MYSQL

Tutorial

Basics to Advanced

MUJAHID RAZA

mujahid.raza_07

www.linkedin.com/in/mujahidraza

Contents

INTRODUCTION

What is Data?

Data is a collection of a distinct small unit of information. It can be used in a variety of forms like text, numbers, media, bytes, etc. it can be stored in pieces of paper or electronic memory, etc.

Word 'Data' is originated from the word 'datum' that means 'single piece of information.' It is plural of the word datum.

In computing, Data is information that can be translated into a form for efficient movement and processing. Data is interchangeable.

What is Database?

A **database** is an organized collection of data, so that it can be easily accessed and managed.

You can organize data into tables, rows, columns, and index it to make it easier to find relevant information.

The **main purpose** of the database is to operate a large amount of information by storing, retrieving, and managing data.

There are many **databases available** like MySQL, Sybase, Oracle, MongoDB, Informix, PostgreSQL, SQL Server, etc.

Evolution of Databases

The database has completed more than 50 years of journey of its evolution from flat-file system to relational and objects relational systems. It has gone through several generations.

- 1968 was the year when **File-Based** database were introduced. In file-based databases, data was maintained in a flat file.
- 1968-1980 was the era of the **Hierarchical Database**. Prominent hierarchical database model was **IBM's** first DBMS. It was called **IMS** (Information Management System).
- **Charles Bachman** developed the first DBMS at Honeywell called Integrated Data Store **(IDS).** It was developed in the early **1960s,** but it was standardized in 1971 by the CODASYL group (Conference on Data Systems Languages).

- **1970 - Present:** It is the era of **Relational Database and Database Management**. In 1970, the relational model was proposed by E.F. Codd.

Database Management System

- Database management system is a software which is used to manage the database. For example: MySQL, Oracle, etc are a very popular commercial database which is used in different applications.
- DBMS provides an interface to perform various operations like database creation, storing data in it, updating data, creating a table in the database and a lot more.
- It provides protection and security to the database. In the case of multiple users, it also maintains data consistency.

DBMS allows users the following tasks:

- **Data Definition:** It is used for creation, modification, and removal of definition that defines the organization of data in the database.
- **Data Updation:** It is used for the insertion, modification, and deletion of the actual data in the database.
- **Data Retrieval:** It is used to retrieve the data from the database which can be used by applications for various purposes.
- **User Administration:** It is used for registering and monitoring users, maintain data integrity, enforcing data security, dealing with concurrency control, monitoring performance and recovering information corrupted by unexpected failure.

Characteristics of DBMS

- It uses a digital repository established on a server to store and manage the information.
- It can provide a clear and logical view of the process that manipulates data.
- DBMS contains automatic backup and recovery procedures.
- It contains ACID properties which maintain data in a healthy state in case of failure.
- It can reduce the complex relationship between data.
- It is used to support manipulation and processing of data.
- It is used to provide security of data.
- It can view the database from different viewpoints according to the requirements of the user.

Advantages of DBMS

- **Controls database redundancy:** It can control data redundancy because it stores all the data in one single database file and that recorded data is placed in the database.
- **Data sharing:** In DBMS, the authorized users of an organization can share the data among multiple users.
- **Easily Maintenance:** It can be easily maintainable due to the centralized nature of the database system.
- **Reduce time:** It reduces development time and maintenance need.
- **Backup:** It provides backup and recovery subsystems which create automatic backup of data from hardware and software failures and restores the data if required.
- **multiple user interface:** It provides different types of user interfaces like graphical user interfaces, application program interfaces

Disadvantages of DBMS

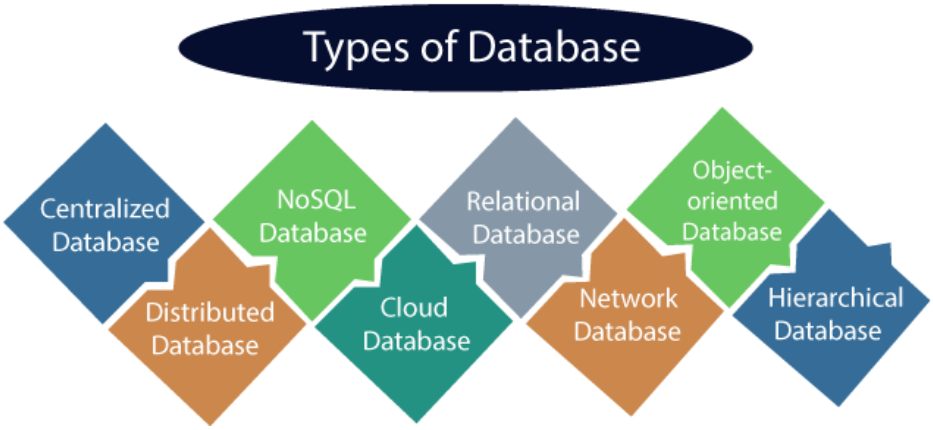
- **Cost of Hardware and Software:** It requires a high speed of data processor and large memory size to run DBMS software.
- **Size:** It occupies a large space of disks and large memory to run them efficiently.
- **Complexity:** Database system creates additional complexity and requirements.
- **Higher impact of failure:** Failure is highly impacted the database because in most of the organization, all the data stored in a single database and if the database is damaged due to electric failure or database corruption then the data may be lost forever.

No.	DBMS	RDBMS
1)	DBMS applications store **data as file**.	RDBMS applications store **data in a tabular form**.
2)	In DBMS, data is generally stored in either a hierarchical form or a navigational form.	In RDBMS, the tables have an identifier called primary key and the data values are stored in the form of tables.
3)	**Normalization is not** present in DBMS.	**Normalization is** present in RDBMS.
4)	DBMS does **not apply any security** with regards to data manipulation.	RDBMS **defines the integrity constraint** for the purpose of ACID (Atomocity, Consistency, Isolation and Durability) property.
5)	DBMS uses file system to store data, so there will be **no relation between the tables**.	in RDBMS, data values are stored in the form of tables, so a **relationship** between these data values will be stored in the form of a table as well.

6)	DBMS has to provide some uniform methods to access the stored information.	RDBMS system supports a tabular structure of the data and a relationship between them to access the stored information.
7)	DBMS **does not support distributed database**.	RDBMS **supports distributed database**.
8)	DBMS is meant to be for small organization and **deal with small data**. it supports **single user**.	RDBMS is designed to **handle large amount of data**. it supports **multiple users**.
9)	Examples of DBMS are file systems, **xml** etc.	Example of RDBMS are **mysql**, **postgre**, **sql server**, **oracle** etc.

Types of Databases

There are various types of databases used for storing different varieties of data:



1) Centralized Database

It is the type of database that stores data at a centralized database system. It comforts the users to access the stored data from different locations through several applications. These applications contain the authentication process to let users access data securely. An example of a Centralized database can be Central Library that carries a central database of each library in a college/university

Advantages of Centralized Database

- It has decreased the risk of data management, i.e., manipulation of data will not affect the core data.
- Data consistency is maintained as it manages data in a central repository.
- It provides better data quality, which enables organizations to establish data standards.
- It is less costly because fewer vendors are required to handle the data sets.

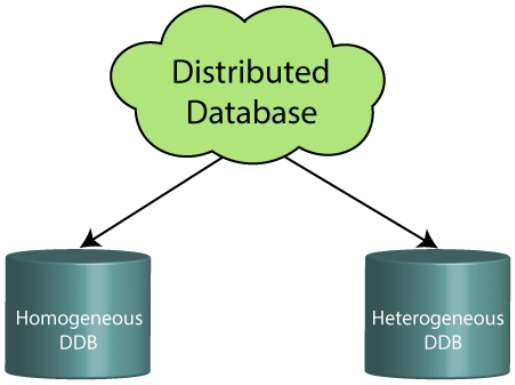
Disadvantages of Centralized Database

- The size of the centralized database is large, which increases the response time for fetching the data.
- It is not easy to update such an extensive database system.
- If any server failure occurs, entire data will be lost, which could be a huge loss.

2) Distributed Database

Unlike a centralized database system, in distributed systems, data is distributed among different database systems of an organization. These database systems are connected via communication links. Such links help the end-users to access the data easily. **Examples** of the Distributed database are Apache Cassandra, HBase, Ignite, etc.

We can further divide a distributed database system into:



- **Homogeneous DDB:** Those database systems which execute on the same operating system and use the same application process and carry the same hardware devices.
- **Heterogeneous DDB:** Those database systems which execute on different operating systems under different application procedures, and carries different hardware devices.

Advantages of Distributed Database

- Modular development is possible in a distributed database, i.e., the system can be expanded by including new computers and connecting them to the distributed system.
- One server failure will not affect the entire data set.

3) Relational Database

This database is based on the relational data model, which stores data in the form of rows(tuple) and columns(attributes), and together forms a table(relation). A relational database uses SQL for storing, manipulating, as well as maintaining the data. E.F. Codd invented the database in 1970. Each table in the database carries a key that makes the data unique from others. **Examples** of Relational databases are MySQL, Microsoft SQL Server, Oracle, etc.

What is table/Relation?

Everything in a relational database is stored in the form of relations. The RDBMS database uses tables to store data. A table is a collection of related data entries and contains rows and columns to store data. Each table represents some real-world objects such as person, place, or event about which information is collected. The organized collection of data into a relational table is known as the logical view of the database.

Properties of a Relation:

- Each relation has a unique name by which it is identified in the database.
- Relation does not contain duplicate tuples.
- The tuples of a relation have no specific order.
- All attributes in a relation are atomic, i.e., each cell of a relation contains exactly one value

What is a row or record?

A row of a table is also called a record or tuple. It contains the specific information of each entry in the table. It is a horizontal entity in the table. For example, The above table contains 5 records.

