q: What is the difference between the WHERE clause and the HAVING clause?
A: The WHERE clause is used to filter rows before grouping and aggregation, while the HAVING clause is used to filter the grouped data after grouping and aggregation. **Q: Can you use the HAVING clause without the GROUP BY clause?** A: No, the HAVING clause is always used in conjunction with the GROUP BY clause. It operates on the grouped data resulting from the GROUP BY clause. **Q: What happens if you include a column in the SELECT statement that is not in the GROUP BY clause?** A: In most SQL implementations, including a column in the SELECT statement that is not in the GROUP BY clause will result in an error. However, some databases allow it with specific configuration settings. **Q: Can you have multiple HAVING clauses in a single SQL query?** A: No, a SQL query can have only one HAVING clause. However, you can use multiple conditions within the HAVING clause using logical operators such as AND and OR. **Q: How is the ORDER BY clause different from the HAVING clause?** A: The ORDER BY clause is used to sort the final result set based on specified columns, while the HAVING clause is used to filter the grouped data based on conditions involving aggregate functions. **Q: Is it possible to use aggregate functions in the HAVING clause?** A: Yes, the HAVING clause is specifically designed to work with aggregate functions. It allows you to apply conditions to the results of aggregate functions. **Q: Can you use the GROUP BY clause without the HAVING clause?** A: Yes, the GROUP BY clause can be used independently to group rows of data without applying any conditions or filters.

Q: What is the order of execution for the GROUP BY and HAVING clauses in a SQL query?
A: The GROUP BY clause is executed first to create the groups, followed by the HAVING clause, which filters the groups based on the specified conditions.
Q: What happens if you interchange the positions of the GROUP BY and HAVING clauses in a SQL query?
A: Swapping the positions of the GROUP BY and HAVING clauses will result in a syntax error. The GROUP BY clause should always precede the HAVING clause.
Q: Can you use non-aggregated columns in the HAVING clause?
A: No, non-aggregated columns should be used in the GROUP BY clause. The HAVING clause operates on aggregated results and conditions should involve aggregate functions.

Subquery

A subquery, also known as a nested or inner query, is a query nested within another SQL query.
It's used to retrieve data that will be used in the main query to filter, sort, or perform calculations.
Subqueries can return a single value, a single column, or a result set (a list of rows and columns).

In-Line Subquery (Scalar Subquery):

Rules:
An in-line subquery is enclosed in parentheses and placed within the WHERE or HAVING clause of the main query.
It must return a single value.
When to Use:
In-line subqueries are used when you need to compare each row in the main query with a specific value or condition.
They are handy when filtering or joining based on a single value from the subquery.
Why Use:
In-line subqueries allow you to retrieve and compare individual values for each row.
They help you filter or manipulate data dynamically.

Syntax:

```
SELECT column1
FROM table1
WHERE column2 = (SELECT column3 FROM table2 WHERE condition);
```

Limitations:
Performance issues may arise when dealing with large datasets.
In-line subqueries should return results for every row in the outer query or handle NULL values gracefully.

Ordinary Subquery (Single-Row Subquery): Rules: An ordinary subquery is a standalone query embedded within another query. It can return multiple rows but must be used with operators like IN, =, <, etc. **When to Use:** Use ordinary subqueries when you need to compare data across different tables or retrieve data for filtering, sorting, or calculations. **Why Use:** They help to extract data from one query to be used in another, providing flexibility and clarity.

Syntax:

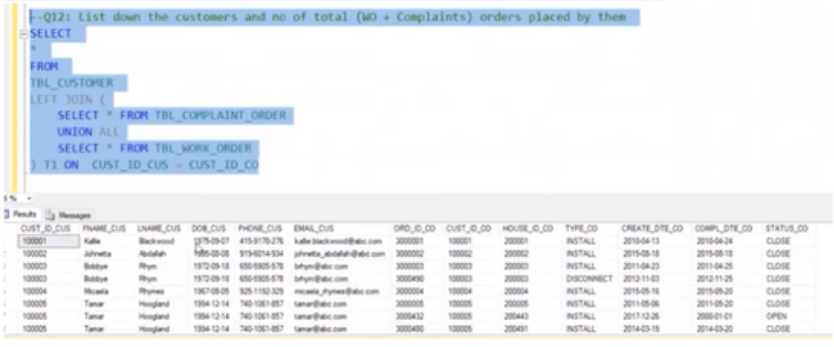
```
SELECT column1
FROM table1
WHERE column2 IN (SELECT column3 FROM table2 WHERE condition);
```

Limitations:
If not optimized, ordinary subqueries can impact performance when dealing with large datasets.
3. Combined Subquery (Correlated Subquery):
Rules:
A correlated subquery refers to a column from the outer query.
It's used when data from both the main and subqueries are needed.
When to Use:
Combine subqueries are used when you need to access data from both the main query and the subquery.
Why Use:
They enable you to perform calculations or filter data based on the relationship between the main query and the subquery.