Properties of a row:

- No two tuples are identical to each other in all their entries.
- All tuples of the relation have the same format and the same number of entries.

- The order of the tuple is irrelevant. They are identified by their content, not by their position.

What is a column/attribute?

A column is a vertical entity in the table which contains all information associated with a specific field in a table. For example, "name" is a column in the above table which contains all information about a student's name.

Properties of an Attribute:

- Every attribute of a relation must have a name.
- Null values are permitted for the attributes.
- Default values can be specified for an attribute automatically inserted if no other value is specified for an attribute.
- Attributes that uniquely identify each tuple of a relation are the primary key.

What is data item/Cells?

The smallest unit of data in the table is the individual data item. It is stored at the intersection of tuples and attributes.

Properties of data items:

- Data items are atomic.
- The data items for an attribute should be drawn from the same domain.

Degree

The total number of attributes that comprise a relation is known as the degree of the table.

Domain

The domain refers to the possible values each attribute can contain. It can be specified using standard data types such as integers, floating numbers, etc. **For example**, An attribute entitled Marital_Status may be limited to married or unmarried values.

NULL Values

The NULL value of the table specifies that the field has been left blank during record creation. It is different from the value filled with zero or a field that contains space.

Data Integrity

There are the following categories of data integrity exist with each RDBMS:

Entity integrity: It specifies that there should be no duplicate rows in a table.

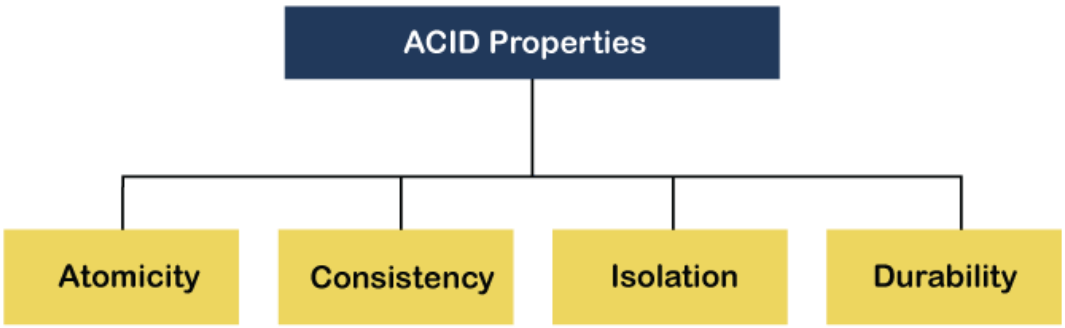
Domain integrity: It enforces valid entries for a given column by restricting the type, the format, or the range of values.

Referential integrity specifies that rows cannot be deleted, which are used by other records.

User-defined integrity: It enforces some specific business rules defined by users. These rules are different from the entity, domain, or referential integrity.

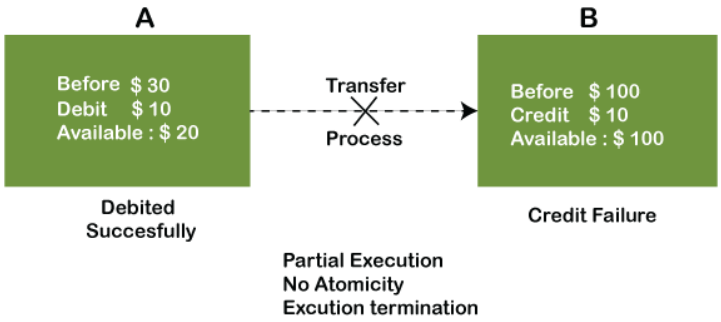
Properties of Relational Database

There are following four commonly known properties of a relational model known as **ACID** properties.



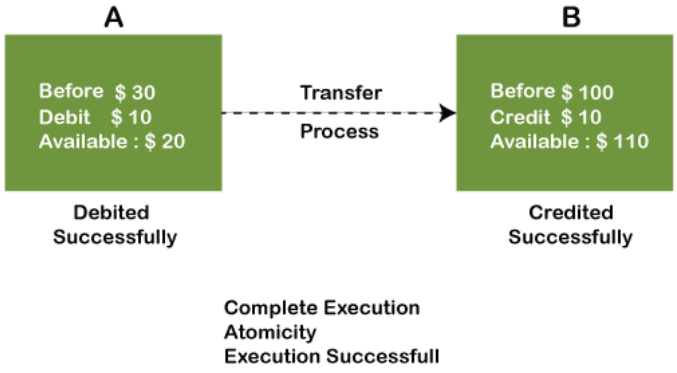
Atomicity :- This ensures the data operation will complete either with success or with failure. It follows the 'all or nothing' strategy. For example, a transaction will either be committed or will abort.

Example: If Remo has account A having $30 in his account from which he wishes to send $10 to Sheero's account, which is B. In account B, a sum of $ 100 is already present. When $10 will be transferred to account B, the sum will become $110. Now, there will be two operations that will take place. One is the amount of $10 that Remo wants to transfer will be debited from his account A, and the same amount will get credited to account B, i.e., into Sheero's account. Now, what happens - the first operation of debit executes successfully, but the credit operation, however, fails. Thus, in Remo's account A, the value becomes $20, and to that of Sheero's account, it remains $100 as it was previously present.



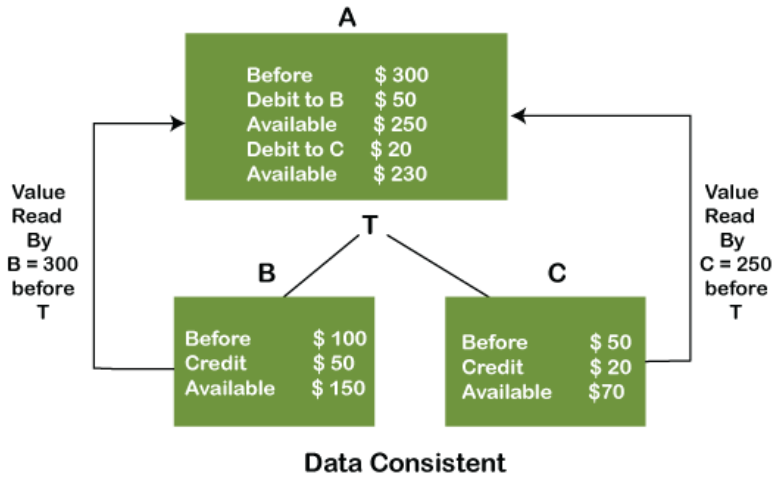
In the above diagram, it can be seen that after crediting $10, the amount is still $100 in account B. So, it is not an atomic transaction.

The below image shows that both debit and credit operations are done successfully. Thus the transaction is atomic.



Thus, when the amount loses atomicity, then in the bank systems, this becomes a huge issue, and so the atomicity is the main focus in the bank systems.

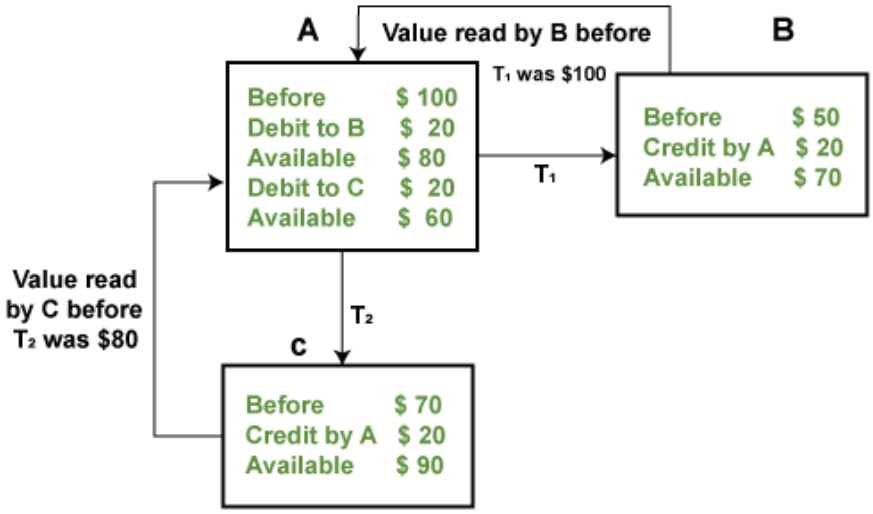
Consistency :- If we perform any operation over the data, its value before and after the operation should be preserved. For example, the account balance before and after the transaction should be correct, i.e., it should remain conserved.



Isolation :- There can be concurrent users for accessing data at the same time from the database. Thus, isolation between the data should remain isolated. For example, when multiple transactions occur at the same time, one transaction effects should not be visible to the other transactions in the database.

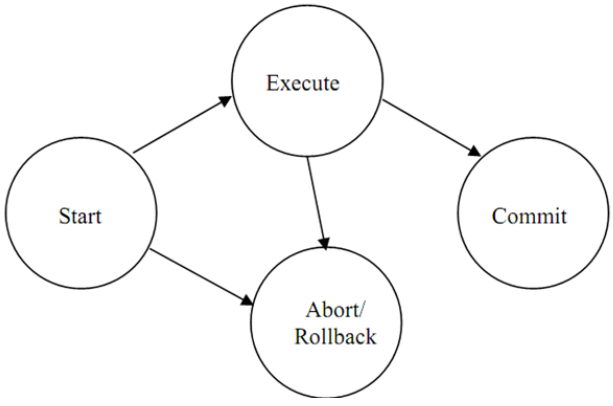
Each transaction will execute without impacting each other.

Example: If two operations are concurrently running on two different accounts, then the value of both accounts should not get affected. The value should remain persistent. As you can see in the below diagram, account A is making T1 and T2 transactions to account B and C, but both are executing independently without affecting each other. It is known as Isolation.