Syntax:

```
SELECT column1
FROM table1 t1
WHERE column2 > (SELECT AVG(column3) FROM table2 t2 WHERE t2.related_column
```

Limitations:
Correlated subqueries can be performance-intensive if not optimized.



```sql
--Q12: List down the customers and no of total (WO + Complaints) orders placed by them
SELECT
FNAME_CUS, LNAME_CUS, DOB_CUS, PHONE_CUS, EMAIL_CUS, COUNT(ORD_ID_CO) AS CNT_OF_ORDERS
FROM
TBL_CUSTOMER
LEFT JOIN (
    SELECT * FROM TBL_COMPLAINT_ORDER
    UNION ALL
    SELECT * FROM TBL_WORK_ORDER
) T1 ON  CUST_ID_CUS = CUST_ID_CO
GROUP BY
FNAME_CUS, LNAME_CUS, DOB_CUS, PHONE_CUS, EMAIL_CUS
```

Results | Messages

FNAME_CUS	LNAME_CUS	DOB_CUS	PHONE_CUS	EMAIL_CUS	CNT_OF_ORDERS
Abel	Maclead	1979-11-19	631-335-3414	abel.maclead@abc.com	0
Adelina	Nabours	1981-08-09	216-4600-409	adelina_nabours@abc.com	1
Adell	Lipkin	1989-11-22	973-7746-156	adell.lipkin@abc.com	1
Ahmed	Angalich	1989-12-01	717-4345-899	ahmed.angalich@abc.com	1
Arlette	Honeywell	1970-10-09	904-7469-448	ahoneywell@abc.com	3
Aja	Gehrett	1969-04-01	973-4065-267	aja_gehrett@abc.com	2
Aline	Jeanty	1963-06-03	574-5686-280	ajeanty@abc.com	1
Alaine	Bergesen	1986-02-13	914-5398-919	alaine_bergesen@abc.net	1

```sql
/* Ordinary Sub query: Get the below data using the tbl_order */

select * from TBL_ORDER
-- Q: Get the category wise sales

-- Q: Get the %age of sales for each category

-- Q: Get all the records with sales more than average sales amount

-- Q: Get the products with the total sales more than overall average sales
```

Union and Union all:

To concatenate the result-set of two or more SELECT statements, use the UNION operator.

Within UNION, all SELECT statements must have an identical number of columns.
Similar data types must also be present in the columns.
Every SELECT statement's columns must be in the same order.

Syntax:

```sql
SELECT column_name FROM table1
UNION
SELECT column_name FROM table2;
```

UNION ALL Syntax

By default, the UNION operator only chooses distinct values. To allow duplicate values, use UNION ALL:

```sql
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

Union and Union all Key Differences:

Duplicate Rows:

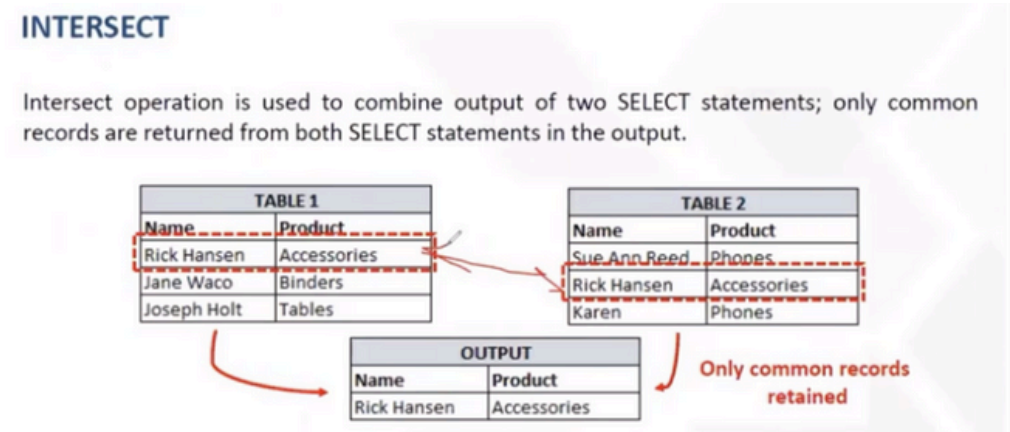
UNION: The UNION operator removes duplicate rows from the final result set. It compares all columns in the result sets and eliminates duplicates.

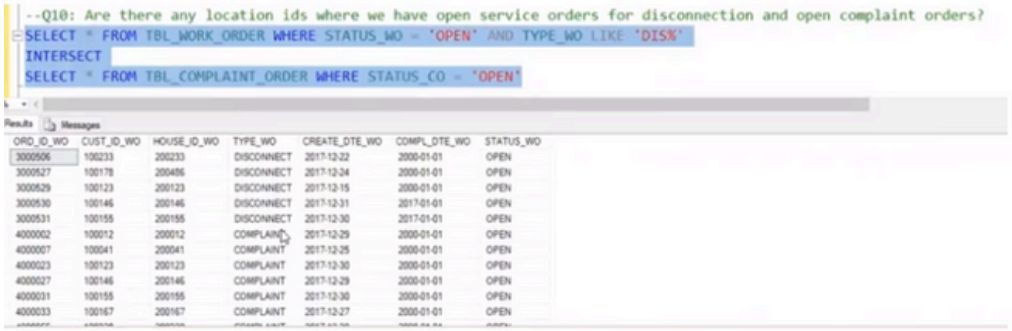
UNION ALL: The UNION ALL operator does not remove duplicate rows. It includes all rows from each SELECT statement, even if there are duplicates.
Perfor mance:
UNION: The UNION operator implicitly performs a sort operation to remove duplicates. This additional sorting process can impact performance, especially when dealing with large result sets.
UNION ALL: The UNION ALL operator does not perform any sorting or elimination of duplicates, making it generally faster than UNION.
Result Set Size:
UNION: The size of the result set produced by the UNION operator may be smaller than the sum of the individual result sets due to the removal of duplicate rows.
UNION ALL: The size of the result set produced by the UNION ALL operator is equal to the sum of the individual result sets, including duplicate rows.
INTERSECT:
Use: The INTERSECT operator returns rows that appear in both result sets of two or more SELECT statements.
Limitations:
Same column and data type requirements as UNION.
Generally not supported in all database systems.
When to Use: Use INTERSECT when you want to find common rows between two or more result sets.

Syntax:

```
SELECT column1, column2 FROM table1
INTERSECT
SELECT column1, column2 FROM table2;
```

Why Use: It helps you identify common data between different sets.





EXCEPT (MINUS in some database systems):

Use: The EXCEPT operator returns rows that are present in the first result set but not in the second result set.

Limitations:

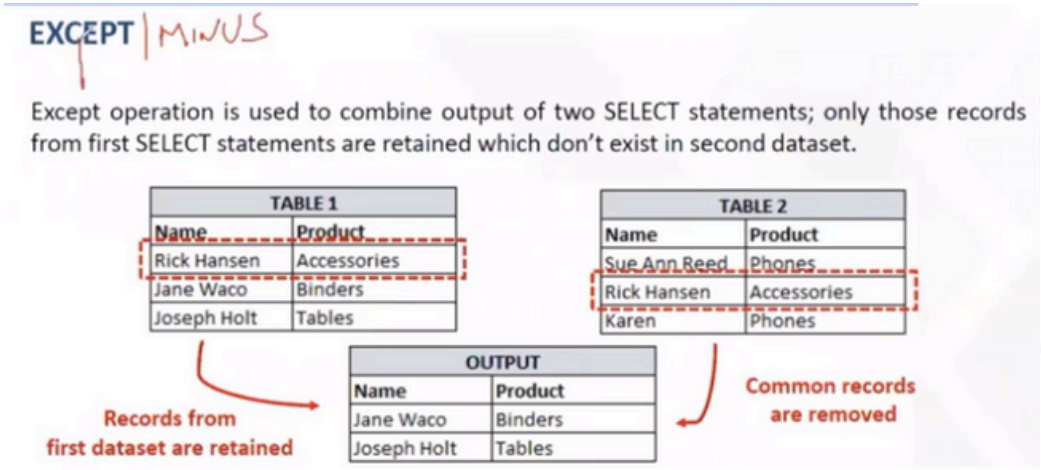
Same column and data type requirements as UNION.
Not supported in all database systems.

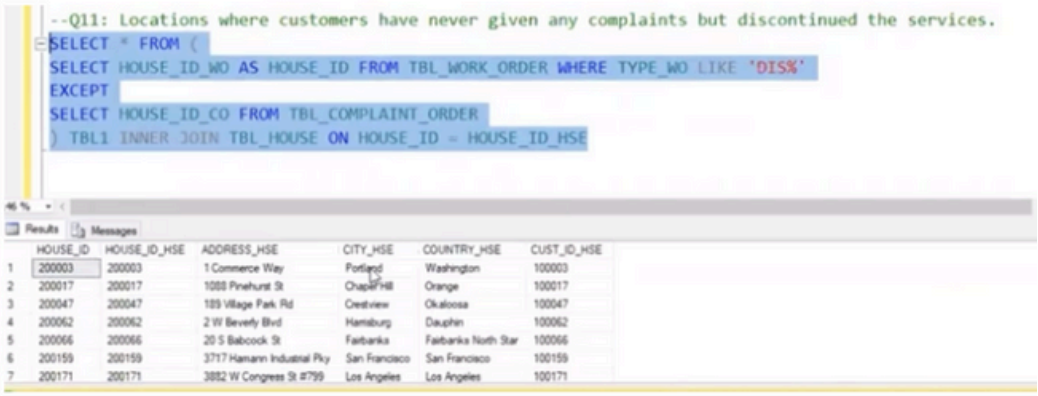
When to Use: Use EXCEPT when you want to find rows that are unique to the first result set.

Syntax:

```
SELECT column1, column2 FROM table1
EXCEPT
SELECT column1, column2 FROM table2;
```

Why Use: It helps you find differences between two result sets.





SQL ANY and ALL Operators:

You can compare a single column value to a variety of other values using the ANY and ALL operators.

The ANY Operator in SQL

Using the ANY operator provides a Boolean value in the form of TRUE if ANY of the values returned by the subquery satisfy the condition ANY denotes that the condition will hold true if the operation holds for ANY of the values in the range. It can be used with comparison operators such as =, >, <, >=, <=, <> (not equal).

Syntax:

```
SELECT column_name
FROM table_name
WHERE column_name operator ANY
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

Operator ALL:

provides a boolean value that, when combined with SELECT, WHERE, and HAVING statements, returns TRUE if ALL of the subquery values satisfy the criteria.
ALL denotes that the operation must be true for each and every value in the range for the condition to be true.