Isolation - Independent execution of T_1 & T_2 by A

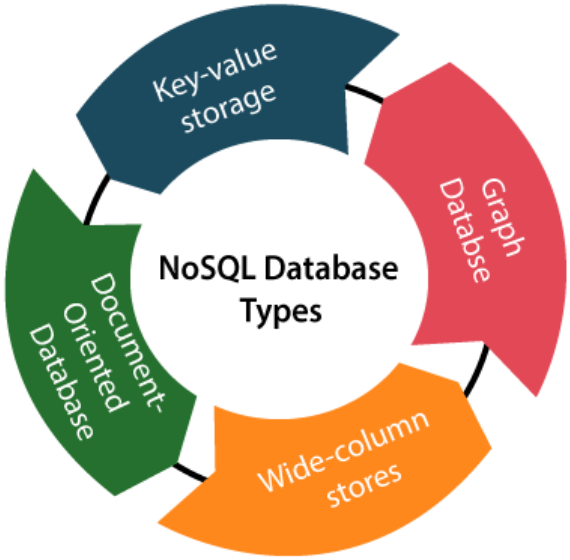
Durability :- It ensures that once it completes the operation and commits the data, data changes should remain permanent.



1.

4) NoSQL Database

Non-SQL/Not Only SQL is a type of database that is used for storing a wide range of data sets. It is not a relational database as it stores data not only in tabular form but in several different ways. It came into existence when the demand for building modern applications increased. Thus, NoSQL presented a wide variety of database technologies in response to the demands. We can further divide a NoSQL database into the following four types



a. **Key-value storage:** It is the simplest type of database storage where it stores every single item as a key (or attribute name) holding its value, together.
b. **Document-oriented Database:** A type of database used to store data as JSON-like document. It helps developers in storing data by using the same document-model format as used in the application code.
c. **Graph Databases:** It is used for storing vast amounts of data in a graph-like structure. Most commonly, social networking websites use the graph database.
d. **Wide-column stores:** It is similar to the data represented in relational databases. Here, data is stored in large columns together, instead of storing in rows.

Advantages of NoSQL Database

- It enables good productivity in the application development as it is not required to store data in a structured format.
- It is a better option for managing and handling large data sets.
- It provides high scalability.
- Users can quickly access data from the database through key-value.

5) Cloud Database

A type of database where data is stored in a virtual environment and executes over the cloud computing platform. It provides users with various cloud computing services (SaaS, PaaS, IaaS, etc.) for accessing the database. There are numerous cloud platforms, but the best options are:

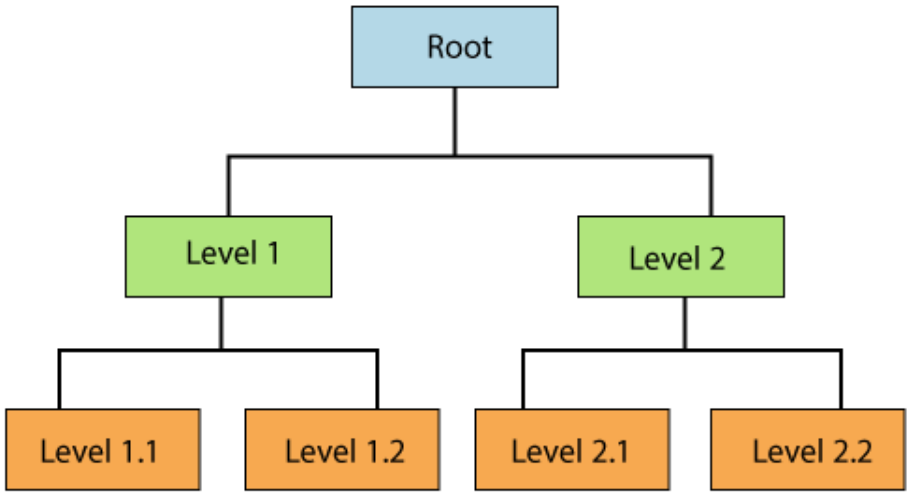
- Amazon Web Services(AWS)
- Microsoft Azure
- Kamatera
- PhonixNAP
- ScienceSoft
- Google Cloud SQL, etc.

6) Object-oriented Databases

The type of database that uses the object-based data model approach for storing data in the database system. The data is represented and stored as objects which are similar to the objects used in the object-oriented programming language.

7) Hierarchical Databases

It is the type of database that stores data in the form of parent-children relationship nodes. Here, it organizes data in a tree-like structure.



Hierarchical Database

8) Network Databases

It is the database that typically follows the network data model. Here, the representation of data is in the form of nodes connected via links between them. Unlike the hierarchical database, it allows each record to have multiple children and parent nodes to form a generalized graph structure.

9) Personal Database

Collecting and storing data on the user's system defines a Personal Database. This database is basically designed for a single user.

10) Operational Database

The type of database which creates and updates the database in real-time. It is basically designed for executing and handling the daily data operations in several businesses. For example, An organization uses operational databases for managing per day transactions.

11) Enterprise Database

Large organizations or enterprises use this database for managing a massive amount of data. It helps organizations to increase and improve their efficiency. Such a database allows simultaneous access to users.

CAP Theorem :-

The CAP theorem states that it is not possible to guarantee all three of the desirable properties – consistency, availability, and partition tolerance at the same time in a distributed system with data replication.

The theorem states that networked shared-data systems can only strongly support two of the following three properties.

Consistency

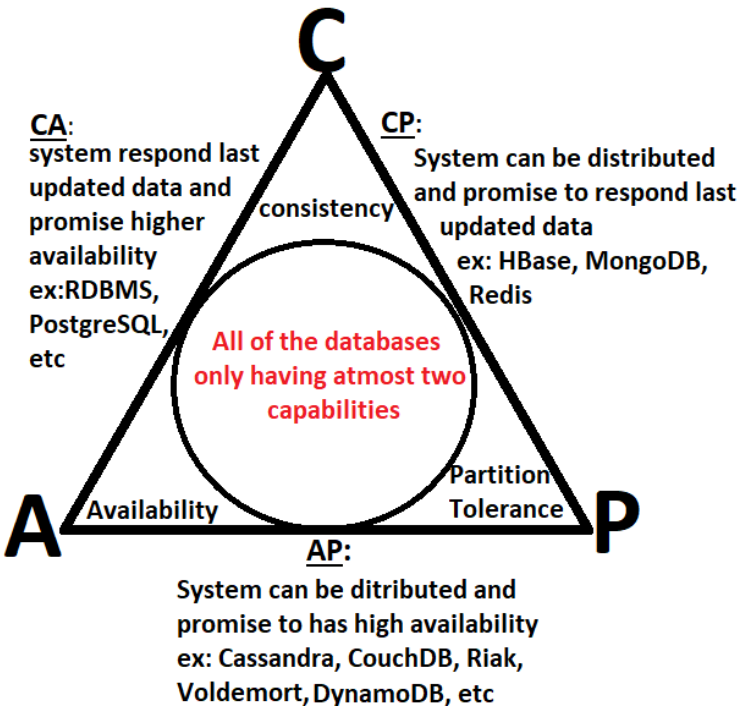
Consistency means that all clients see the same data at the same time, no matter which node they connect to. For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed ‘successful.’

Availability

Availability means that any client making a request for data gets a response, even if one or more nodes are down. Another way to state this—all working nodes in the distributed system return a valid response for any request, without exception.

Partition tolerance

A partition is a communications break within a distributed system—a lost or temporarily delayed connection between two nodes. Partition tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.



- **CA(Consistency and Availability)-**

  The system prioritizes availability over consistency and can respond with possibly stale data.

  Example databases: Cassandra, CouchDB, Riak, Voldemort.

- **AP(Availability and Partition Tolerance)-**

  The system prioritizes availability over consistency and can respond with possibly stale data.

The system can be distributed across multiple nodes and is designed to operate reliably even in the face of network partitions.

Example databases: Amazon DynamoDB, Google Cloud Spanner.

- **CP(Consistency and Partition Tolerance)-**

The system prioritizes consistency over availability and responds with the latest updated data.

The system can be distributed across multiple nodes and is designed to operate reliably even in the face of network partitions.

Example databases: Apache HBase, MongoDB, Redis.

SQL

SQL is a "standard language for accessing and manipulating databases".

Structured Query Language (SQL), which is a computer language for storing, manipulating, and retrieving data stored in relational database management systems (RDBMS). SQL was developed at IBM by **Donald Chamberlin** , **Donald C. Messerli** , and **Raymond F. Boyce** in the year 1970s.

Characteristics of SQL :

- SQL is an "ANSI" and "ISO" standard computer language for creating and manipulating database.
- It allows the user to create, update, delete and retrieve data from a database.
- It is very simple and easy to learn.
- It is used specifically for relational databases.
- It works with database programs like DB2, Oracle, MS Access, etc.

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

MySQL

MySQL is currently the most popular database management system software used for managing the relational database. It is open-source database software, which is supported by Oracle Company. It is fast, scalable, and easy to use database management system in comparison with Microsoft SQL Server and Oracle Database.

Characteristics of MySQL:

Relational Database Management System (RDBMS)

MySQL is a relational database management system. This database language is based on the SQL queries to access and manage the records of the table.

Easy to use

MySQL is easy to use. We have to get only the basic knowledge of SQL. We can build and interact with MySQL by using only a few simple SQL statements.

It is secure

MySQL consists of a solid data security layer that protects sensitive data from intruders. Also, passwords are encrypted in MySQL.

Client/ Server Architecture

MySQL follows the working of a client/server architecture. There is a database server (MySQL) and arbitrarily many clients (application programs), which communicate with the server; that is, they can query data, save changes, etc.

Free to download

MySQL is free to use so that we can download it from MySQL official website without any cost.

It is scalable

MySQL supports multi-threading that makes it easily scalable. It can handle almost any amount of data, up to as much as 50 million rows or more. The default file size limit is about 4 GB. However, we can increase this number to a theoretical limit of 8 TB of data.

Speed

MySQL is considered one of the very fast database languages, backed by a large number of the benchmark test.

High Flexibility

MySQL supports a large number of embedded applications, which makes MySQL very flexible.

Compatible on many operating systems

MySQL is compatible to run on many operating systems, like Novell NetWare, Windows* Linux*, many varieties of UNIX* (such as Sun* Solaris*, AIX, and DEC* UNIX), OS/2, FreeBSD*, and others. MySQL also provides a facility that the clients can run on the same computer as the server or on another computer (communication via a local network or the Internet).

Allows roll-back

MySQL allows transactions to be rolled back, commit, and crash recovery.

Memory efficiency

Its efficiency is high because it has a very low memory leakage problem.

High Performance

MySQL is faster, more reliable, and cheaper because of its unique storage engine architecture. It provides very high-performance results in comparison to other databases without losing an essential functionality of the software. It has fast loading utilities because of the different cache memory.

High Productivity

MySQL uses Triggers, Stored procedures, and views that allow the developer to give higher productivity.

Platform Independent

It can download, install, and execute on most of the available operating systems.

Partitioning

This feature improves the performance and provides fast management of the large database.

GUI Support

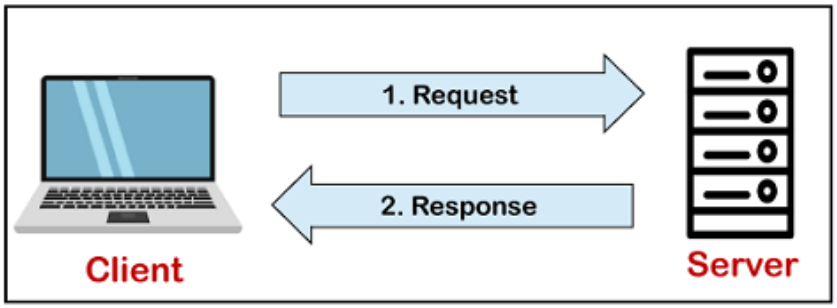
MySQL provides a unified visual database graphical user interface tool named "**MySQL Workbench**" to work with database architects, developers, and Database Administrators. MySQL Workbench provides SQL development, data modeling, data migration, and comprehensive administration tools for server configuration, user administration, backup, and many more. MySQL has a fully GUI supports from MySQL Server version 5.6 and higher.

Difference Between MYSQL and SQL:

MySQL	SQL
MySQL was owned by Oracle corporation.	SQL is developed by Microsoft Corporation.
MySQL is open source and accessible to any and everyone for free.	SQL is not open to others for free.
MySQL supports basic programming languages like C, C++, Python, Ruby, etc.	SQL is in itself a programming language used for database systems.
MySQL available only in the English language.	SQL is available in different languages.
MySQL doesn't support user-defined functions and XML.	SQL supports user-defined functions and XML.

How MySQL Works?

MySQL follows the working of Client-Server Architecture. This model is designed for the end-users called clients to access the resources from a central computer known as a server using network services. Here, the clients make requests through a graphical user interface (GUI), and the server will give the desired output as soon as the instructions are matched. The process of MySQL environment is the same as the client-server model.



Reasons for popularity

MySQL is becoming so popular because of these following reasons:

- MySQL is an open-source database, so you don't have to pay a single penny to use it.
- MySQL is a very powerful program that can handle a large set of functionality of the most expensive and powerful database packages.
- MySQL is customizable because it is an open-source database, and the open-source GPL license facilitates programmers to modify the SQL software according to their own specific environment.
- MySQL is quicker than other databases, so it can work well even with the large data set.
- MySQL supports many operating systems with many languages like PHP, PERL, C, C++, JAVA, etc.
- MySQL uses a standard form of the well-known SQL data language.
- MySQL is very friendly with PHP, the most popular language for web development.

Application of MySQL :

- MySQL used in E-Commerce websites.
- MySQL used in Data Warehousing.

- MySQL is used in the Login Application.

Disadvantages/Drawback of MySQL

Following are the few disadvantages of MySQL:

- MySQL version less than 5.0 doesn't support ROLE, COMMIT, and stored procedure.
- MySQL does not support a very large database size as efficiently.
- MySQL doesn't handle transactions very efficiently, and it is prone to data corruption.
- MySQL is accused that it doesn't have a good developing and debugging tool compared to paid databases.
- MySQL doesn't support SQL check constraints.

MySQL Data Types

A Data Type specifies a particular type of data, like integer, floating points, Boolean, etc. It also identifies the possible values for that type, the operations that can be performed on that type, and the way the values of that type are stored. In MySQL, each database table has many columns and contains specific data types for each column.

We can determine the data type in MySQL with the following characteristics:

- The type of values (fixed or variable) it represents.
- The storage space it takes is based on whether the values are a fixed-length or variable length.
- Its values can be indexed or not.
- How MySQL performs a comparison of values of a particular data type.

MySQL supports a lot number of SQL standard data types in various categories. It uses many different data types that can be broken into the following categories: numeric, date and time, string types, spatial types, and JSON data types.

Numeric Data Type

MySQL has all essential SQL numeric data types. These data types can include the exact numeric data types (For example, integer, decimal, numeric, etc.), as well as the approximate numeric data types (For example, float, real, and double precision). It also

supports BIT datatype to store bit values. In MySQL, numeric data types are categories into two types, either signed or unsigned except for bit data type.

The following table contains all numeric data types that support in MySQL:

Data Type Syntax	**Description**
TINYINT	It is a very small integer that can be signed or unsigned. If signed, the allowable range is from -128 to 127. If unsigned, the allowable range is from 0 to 255. We can specify a width of up to 4 digits. It takes 1 byte for storage.
SMALLINT	It is a small integer that can be signed or unsigned. If signed, the allowable range is from -32768 to 32767. If unsigned, the allowable range is from 0 to 65535. We can specify a width of up to 5 digits. It requires 2 bytes for storage.
MEDIUMINT	It is a medium-sized integer that can be signed or unsigned. If signed, the allowable range is from -8388608 to 8388607. If unsigned, the allowable range is from 0 to 16777215. We can specify a width of up to 9 digits. It requires 3 bytes for storage.
INT	It is a normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. We can specify a width of up to 11 digits. It requires 4 bytes for storage.
BIGINT	It is a large integer that can be signed or unsigned. If signed, the allowable range is from -9223372036854775808 to 9223372036854775807. If unsigned, the allowable range is from 0 to 18446744073709551615. We can specify a width of up to 20 digits. It requires 8 bytes for storage.
FLOAT(m,d)	It is a floating-point number that cannot be unsigned. You can define the display length (m) and the number of decimals (d). This is not required and will default to 10,2, where 2 is the number of decimals, and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a float type. It requires 2 bytes for storage.
DOUBLE(m,d)	It is a double-precision floating-point number that cannot be unsigned. You can define the display length (m) and the number of decimals (d). This is not required and will default to 16,4, where 4 is the number of decimals. Decimal precision can go to 53 places for a double. Real is a synonym for double. It requires 8 bytes for storage.

DECIMAL(m,d)	An unpacked floating-point number that cannot be unsigned. In unpacked decimals, each decimal corresponds to one byte. Defining the display length (m) and the number of decimals (d) is required. Numeric is a synonym for decimal.
BIT(m)	It is used for storing bit values into the table column. Here, M determines the number of bit per value that has a range of 1 to 64.
BOOL	It is used only for the true and false condition. It considered numeric value 1 as true and 0 as false.
BOOLEAN	It is Similar to the BOOL.

Date and Time Data Type:

This data type is used to represent temporal values such as date, time, datetime, timestamp, and year. Each temporal type contains values, including zero. When we insert the invalid value, MySQL cannot represent it, and then zero value is used.

The following table illustrates all date and time data types that support in MySQL:

Data Type Syntax	Maximum Size	Explanation
YEAR[(2|4)]	Year value as 2 digits or 4 digits.	The default is 4 digits. It takes 1 byte for storage.
DATE	Values range from '1000-01-01' to '9999-12-31'.	Displayed as 'yyyy-mm-dd'. It takes 3 bytes for storage.
TIME	Values range from '-838:59:59' to '838:59:59'.	Displayed as 'HH:MM:SS'. It takes 3 bytes plus fractional seconds for storage.
DATETIME	Values range from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.	Displayed as 'yyyy-mm-dd hh:mm:ss'. It takes 5 bytes plus fractional seconds for storage.
TIMESTAMP(m)	Values range from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' TC.	Displayed as 'YYYY-MM-DD HH:MM:SS'. It takes 4 bytes plus fractional seconds for storage.

String Data Types:

The string data type is used to hold plain text and binary data, for example, files, images, etc. MySQL can perform searching and comparison of string value based on the pattern matching such as LIKE operator, Regular Expressions, etc.

The following table illustrates all string data types that support in MySQL:

Data Type Syntax	Maximum Size	Explanation
CHAR(size)	It can have a maximum size of 255 characters.	Here size is the number of characters to store. Fixed-length strings. Space padded on the right to equal size characters.
VARCHAR(size)	It can have a maximum size of 255 characters.	Here size is the number of characters to store. Variable-length string.
TINYTEXT(size)	It can have a maximum size of 255 characters.	Here size is the number of characters to store.
TEXT(size)	Maximum size of 65,535 characters.	Here size is the number of characters to store.
MEDIUMTEXT(size)	It can have a maximum size of 16,777,215 characters.	Here size is the number of characters to store.
LONGTEXT(size)	It can have a maximum size of 4GB or 4,294,967,295 characters.	Here size is the number of characters to store.
BINARY(size)	It can have a maximum size of 255 characters.	Here size is the number of binary characters to store. Fixed-length strings. Space padded on the right to equal size characters. (introduced in MySQL 4.1.2)
VARBINARY(size)	It can have a maximum size of 255 characters.	Here size is the number of characters to store. Variable-length string. (introduced in MySQL 4.1.2)
ENUM	It takes 1 or 2 bytes that depend on the number of enumeration values. An ENUM can have a maximum of 65,535 values.	It is short for enumeration, which means that each column may have one of the specified possible values. It uses numeric indexes (1, 2, 3…) to represent string values.
SET	It takes 1, 2, 3, 4, or 8 bytes that depends on the number of set members. It can store a maximum of 64 members.	It can hold zero or more, or any number of string values. They must be chosen from a predefined list of values specified during table creation.