Syntax:

```
SELECT ALL column_name
FROM table_name
WHERE condition;
```

Differences between ANY and ALL:

Comparison Logic:

ANY: The ANY operator returns true if the comparison is true for at least one value in the set.
ALL: The ALL operator returns true if the comparison is true for all values in the set.
Usage in Subqueries:

ANY: The ANY operator compares the single value with each value in the set individually.
ALL: The ALL operator compares the single value with all values in the set collectively.

MySQL Functions:

There are several built-in functions in MySQL.
This reference includes sophisticated MySQL functions as well as string, numeric, date, and other data types.

String Functions:

String data types can be manipulated and operated on using string functions in SQL. They offer a range of options for extracting, changing, and editing string values. Here is a thorough description of several typical SQL string functions:

1. CONCAT(str1, str2, ...): combines one or more strings into one.

Example:

```
SELECT CONCAT('Hello', ' ', 'World')
```

Output: Returns 'Hello World'.

2. LENGTH(str): returns the length of a string, or the number of characters.

Example:

```
SELECT LENGTH('Hello')
```

Output: 5 is returned by

3.UPPER(str): a string is changed to uppercase.

Example:

```
SELECT UPPER("hello")
```

Output: “HELLO”

4. LOWER(str): lowercases a string of characters.

Example: The result of SELECT LOWER('WORLD'

```
SELECT LOWER('WORLD')
```

Output: is "world".

5. SUBSTRING(str, start, length): extracts a substring with a specified length and beginning at a particular location from a string.

Example:

```
SELECT SUBSTRING('Hello World', 7, 5)
```

Output: returns 'World'.

6. STRAIGHT(str, length): gets a certain amount of characters from a string's left side.

Example:

```
SELECT LEFT('Database', 4)
```

Output: 'Data'.

7. Wrong(str, length): a string's right side is emptied of a certain amount of characters.

For instance,

```
SELECT RIGHT("Table," "3")
```

Output: gives "ble."

8. TRIM([characters FROM] str]): Removes certain characters or leading and trailing spaces from a string.

Example:

```
SELECT TRIM('Hello  ')
```

Output: is "Hello".

9. REPLACE(str, replacement value, search value): replaces a substring with a new substring wherever it appears in the string.

Example:

```
SELECT REPLACE('Hello, World!', 'World', 'SQL')
```

Output: Hello, SQL!

10.STRING(str, search str): returns the location of a substring's first occurrence within a string.

Example:

```
SELECT INSTR('Hello World', 'World')
```

Output: There are 7 results from the query.

11. REVERSE(str): Reverses the order of characters in a string.
12. LEFT(): Returns sub string from the left of given size or characters.

Example:

```
SELECT LEFT('MYSQL IS', 5);
```

Output: MYSQL

13. RIGHT(): Returns a sub string from the right end of the given size.

Example:

```
SELECT RIGHT('MYSQL.COM', 4)
```

Output: .com

Date Functions:

In SQL, date functions are used to perform various operations on date and time data types. They provide capabilities for manipulating, extracting, and formatting date and time values.

1.CURRENT_DATE():
Returns the current date.

Example:

```
SELECT CURRENT_DATE()
```

Output: gives the current date.

2. CURRENT_TIME():
Gives the current time.

```
SELECT CURRENT_TIME()
```

Output: the current time.

3.CURRENT_TIMESTAMP(): Gives both the current date and time.
Example:

```
SELECT CURRENT_TIMESTAMP()
```

Output: current date and time.

4. EXTRACT(part FROM date): Extracts a part of (year, month, day, hour, minute, second, etc.) from a date.
Example:

```
SELECT EXTRACT(YEAR FROM '2023-07-14')
```

Output: returns 2023.

6. DATEADD(datepart, interval, date): Adds a specific interval of time to a date.

Example:

```
SELECT DATEADD(MONTH, 3, '2022-01-01')
```

Output: returns '2022-04-01'.

6. DATEDIFF(datepart, start_date, end_date): Calculates the difference between two dates as per mentioned date part (year, month, day, etc.).

Example:

```
SELECT DATEDIFF(DAY, '2022-01-01', '2022-01-31')
```

Output: returns 30.

7. DATEPART(datepart, date): Gives the mentioned part (year, month, day, hour, minute, second, etc.) of a date.

Example:

```
SELECT DATEPART(HOUR, '2022-01-01 09:30:45')
```

Output: returns 9.