Binary Large Object Data Types (BLOB):

BLOB in MySQL is a data type that can hold a variable amount of data. They are categories into four different types based on the maximum length of values can hold.

The following table shows all Binary Large Object data types that support in MySQL:

Data Type Syntax	Maximum Size
TINYBLOB	It can hold a maximum size of 255 bytes.
BLOB(size)	It can hold the maximum size of 65,535 bytes.
MEDIUMBLOB	It can hold the maximum size of 16,777,215 bytes.
LONGBLOB	It can hold the maximum size of 4gb or 4,294,967,295 bytes.

Spatial Data Types

It is a special kind of data type which is used to hold various geometrical and geographical values. It corresponds to OpenGIS classes. The following table shows all spatial types that support in MySQL:

Data Types	Descriptions
GEOMETRY	It is a point or aggregate of points that can hold spatial values of any type that has a location.
POINT	A point in geometry represents a single location. It stores the values of X, Y coordinates.
POLYGON	It is a planar surface that represents multisided geometry. It can be defined by zero or more interior boundary and only one exterior boundary.
LINESTRING	It is a curve that has one or more point values. If it contains only two points, it always represents Line.
GEOMETRYCOLLECTION	It is a kind of geometry that has a collection of zero or more geometry values.
MULTILINESTRING	It is a multi-curve geometry that has a collection of linestring values.
MULTIPOINT	It is a collection of multiple point elements. Here, the point cannot be connected or ordered in any way.

MULTIPLYGON	It is a multisurface object that represents a collection of multiple polygon elements. It is a type of two-dimensional geometry.

JSON Data Type

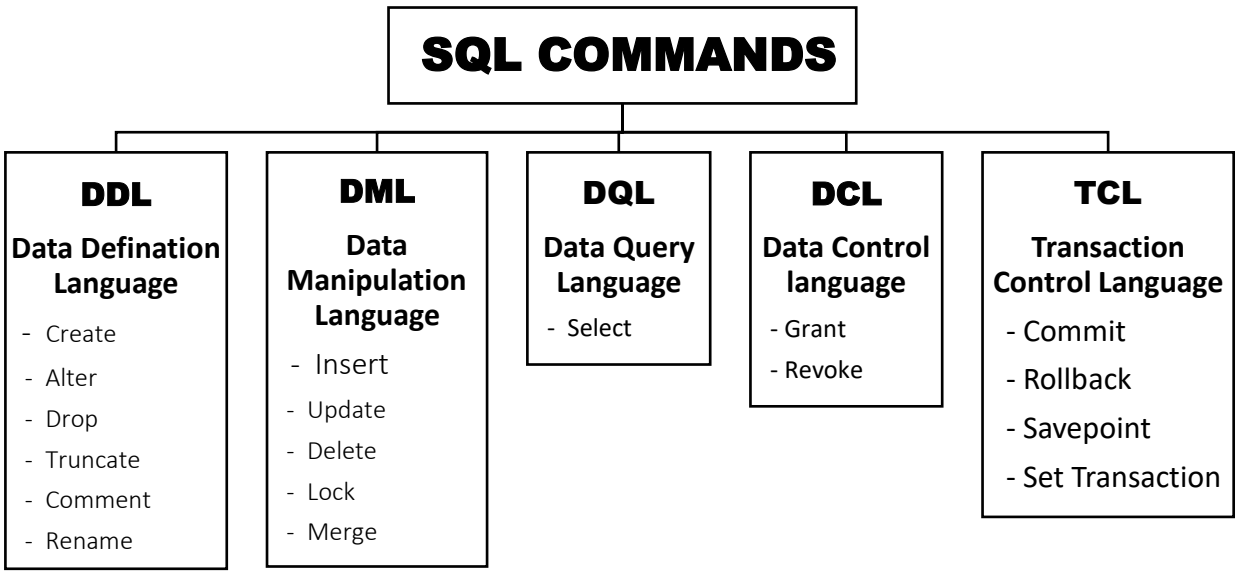
MySQL provides support for native JSON data type from the version v5.7.8. This data type allows us to store and access the JSON document quickly and efficiently.

The JSON data type has the following advantages over storing JSON-format strings in a string column:

1. It provides automatic validation of JSON documents. If we stored invalid documents in JSON columns, it would produce an error.
2. It provides an optimal storage format.

SQL Commands

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.



1. Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

1. **CREATE :-** It is used to create a new table in the database.

```
CREATE TABLE TABLE_NAME (
Column 1 datatype,
 Column 2 datatype,
----------
);
```

2. **DROP :-** It is used to delete both the structure and record stored in the table

   DROP TABLE table_name;

3. **ALTER :-** It is used to alter the structure of the database. Allow you to add, modify and delete columns of an existing table.

   CREATE TABLE Table_name Add column_name datatype;

4. **TRUNCATE :-** It is used to drop all the rows from the table. It is similar to the delete statement with no where clause.

   TRUNCATE TABLE table_name;

2. Data Manipulation Language (DML)

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

1. **INSERT :-** It is used to insert data into the row of a table.

   INSERT INTO TABLE_NAME
   (col1, col2, col3,.... col N)
   VALUES (value1, value2, value3, ... valueN);

2. **UPDATE :-** This command is used to update or modify the value of a column in the table.

   UPDATE table_name
   SET [column_name1= value1,...column_nameN = valueN]
   [WHERE CONDITION]

3. **DELETE :-** It is used to remove one or more row from a table.

   DELETE FROM table_name [WHERE condition];

3. Data Control Language (DCL)

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

1. **Grant :-** It is used to give user access privileges to a database.

   GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;

2. **Revoke :-** It is used to take back permissions from the user.

   REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;

4. Transaction Control Language (TCL)

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

1. **Commit :-** Commit command is used to save all the transactions to the database.

   ```
   DELETE FROM CUSTOMERS
   WHERE AGE = 25;
   COMMIT;
   ```

2. **Rollback :-** Rollback command is used to undo transactions that have not already been saved to the database.

   ```
   DELETE FROM CUSTOMERS
   WHERE AGE = 25;
   ROLLBACK;
   ```

3. **SAVEPOINT :-** It is used to roll the transaction back to a certain point without rolling back the entire transaction.

   ```
   SAVEPOINT SAVEPOINT_NAME;
   ```

5. Data Query Language (DQL)

DQL is used to fetch the data from the database.

It uses only one command:

SELECT :- This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

```
SELECT expressions
FROM TABLES
WHERE conditions;
```

 # Create Database Command.

```
create database test_db;
create database test1;
```

 # Get the list of existing Database.

```
show databases;
```

Database
information_schema
mysql
performance_schema
test1
test_db

 # Command to Drop Database.

```
drop database test1;
```

 # Go inside the particular database.

```
use test_db;
```

 # Command to create table.

```
create table if not exists Employees (
  id int,
  name VARCHAR(50),
  age int,
  hiring_date date,
  salary int,
```

```
city VARCHAR(50)
);
```

 # Insert data into the table.

```
insert into employees values(1, 'mujahid',24,'2021-08-12',10000, 'aligarh');
insert into employees values(2, 'ali',27,'2021-08-12',20000, 'pilibhit');
insert into employees values(3, 'kamran',26,'2021-08-17',11000, 'agra');
insert into employees values(4, 'azam',28,'2021-08-17',12000, 'bareilly');
insert into employees values(5, 'maya',29,'2021-08-19',17000, 'delhi');
```

 # Use select command to query the data.

```
select * from employees;
```

id	name	age	hiring_date	salary	city
1	mujahid	24	12-08-2021	10000	aligarh
2	ali	27	12-08-2021	20000	pilibhit
3	kamran	26	17-08-2021	11000	agra
4	azam	28	17-08-2021	12000	bareilly
5	maya	29	19-08-2021	17000	delhi

 # Add a new column named DOB in the table.

```
Alter table employees add DOB date;
select * from employees;
```

id	name	age	hiring_date	salary	city	DOB
1	mujahid	24	12-08-2021	10000	aligarh	NULL
2	ali	27	12-08-2021	20000	pilibhit	NULL
3	kamran	26	17-08-2021	11000	agra	NULL
4	azam	28	17-08-2021	12000	bareilly	NULL
5	maya	29	19-08-2021	17000	delhi	NULL

 # Modify existing column in a table or change of name column or increase lenght of name COLUMN

```
alter table employees modify column name varchar(100);
```

 # Rename the column name to emp_name

```
alter table employees rename column name to emp_name;
```

 # Delete existing column from given table or remove DOB column from employee table.

alter table employees drop column DOB;

select * from employees;

id	emp_name	age	hiring_date	salary	city
1	mujahid	24	12-08-2021	10000	aligarh
2	ali	27	12-08-2021	20000	pilibhit
3	kamran	26	17-08-2021	11000	agra
4	azam	28	17-08-2021	12000	bareilly
5	maya	29	19-08-2021	17000	delhi

 # Difference between Drop & Truncate command.

truncate table employees;

select * from employees;

id	emp_name	age	hiring_date	salary	city

drop table employees;

select * from employees;

Output :- Table doest not exist (both Schema as well as record deleted)

Operation with select command.

 # How to count total records.

select count(*) from employees;

 # Alias declaration.

select count(*) as total_rows_count from employees;

total_rows_count
5

 # Display specific column in the final RESULT

select name,salary from employees;

name	salary
mujahid	10000
ali	20000
kamran	11000
azam	12000
maya	17000

 # Print unique hiring_date from the employees table when employees joined it.

```
select distinct(hiring_date) from employees;
select distinct(hiring_date) as distinct_hiring_date from employees;
```

distinct_hiring_date
12-08-2021
17-08-2021
19-08-2021

 # How many unique age values in the table??

```
select count(distinct(age)) as total_unique_age from employees;
```

total_unique_age
5

 # Increment salary of each employee by 20% display final result with new salary

```
select id,
       name,
       salary as old_salary,
       (salary + salary*0.2) as new_salary
from employees;
```

id	name	old_salary	new_salary
1	mujahid	10000	12000
2	ali	20000	24000
3	kamran	11000	13200
4	azam	12000	14400
5	maya	17000	20400

Update command.

```
set SQL_SAFE_UPDATES = 0;
```

 # Update will be made for all rows

```
update employees set age = 20;
select * from employees;
```

id	name	age	hiring_date	salary	city
1	mujahid	20	12-08-2021	10000	aligarh
2	ali	20	12-08-2021	20000	pilibhit
3	kamran	20	17-08-2021	11000	agra
4	azam	20	17-08-2021	12000	bareilly
5	maya	20	19-08-2021	17000	delhi

 # Update the salary of employee after giving 20% increment.

```
update employees set salary = salary + salary * 0.2;
select * from employees;
```

id	name	age	hiring_date	salary	city
1	mujahid	20	12-08-2021	12000	aligarh
2	ali	20	12-08-2021	24000	pilibhit
3	kamran	20	17-08-2021	13200	agra
4	azam	20	17-08-2021	14400	bareilly
5	maya	20	19-08-2021	20400	delhi

 # Update the salary of employee who joined the company on 2021-08-17 to 80000

```
update employees set salary = 80000 where hiring_date = '2021-08-12';
select * from employees;
```

id	name	age	hiring_date	salary	city
1	mujahid	20	12-08-2021	80000	aligarh
2	ali	20	12-08-2021	80000	pilibhit
3	kamran	20	17-08-2021	13200	agra
4	azam	20	17-08-2021	14400	bareilly
5	maya	20	19-08-2021	20400	delhi

MySQL Constraints

The constraint in MySQL is used to specify the rule that allows or restricts what values/data will be stored in the table. They provide a suitable method to ensure data accuracy and integrity inside the table. It also helps to limit the type of data that will be inserted inside the table. If any interruption occurs between the constraint and data action, the action is failed.

Types of MySQL Constraints

Constraints in MySQL is classified into two types:

1. **Column Level Constraints:** These constraints are applied only to the single column that limits the type of particular column data.
2. **Table Level Constraints:** These constraints are applied to the entire table that limits the type of data for the whole table.

How to create constraints in MySQL

We can define the constraints during a table created by using the CREATE TABLE statement. MySQL also uses the ALTER TABLE statement to specify the constraints in the case of the existing table schema.

Constraints used in MySQL

The following are the most common constraints used in the MySQL:

- NOT NULL
- CHECK
- DEFAULT
- PRIMARY KEY
- AUTO_INCREMENT
- UNIQUE
- INDEX
- ENUM
- FOREIGN KEY

NOT NULL Constraint

1. This constraint specifies that the column cannot have NULL or empty values.
2. The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

```
Create table student (
Stud_id int NOT NULL,
Name varchar(50) NOT NULL,
Address varchar(100) NOT NULL
);
```

UNIQUE Constraint

1. This constraint ensures that all values inserted into the column will be unique. It means a column cannot stores duplicate values.
2. The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.
3. A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.
4. Note that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

```
Create table student (
stud_id int NOT NULL,
Name varchar(50) NOT NULL,
Address varchar(100) NOT NULL,
UNIQUE (stud_id)
);
```

SQL UNIQUE Constraint on ALTER TABLE

```
ALTER TABLE student ADD UNIQUE (stud_id)
```

CHECK Constraint

It controls the value in a particular column. It ensures that the inserted value in a column must be satisfied with the given condition.

1. The CHECK constraint is used to limit the value range that can be placed in a column.
2. If you define a CHECK constraint on a single column it allows only certain values for this column.
3. If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

```
Create table student (
stud_id int NOT NULL,
Name varchar(50) NOT NULL,
Address varchar(100) NOT NULL,
CHECK (stud_id > 0)
);
```

CHECK constraint on multiple columns,

```
Create table student (
stud_id int NOT NULL,
Name varchar(50) NOT NULL,
Address varchar(100) NOT NULL,
Constraint chk_student CHECK (stud_id > 0 AND Address = "abc")
);
```

SQL CHECK Constraint on ALTER TABLE

```
ALTER TABLE student ADD CHECK (student_id > 0)
```

SQL DEFAULT Constraint

1. The DEFAULT constraint is used to insert a default value into a column.

2. The default value will be added to all new records, if no other value is specified.

```
Create table student (
stud_id int NOT NULL,
Name varchar(50) NOT NULL,
Address varchar(100) NOT NULL,
City varchar(50) DEFAULT 'Aligarh'
);
```

AUTO INCREMENT

1. Auto-increment allows a unique number to be generated when a new record is inserted into a table.
2. Very often we would like the value of the primary key field to be created automatically every time a new record is inserted.

```
Create table student (
stud_id int NOT NULL AUTO_INCREMENT,
Name varchar(50) NOT NULL,
Address varchar(100) NOT NULL,
City varchar(50)
);
```

By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.

ENUM Constraint

The ENUM data type in MySQL is a string object. It allows us to limit the value chosen from a list of permitted values in the column specification at the time of table creation. It is short for enumeration, which means that each column may have one of the specified possible values. It uses numeric indexes (1, 2, 3…) to represent string values.

The following illustration creates a table named "shirts" that contains three columns: id, name, and size. The column name "size" uses the ENUM data type that contains small, medium, large, and x-large sizes.

```
CREATE TABLE Shirts (
        id INT PRIMARY KEY AUTO_INCREMENT,
        name VARCHAR(35),
        size ENUM('small', 'medium', 'large', 'x-large')

INSERT INTO Shirts(id, name, size)
    VALUES (1,'t-shirt', 'medium'),
                (2, 'casual-shirt', 'small'),
                (3, 'formal-shirt', 'large');

    SELECT * FROM Shirts;
```

id	name	size
1	t-shirt	medium
2	casual-shirt	small
3	formal-shirt	large

```
CREATE table if not exists employee_with_constraints
(
   id int NOT NULL,
   name VARCHAR(50) NOT NULL,
   salary DOUBLE,
   hiring_date Date DEFAULT '2021-09-25',
   UNIQUE(ID),
   CHECK (salary > 1000)
);
```

Example 1 for integrity constraint failure

Exception will be thrown -> column 'id' cannot be Null

```
insert into employee_with_constraints values(null,"mujahid", 1000, '2023-03-22');
```

Example 2 for integrity constraint failure

Exception will be thrown -> column 'name' cannot be Null

```
insert into employee_with_constraints values(3,null, 1000, '2023-03-22');
```

Example 3 for integrity constraint failure
Exception will be thrown -> Check constraint 'employee_with_constraints_chk_1' is violated.
insert into employee_with_constraints values(1,"mujahid", 500, '2023-03-22');

insert correct data
insert into employee_with_constraints values(1,"mujahid", 1500, '2023-03-22');

Eexample 4 for integrity constraint failure
Exception will be thrown -> Duplicate entry '1' for key 'employee_with_constraints.id'
insert into employee_with_constraints values(1,"mujahid", 1200, '2023-03-24');

Example 5 for integrity constraint
insert into employee_with_constraints values(2,"mujahid", 1200, null);
select * from employee_with_constraints;

id	emp_name	salary	hiring_date
2	mujahid	1200	NULL

Drop constraint from existing table

alter table employee drop constraint id_unique;

Add unique integrity constraint on id COLUMN

alter table employee add constraint id_unique UNIQUE(id);

Operators

In simple operator can be defined as an entity used to perform operations in a table.

Operators are the foundation of any programming language. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands.

Types of SQL Operators

- Arithmetic operator
- Comparison operator
- Logical operator

Arithmetic Operators

We can use various arithmetic operators on the data stored in the tables. Arithmetic Operators are:

Operator	Description
+	The addition is used to perform an addition operation on the data values.
–	This operator is used for the subtraction of the data values.
/	This operator works with the 'ALL' keyword and it calculates division operations.
*	This operator is used for multiplying data values.
%	Modulus is used to get the remainder when data is divided by another.

Comparison Operators

Another important operator in SQL is a comparison operator, which is used to compare one expression's value to other expressions. SQL supports different types of comparison operator, We will use the **WHERE** command along with the conditional operator to achieve this in SQL. which is described below:

Operator	Description
=	Equal to.
>	Greater than.
<	Less than.
>=	Greater than equal to.
<=	Less than equal to.
<>	Not equal to.

Logical Operators

The Logical operators are those that are true or false. They return true or false values to combine one or more true or false values.

In SQL, the AND & OR operators are used for filtering the data and getting precise results based on conditions. The SQL **AND** & **OR** operators are also used to combine multiple conditions. These two operators can be combined to test for multiple conditions in a SELECT, INSERT, UPDATE, or DELETE statement.

When combining these conditions, it is important to use parentheses so that the database knows what order to evaluate each condition.

- The AND and OR operators are used with the WHERE clause.
- These two operators are called conjunctive operators.