8. DATE_FORMAT(date, format): Change a date according to the specified format.

Example:

```
SELECT DATE_FORMAT('2022-01-01', '%Y-%m-%d')
```

Output: returns '2022-01-01'.

9. TO_CHAR(date, format): Converts a date to a string using the specified format.

Example:

```
SELECT TO_CHAR('2022-01-01', 'YYYY-MM-DD')
```

Output: returns '2022-01-01'.

10.DATE_TRUNC(part, date): Truncates a date to the specified part (year, month, day, hour, minute, second, etc.).

Example:

```
SELECT DATE_TRUNC('MONTH', '2022-01-15')
```

Output: returns '2022-01-01'.

11. ADD_MONTHS(date, months): Adds the specified number of months to a date.

Example:

```
SELECT ADD_MONTHS('2022-01-01', 3)
```

Output: Returns '2022-04-01'.

12. LAST_DAY(date): Gives the last day of the month for a given date.

Example:

```
SELECT LAST_DAY('2022-02-15')
```

Output: returns '2022-02-28'.

13. DAYNAME(date): Gives the name of the day for a given date.

Example:

```
SELECT DAYNAME('2022-01-01')
```

Output: returns 'Saturday'.

14. MONTHNAME(date): Gives the name of the month for a given date.

Example:

```
SELECT MONTHNAME('2022-01-01')
```

Output: Returns 'January'.

15. NOW(): Gives the current date and time.

Example: SELECT NOW() Gives the current date and time.

Window Function- Advance:

SQL window functions are analytical operations on a group of rows known as a "window" or "frame" within the result set. They enable the execution of calculations and aggregations on a per-row basis while taking the window's context into account. A thorough explanation of numerous window functions is provided below, along with a sample code: but lets first understand what is difference between aggregation function and window function.
In SQL, sets of rows in a table can be calculated using both aggregate functions and window functions. The methods they use to organise and interpret the data vary, though.

Aggregate functions:

Calculation scope: Aggregate functions operate on a set of rows and return a single value for the entire set.
Grouping: They are typically used with the GROUP BY clause to partition the data into groups and calculate a result for each group.
Result granularity: Aggregate functions collapse multiple rows into a single result For example, SUM, AVG, COUNT, MAX, and MIN are aggregate functions.
Usage: They are used to obtain summary statistics or perform calculations such as calculating the total sales per category, average salary per department, or counting the number of orders per customer.

Syntax:

```
SELECT department, AVG(salary) AS average_salary
FROM employees
GROUP BY department;
```

Window functions:

- Calculation scope: Window functions perform calculations on a "window" or a subset of rows within a result set.
- Ordering: They are typically used with the ORDER BY clause to define the window's ordering within the result set.
- Result granularity: Window functions retain the individual rows in the result set but can calculate values based on the window's defined scope.
- Usage: They are used to compute values that are related to the current row but involve other rows in the result set. Examples of window functions include RANK, ROW_NUMBER, LAG, LEAD, and SUM with the OVER clause.

HOW Window function works:

The OVER clause, which specifies the window's borders and ordering, is frequently combined with window functions in programming.

SYNTAX:

```
function_name([arguments]) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY order_expression [ASC|DESC]]
    [ROWS/RANGE frame_clause]
)
```

function_name: Window function name you can use, such as ROW_NUMBER, SUM, AVG, etc.
arguments: Its optional arguments that the window function accept. For example, SUM(column_name) would calculate the sum of the specified column.
PARTITION BY: Optional clause that divides the rows into partitions or groups based on one or more expressions. The window function is applied separately to each partition.
ORDER BY: Optional clause that specifies the order in which the rows within each partition should be processed by the window function. It can use one or more columns and can be sorted in ascending (ASC) or descending (DESC) order.
ROWS/RANGE frame_clause: Optional clause that defines the window frame or the subset of rows within each partition over which the window function operates.

1. ROW_NUMBER(): Each row in a window is given a distinct sequential number using the ROW NUMBER() function.
Example:

```
SELECT ROW_NUMBER()OVER(ORDER BY PRICE DESC) as row_no, Price
FROM aemf2
```

row_no	Price
Filter...	Filter...
1	18545.45028
2	16445.61469
3	13664.30592
4	13656.35883
5	12942.99138
6	8130.668104
7	7782.907225
8	6943.70098

2. RANK():

Gives each row in a window a rank, leaving gaps for tied values.

Example:

```
SELECT RANK()OVER(ORDER BY PRICE DESC) as row_no, Price
FROM aemf2;
```

3. DENSE_RANK():

Gives each row in a window a rank, leaving no gaps for tied values.

Example:

```
SELECT DENSE_RANK()OVER(ORDER BY PRICE DESC) as row_no, Price
FROM ;
```

4. NTILE():

Divides a window's rows into a predetermined number of groups or "tiles."

Example:

```
SELECT NTILE()OVER(ORDER BY PRICE DESC) as row_no, Price
FROM ;
```

5. LAG():

Accesses a previous row's value within a window.

Example:

```
SELECT PRICE,LAG(PRICE)OVER(ORDER BY DAY DESC) as row_no, Price
FROM aemf2 ;
```

Output:

PRICE	row_no	Price
171.3297338	NULL	171.3297338
166.8887175	171.3297338	166.8887175
519.3651684	166.8887175	519.3651684
362.9946474	519.3651684	362.9946474
304.7939602	362.9946474	304.7939602
240.0486174	304.7939602	240.0486174
339.38713979999994	240.0486174	339.38713979999994
360.6572704	339.38713979999994	360.6572704
150.7608162	360.6572704	150.7608162
139.0739312	150.7608162	139.0739312

6. LEAD():

Accesses the value of a subsequent row within a window.

Example:

```
SELECT PRICE,LEAD(PRICE)OVER(ORDER BY DAY DESC) as row_no, Price
FROM aemf2 ;
```

Output:

PRICE	row_no	Price
171.3297338	166.8887175	171.3297338
166.8887175	519.3651684	166.8887175
519.3651684	362.9946474	519.3651684
362.9946474	304.7939602	362.9946474
304.7939602	240.0486174	304.7939602
240.0486174	339.38713979999994	240.0486174
339.38713979999994	360.6572704	339.38713979999994
360.6572704	150.7608162	360.6572704

7. FIRST_VALUE():

Accesses a previous row's value within a window.

Example:

```
SELECT FIRST_VALUE(PRICE)OVER(ORDER BY DAY DESC) as FIRST_V, Price
FROM aemf2 ;
```

Output:

FIRST_V	Price
171.3297338	171.3297338
171.3297338	166.8887175
171.3297338	519.3651684
171.3297338	362.9946474
171.3297338	304.7939602
171.3297338	240.0486174
171.3297338	339.38713979999994

8. PERCENT_RANK():

Accesses a previous row's value within a window.

Example:

```
SELECT Sales, PERCENT_RANK() OVER (ORDER BY Sales) AS PercentileRank
FROM SalesData;
```

CTE(Common Table Expression) IN SQL:

Known in SQL as a Common Table Expression (CTE), a temporary named result set may be referred to within a query. Having the ability to build a subquery and utilize it more tha once within a single query might help make complex queries easier to comprehend and maintain. A detailed explanation of CTE is provided below, along with an example:
A CTE's two constituent parts, the anchor member and the recursive member, are defined by the WITH clause (optional for recursive queries). The anchor member of the CTE serves as its foundation, while the recursive member builds upon it through iterative processing.

Benefits of using CTE:

Improved Readability: CTEs make complex queries easier to understand by breaking them into logical sections. This improves code readability and maintainability.
Code Reusability: CTEs can be referenced multiple times within the same query, eliminating the need to repeat the subquery logic. This improves code efficiency and reduces the chances of errors.
Recursive Queries: CTEs can be used for recursive queries where a query refers to its own output. This enables hierarchical and iterative processing, such as retrieving hierarchical data or calculating running totals.

Simple Example:

Now, let's Extract the names of employees along with their corresponding department names using a CTE.

```
WITH EmployeeDepartments AS (
  SELECT E.EmployeeID, E.FirstName, E.LastName, D.DepartmentName
  FROM Employees3 AS E
  JOIN Departments AS D ON E.DepartmentID = D.DepartmentID
)
SELECT EmployeeID, FirstName, LastName, DepartmentName
FROM EmployeeDepartments;
```

Output:

EmployeeID	FirstName	LastName	DepartmentName
Filter...	Filter...	Filter...	Filter...
1	John	Doe	Sales
2	Jane	Smith	Sales
3	Michael	Johnson	Marketing

Explaination:

In this example, we define a CTE as "EmployeeDepartments" using the WITH clause. The CTE selects the employee details (EmployeeID, FirstName, LastName) and their corresponding department names (DepartmentName) by joining the "Employees" and "Departments" tables.
Finally, we use the CTE within the main query to directly select the desired columns from the CTE, resulting in a clear and concise syntax.

Recursive Common Table Expression (CTE):

An SQL technique called a recursive Common Table Expression (CTE) enables you to query hierarchical data or data with recursive relationships. It aids in the resolution of issues where each row depends on the outcomes of earlier rows returned by the same query. Here is a brief explanation in everyday language:
Consider a table of employees where the columns "EmployeeID" and "ManagerID" indicate which employees report to which managers. You may locate every employee in the reporting chain beginning with a single employee using a recursive CTE.