Operator	Description
AND	Logical AND compares two Booleans as expressions and returns true when both expressions are true.
OR	Logical OR compares two Booleans as expressions and returns true when one of the expressions is true.
NOT	Not takes a single Boolean as an argument and change its value from false to true or from true to false.

Special Operators

Operators	Description
ALL	ALL is used to select all records of a SELECT STATEMENT. It compares a value to every value in a list of results from a query. The ALL must be preceded by the comparison operators and evaluated to TRUE if the query returns no rows.
ANY	ANY compares a value to each value in a list of results from a query and evaluates to true if the result of an inner query contains at least one row.
BETWEEN	The SQL BETWEEN operator tests an expression against a range. The range consists of a beginning, followed by an AND keyword and an end expression.
IN	The IN operator checks a value within a set of values separated by commas and retrieves the rows from the table that match.

EXISTS	The EXISTS checks the existence of a result of a subquery. The EXISTS subquery tests whether a subquery fetches at least one row. When no data is returned then this operator returns 'FALSE'.
SOME	SOME operator evaluates the condition between the outer and inner tables and evaluates to true if the final result returns any one row. If not, then it evaluates to false.
UNIQUE	The UNIQUE operator searches every unique row of a specified table.

Arithmetic Operators

Table :- employee

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
2	Raza	27	12-08-2021	20000	pilibhit
3	Smith	26	17-08-2021	11000	agra
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

```
select    id, `name`, salary, salary + 100 as new_salary from employee;
```

id	name	salary	new_salary
1	Alex	10000	10100
2	Raza	20000	20100
3	Smith	11000	11100
4	Anna	12000	12100
5	Maya	17000	17100

select id, `name`, salary, salary * id as update_salary from employee;

id	name	salary	update_salary
1	Alex	10000	10000
2	Raza	20000	40000
3	Smith	11000	33000
4	Anna	12000	48000
5	Maya	17000	85000

Comparison Operators

Query :- list all employees who are getting salary more than 12000;

select * from employee where salary > 12000;

id	name	age	hiring_date	salary	city
2	Raza	27	12-08-2021	20000	pilibhit
5	Maya	29	19-08-2021	17000	delhi

Query :- list all employees who are getting salary more than or equal to 20000;

select * from employee where salary >= 20000;

id	name	age	hiring_date	salary	city
2	Raza	27	12-08-2021	20000	pilibhit

Query :- list all employees who are getting salary less than 12000;

select * from employee where salary < 12000;

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
3	Smith	26	17-08-2021	11000	agra

Query :- list all employees who are getting salary less than or equal to 12000;

```
select * from employee where salary <= 12000;
```

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
3	Smith	26	17-08-2021	11000	agra
4	Anna	28	17-08-2021	12000	bareilly

Query :- filter the records where age of employees is equal to 27

```
select * from employee where age = 27;
```

id	name	age	hiring_date	salary	city
2	Raza	27	12-08-2021	20000	pilibhit

Query :- filter the records where age of employees is not equal to 27

```
select * from employee where age != 27;
select * from employee where age <> 27;
```

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
3	Smith	26	17-08-2021	11000	agra
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

Logical Operators

Query :- find the employee who join the company on '2021-08-17' and their salary is less than 11500

```
select * from employee where hiring_date = '2021-08-17' and salary < 11500;
```

id	name	age	hiring_date	salary	city
3	Smith	26	17-08-2021	11000	agra

Query :- find the employee who join the company after '2021-08-17' or their salary is less than 15000

```
select * from employee where hiring_date > '2021-08-17' or salary < 15000;
```

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
3	Smith	26	17-08-2021	11000	agra
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

Query :-

```
select * from employee where not city = 'agra';
```

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
2	Raza	27	12-08-2021	20000	pilibhit
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

Between Operator

The SQL BETWEEN condition allows you to easily test if an expression is within a range of values (inclusive). The values can be text, date, or numbers. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

Query :- Get all employees date who joined the company between hiring_date '2021-08-15' to '2021-08-20'

```
select * from employee where hiring_date between '2021-08-15' and '2021-08-20';
```

id	name	age	hiring_date	salary	city
3	Smith	26	17-08-2021	11000	agra
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

Query :- Get all employees date who are getting salary in the range of 15000 to 28000.

```
select * from employee where salary between 15000 and 28000;
```

id	name	age	hiring_date	salary	city
2	Raza	27	12-08-2021	20000	pilibhit
5	Maya	29	19-08-2021	17000	delhi

Like Operator

LIKE operator is used to perform pattern matching to find the correct result. It is used in SELECT, INSERT, UPDATE and DELETE statement with the combination of WHERE clause.

% Wildcard used zero, one or more than one characters.
_ **Wildcard** used only one character.

Query :- Get all the employees whose name start with 'A'

```
select * from employee where name like 'A%';
```

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
4	Anna	28	17-08-2021	12000	bareilly

Query :- Get all the employees whose name end with 'x'

```
select * from employee where name like '%x';
```

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh

Query :- Get all the employees whose name start with 'R' and end with 'a'

```
select * from employee where name like 'R%a';
```

id	name	age	hiring_date	salary	city
2	Raza	27	12-08-2021	20000	pilibhit

Query :- Get all those employees whose name will exact 4 characters

```
select * from employee where name like '____';
```

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
2	Raza	27	12-08-2021	20000	pilibhit
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

IS NULL & IS NOT NULL Operator

Table : - persons

P_id	P_name	age	salary
1001	Adam	27	NULL
1002	HR	NULL	273543

Query :- Get all those persons whose age value is null

```
select * from persons where age Is Null;
```

Query :- Get all those persons whose salary is not null

```
select * from persons where salary Is not Null;
```

P_id	P_name	age	salary
1002	HR	NULL	273543

IN Operator

The MySQL IN condition is used to reduce the use of multiple OR conditions in a SELECT, INSERT, UPDATE and DELETE statement.

```
SELECT *
FROM table
WHERE column_name IN (value1, value2.....);
```

Table :- employee

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
2	Raza	27	12-08-2021	20000	pilibhit
3	Smith	26	17-08-2021	11000	agra
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

Query :-

```
SELECT *
FROM employee
WHERE `name` IN ('Raza', 'Anna', 'Maya' );
```

id	name	age	hiring_date	salary	city
2	Raza	27	12-08-2021	20000	pilibhit
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

Keys

Keys are one of the basic requirements of a relational database model. It is widely used to identify the tuples(rows) uniquely in the table. We also use keys to set up relations amongst various columns and tables of a relational database.

Different Types of Keys :-

1. Candidate Key
2. Primary Key
3. Super Key
4. Alternate Key
5. Foreign Key
6. Composite Key

Primary Key:

There can be more than one candidate key in students table out of which one can be chosen as the primary key. For Example, **stud_id**, as well as **phone**, are candidate keys for students table but **stud_id** can be chosen as the primary key (only one out of many candidate keys).

```
create table students
(
stud_id int,
stud_name varchar(20),
address varchar(50),
phone Bigint,
primary key(stud_id)
);
insert into students values(01, 'Raza' , 'Aligarh', 9867453210);
insert into students values(02, 'Amaan' , 'Pilibhit', 7865432546);
insert into students values(03, 'Ahmad' , 'Agra', 9876563420);
insert into students values(04, 'Sahil' , 'Delhi', 9078654328);

select * from students;
```

stud_id	stud_name	address	phone
1	Raza	Aligarh	9867453210
2	Amaan	Pilibhit	7865432546
3	Ahmad	Agra	9876563420
4	Sahil	Delhi	9078654328

- Primary key is used to identify each record in a table **uniquely**.
- Each table can contain only one primary key.
- The primary key column cannot be null or empty.
- It is a unique key.
- MySQL does not allow us to insert a new row with the existing primary key.
- It is recommended to use INT or BIGINT data type for the primary key column.

We can create a primary key in two ways:

- CREATE TABLE Statement
- ALTER TABLE Statement

Primary Key Using CREATE TABLE Statement :-

```
CREATE TABLE table_name(
    col1 datatype PRIMARY KEY,
    col2 datatype,
    ...
);
```

```
CREATE TABLE table_name(
    col1 INT AUTO_INCREMENT PRIMARY KEY,
    col2 datatype,
    ...
 );
```

Primary Key Using ALTER TABLE Statement :-

```
ALTER TABLE table_name ADD PRIMARY KEY(column_list);
```

Example : alter table students ADD PRIMARY KEY(stud_id);

DROP Primary Key :-

```
ALTER TABLE table_name  DROP PRIMARY KEY;
```

Primary Key vs. Unique Key :-

Both has to store distint/unique value.

Primary Key	Unique Key
It is used to identify each record in a table uniquely.	It also determines each row of the table uniquely in the absence of a primary key.
It does not allow to store a NULL value into the primary key column.	It can accept only one NULL value into the unique key column.
A table can have only one primary key.	A table can have more than one unique key.
It creates a clustered index.	It creates a non-clustered index.

Candidate Key :-

A part from primary key if there is any column or set of column which can uniquely identify record.

A characteristic or group of attributes that can uniquely identify a tuple is known as a candidate key in MySQL Keys.

The remaining properties, with the exception of the primary key, are considered candidate keys. Candidates have the same strength as the primary key.

Take a look at the "**students**" table as an example. **Stud_id, stud_name, address** and **phone** are the four attributes in this table. **Stud_id** and **phone** will have unique values, however**, stud_name** and **address** may contain duplicate values because multiple students may have the same name.

The candidate keys here are [**stud_id**] and [**phone**].

- It is a minimal super key.
- It is a super key with no repeated data is called a candidate key.