Example: Suppose, You have this employee data.

```
EmployeeID | ManagerID
----------------------
1          | NULL
2          | 1
3          | 1
4          | 2
5          | 2
```

You want to find all employees reporting to EmployeeID 1. Here's a simple recursive CTE to achieve that:

Input:

```
WITH RecursiveCTE AS (
  SELECT EmployeeID, ManagerID
  FROM Employees
  WHERE EmployeeID = 1 -- Starting with EmployeeID 1

  UNION ALL

  SELECT E.EmployeeID, E.ManagerID
  FROM Employees AS E
  INNER JOIN RecursiveCTE AS R
  ON E.ManagerID = R.EmployeeID
)
SELECT EmployeeID, ManagerID
FROM RecursiveCTE;
```

Output:

```
EmployeeID | FirstName | LastName | Salary | SalaryRange
-----------------------------------------------------
1          | John      | Doe      | 50000  | Low
2          | Jane      | Smith    | 60000  | Medium
3          | Michael   | Johnson  | 75000  | Medium
4          | Sarah     | Brown    | 90000  | High
```

In this example, we define a recursive CTE named "RecursiveCTE." We start with the anchor member, which selects EmployeeID 1 from the Employees table.
Then, we have the recursive member, which joins the Employees table with the Recursive CTE itself, matching the ManagerID of the current row with the EmployeeID of the previous rows. It continues to add rows until no more matches are found.
Finally, we select the EmployeeID and ManagerID from the RecursiveCTE, giving us all the employees reporting to EmployeeID 1.
Recursive CTEs are helpful when dealing with hierarchical data structures like organizational charts or family trees

Case When –Conditional :

The SQL CASE WHEN statement is a conditional expression that enables you to take various actions in response to various situations. It resembles making choices when writing SQL queries.

Rules:

The CASE statement allows you to perform conditional logic in SQL queries.
You can use it in the SELECT, WHERE, HAVING, and ORDER BY clauses.
It can return a single value (simple CASE) or multiple values (searched CASE).
CASE is followed by one or more WHEN clauses and an optional ELSE clause.

When to Use:
Use the CASE statement when you need to transform data conditionally or categorize data into different groups.
It's handy for custom calculations, mapping values, and generating derived columns.
CASE is particularly useful when you need to create calculated fields in your query results.
Why Use:
CASE adds a layer of flexibility to your SQL queries by allowing you to make decisions based on data conditions.
It can simplify your queries by reducing the need for complex IF-THEN-ELSE logic in your application code.
It's a valuable tool for data transformation and reporting.
Syntax:

```
-- Simple CASE statement
SELECT column1,
       CASE
           WHEN condition1 THEN result1
           WHEN condition2 THEN result2
           ELSE result3
       END AS new_column
FROM table;
```

Limitations:
Avoid nesting too many CASE statements within each other, as it can make your SQL hard to read and maintain.
CASE is not a suitable replacement for complex control flow that should be handled by application code.
Important Notes:
The ELSE clause is optional but recommended for handling unexpected or NULL values.
When CASE is used in the WHERE clause, it filters rows based on the conditions.
In the SELECT clause, CASE creates a new column in the result set.
In the ORDER BY clause, it allows conditional sorting of the result set.
CASE can be used with any data type, and the result must be consistent in data type across all branches.

Explanation:

Consider a database of students that includes the columns "StudentID," "FirstName," "LastName," and "Grade." You should put pupils into the following categories depending on their grades: "Excellent," "Good," "Average," and "Below Average." This is made possible via the CASE WHEN statement.
Example:
Let's say you have the following student data:

```
StudentID | FirstName | LastName | Grade
-----------------------------------------
1         | John      | Doe      | 85
2         | Jane      | Smith    | 75
3         | Michael   | Johnson  | 92
4         | Sarah     | Brown    | 65
```

You want to categorize students based on their grades. Here is a easy example using the CASE WHEN statement:

Input:

```
SELECT
  StudentID,
  FirstName,
  LastName,
  Grade,
  CASE
    WHEN Grade >= 90 THEN 'Excellent'
    WHEN Grade >= 80 AND Grade < 90 THEN 'Good'
    WHEN Grade >= 70 AND Grade < 80 THEN 'Average'
    ELSE 'Below Average'
  END AS GradeCategory
FROM
  Students;
```

In this example, the grade category for each student is determined using the CASE WHE statement within the SELECT statement. This is how it goes:
The CASE WHEN statement examines each condition individually for each row.
It labels the GradeCategory column as "Excellent" if the grade is more than or equal to 90.
The GradeCategory column is given the designation "Good" if the grade falls within the range of 80 and 89 (inclusive).
The GradeCategory column is given the label "Average" if the grade falls within the range of 70 and 79 (inclusive).
It labels the GradeCategory column as "Below Average" for any other grade values.
The outcome will include the original columns (StudentID, FirstName, LastName, Grade), as well as GradeCategory, another column.

Output:

```
StudentID | FirstName | LastName | Grade | GradeCategory
-----------------------------------------------------
1         | John      | Doe      | 85    | Good
2         | Jane      | Smith    | 75    | Average
3         | Michael   | Johnson  | 92    | Excellent
4         | Sarah     | Brown    | 65    | Below Average
```

When doing conditional evaluations and taking different actions in response to particular circumstances, the CASE WHEN statement is helpful. It aids in classifying or transforming data according to logical criteria in your SQL queries.

Pivoting and Unpivoting Techniques:

Pivoting: Using the pivoting and unpivoting techniques, SQL can change the format of data from row-wise to column-wise (pivoting) or from column-wise to row-wise (unpivoting) (unpivoting). Here is a thorough explanation of both ideas with straightforward examples:
Turning rows into columns is known as pivoting. When you wish to rotate or transform data to generate a summary view, it can be helpful. For this, the PIVOT operator is frequently employed. Here's an Example:

Turning rows into columns is known as pivoting. When you wish to rotate or transform data to generate a summary view, it can be helpful. For this, the PIVOT operator is frequently employed. Here's an Example:
Suppose you have a table named "Sales" with columns "Product," "Region," and "Revenue." The table captures the revenue generated by each product in different regions.

```
  Product | Region | Revenue
-------------------------
A       | East   | 1000
A       | West   | 1500
B       | East   | 2000
B       | West   | 1200
```

You can use the PIVOT operator to pivot the data and display the revenue for each product in separate columns based on the region:

```
SELECT Product,
    [East] AS Revenue_East,
    [West] AS Revenue_West
FROM Sales PIVOT (
        SUM(Revenue) FOR Region IN ([East], [West])
    ) AS PivotTable;
```

The PIVOT operator is used to convert the rows into columns in the aforementioned example. The revenue is determined for each mix of product and area using the SUM(Revenue) aggregation function. This will have the following effects:
Output:

```
    Product | Revenue_East | Revenue_West
-----------------------------------
A       | 1000         | 1500
B       | 2000         | 1200
```

Unpivoting:

The process of unpivoting involves changing columns into rows. When normalising or transforming data to do analysis or other activities, it is helpful. Usually, the UNPIVOT operator is employed for this. Here's an illustration.
Lets take last example: you have a table named "Sales" with columns "Product," "Revenue_East," and "Revenue_West." Each column represents the revenue for a specific product in the respective region.

```
    Product | Revenue_East | Revenue_West
-----------------------------------
A       | 1000         | 1500
B       | 2000         | 1200
```

To unpivot the data and display the revenue for each product in different regions in separate rows, you can use the UNPIVOT operator:

```
SELECT Product,
    Region,
    Revenue
FROM Sales UNPIVOT (
        Revenue FOR Region IN (Revenue_East, Revenue_West)
    ) AS UnpivotTable;
```

The UNPIVOT operator is used in the example above to convert the columns into rows. Each time a product and area are combined, a new row is created with the matching revenue. This will have the following effects:

Output:

```
Product | Region | Revenue
------------------------
A       | East   | 1000
A       | West   | 1500
B       | East   | 2000
B       | West   | 1200
```

What is View in SQL:

In SQL, views are a subset of virtual tables. The rows and columns of a view are identical to those in a database's actual table. By choosing fields from one or more database tables, we can build a view. A view may include all table rows or only certain rows according to a set of criteria. This post will teach us how to add, remove, and update Views.
Example: Create View

Syntax:

```
CREATE VIEW employee_details AS
SELECT employee_id, employee_name, department
FROM employees
WHERE department = 'IT';
```

view named employee_details is created that displays the employee ID, name, and department for employees in the IT department.

See the data in View:

```
SELECT * FROM employee_details
```

Create VIEW fro multiple tables:

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

If you want to see all view tables:
Syntax:

```
use "database_name";
show full tables where table_type like "%VIEW"
```

Stored Procedures:

SQL-prepared code that has been saved as a stored procedure can be used repeatedly. So, if you frequently develop SQL queries, save them as stored procedures and call them to execute them. Additionally, you may supply parameters to stored procedures so that they can take action based on the values of the parameters you pass.
Pre-compiled SQL statements are saved in the database as stored procedures.
They enable you to combine several SQL statements into a single, repeatable unit.
Stored procedures have the ability to process complex operations, receive input parameters, and return values or result sets.

Lets Create Stored Procedure:

Syntax:

```
CREATE PROCEDURE procedure_name
(parameter1 data_type, parameter2 data_type, ...)
AS
BEGIN
   – SQL statements to be executed
END
```

Explanation of the Parameter

The parameters are the most crucial component. Values are passed to the Procedure using parameters. There are various types of parameters, including the following:
BEGIN: This is what actually executes, or we could say it is a section that can be executed.
END: The code will run up until this point.

Execute the procedure

EXEC procedure_name parameter1_value, parameter2_value,

Triggers in SQL:

When a particular event takes place on a table, a SQL trigger is a database object linked to that table that automatically executes a group of SQL queries. Within a database, triggers are used to automate some tasks, ensure data integrity, and enforce business rules. They can be triggered by a variety of actions, such as inserting, updating, or deleting data from a table, and they let you carry out extra actions in response to those actions.

How are SQL triggers implemented?

SQL triggers are associated with a particular table and defined using SQL statements. The related trigger code is automatically executed when the indicated trigger event (such as an INSERT, UPDATE, or DELETE) takes place on that table. The trigger code may include SQL statements that enforce restrictions, modify data in the same or different tables, or do other actions. Triggers can be set to run before or after the triggering event and are defined to run inside the transaction scope.

@PREMMANDAL

BUSINESS & DATA ANALYST

Was this helpful

WOULD YOU MIND SHOWING YOUR SUPPORT BY GIVING IT A LIKE?

- Prem Mandal