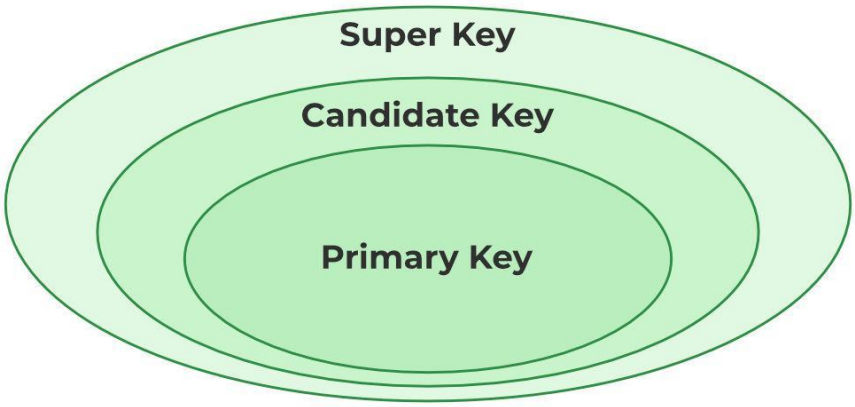
- The minimal set of attributes that can uniquely identify a record.
- It must contain unique values.
- It can contain NULL values.
- Every table must have at least a single candidate key.
- A table can have multiple candidate keys but only one primary key (the primary key cannot have a NULL value, so the candidate key with a NULL value can't be the primary key).

Super Key :-

The set of attributes that can uniquely identify a tuple is known as Super Key. A super key is a group of single or multiple keys that identifies rows in a table. It supports NULL values.

- Adding zero or more attributes to the candidate key generates the super key.
- A candidate key is a super key but vice versa is not true.

For example, (**stud_id, stud_name**) in the following **students** database, two students' **names** can be the same, but their **stud_id** cannot

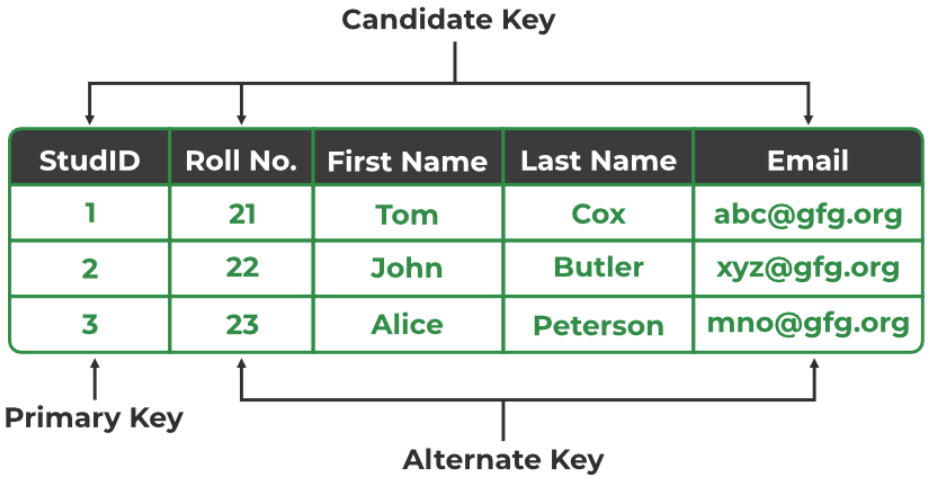


Alternate Key :-

The candidate key other than the primary key is called an alternate key.

- All the keys which are not primary keys are called alternate keys.
- It is a secondary key.
- It contains two or more fields to identify two or more records.
- These values are repeated.

Example : - In the **students** table, **stud_id** and **phone** are candidate key for student but **phone** will be an **alternat**e key.



Foreign Key :-

The foreign key is used to link one or more than one table together. It is also known as the **referencing** key. A foreign key matches the primary key field of another table. It means a foreign key field in one table refers to the primary key field of the other table. It identifies each row of another table uniquely that maintains the **referential integrity** in MySQL.

A foreign key makes it possible to create a parent-child relationship with the tables. In this relationship, the parent table holds the initial column values, and column values of child table reference the parent column values. MySQL allows us to define a foreign key constraint on the child table.

- It is a key it acts as a primary key in one table and it acts as secondary key in another table.
- It combines two or more relations (tables) at a time.
- They act as a cross-reference between the tables.

Syntax

Following are the basic syntax used for defining a foreign key using CREATE TABLE OR ALTER TABLE statement in the MySQL:

```
[CONSTRAINT constraint_name]
FOREIGN KEY [foreign_key_name] (col_name, ...)
REFERENCES parent_tbl_name (col_name,...)
ON DELETE referenceOption
ON UPDATE referenceOption
```

Constraint_name :- It specifies the name of the foreign key constraint. If we have not provided the constraint name, MySQL generates its name automatically.

Col_name :- It is the names of the column that we are going to make foreign key.

Parent_tbl_name :- It specifies the name of a parent table followed by column names that reference the foreign key columns.

Refrence_option :- It is used to ensure how foreign key maintains referential integrity using ON DELETE and ON UPDATE clause between parent and child table.

MySQL contains **five** different referential options, which are given below:

CASCADE :- It is used when we delete or update any row from the parent table, the values of the matching rows in the child table will be deleted or updated automatically.

SET NULL :- It is used when we delete or update any row from the parent table, the values of the foreign key columns in the child table are set to NULL.

RESTRICT :- It is used when we delete or update any row from the parent table that has a matching row in the reference(child) table, MySQL does not allow to delete or update rows in the parent table.

NO ACTION :- It is similar to RESTRICT. But it has one difference that it checks referential integrity after trying to modify the table.

SET DEFAULT :- The MySQL parser recognizes this action. However, the InnoDB and NDB tables both rejected this action.

```
create table persons
(
person_id int auto_increment,
person_name varchar(50),
Age int,
primary key(person_id)
);

insert into persons (person_name, Age) values
('Virat', 32),
('Rohit', 33),
```

```
('Rahul' ,30);

select * from persons;
```

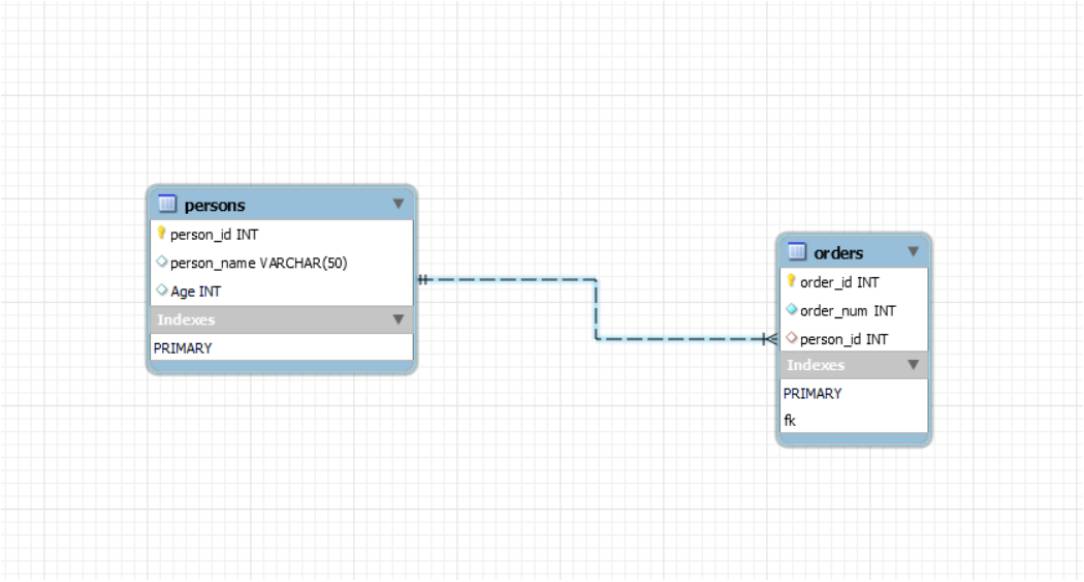
person_id	person_name	Age
1	Virat	32
2	Rohit	33
3	Rahul	30

FOREIGN KEY on CREATE TABLE :-

```
create table orders
(
order_id int not null auto_increment,
order_num int not null,
person_id int,
constraint pk primary key (order_id),
constraint fk foreign key (person_id) references persons(person_id)
);

insert into orders (order_num, person_id) values
(4563, 2),
(6745, 1),
(3454, 2),
(0089, 3);

select * from orders;
```

order_id	order_num	person_id
1	4563	2
2	6745	1
3	3454	2
4	89	3

EER Diagram :-



The "PersonID" column in the "Persons" table is the **PRIMARY KEY** in the "Persons" table.

The "PersonID" column in the "Orders" table is a **FOREIGN KEY** in the "Orders" table.

The **FOREIGN KEY** constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

FOREIGN KEY on ALTER TABLE :-

ALTER TABLE orders

ADD CONSTRAINT **FK**

FOREIGN KEY **(person_id)** REFERENCES **persons(person_id);**

DROP a FOREIGN KEY Constraint :-

```
ALTER TABLE Orders
DROP FOREIGN KEY FK;
```

Foreign Key Checks

MySQL has a special variable **foreign_key_cheks** to control the foreign key checking into the tables. By default, it is enabled to enforce the referential integrity during the normal operation

on the tables. This variable is dynamic in nature so that it supports global and session scopes both.

Sometimes there is a need for disabling the foreign key checking, which is very useful when:

- We drop a table that is a reference by the foreign key.
- We import data from a CSV file into a table. It speeds up the import operation.
- We use ALTER TABLE statement on that table which has a foreign key.
- We can execute load data operation into a table in any order to avoid foreign key checking.

The following statement allows us to **disable** foreign key checks:

```
SET foreign_key_checks = 0;
```

The following statement allows us to **enable** foreign key checks:

```
SET foreign_key_checks = 1;
```

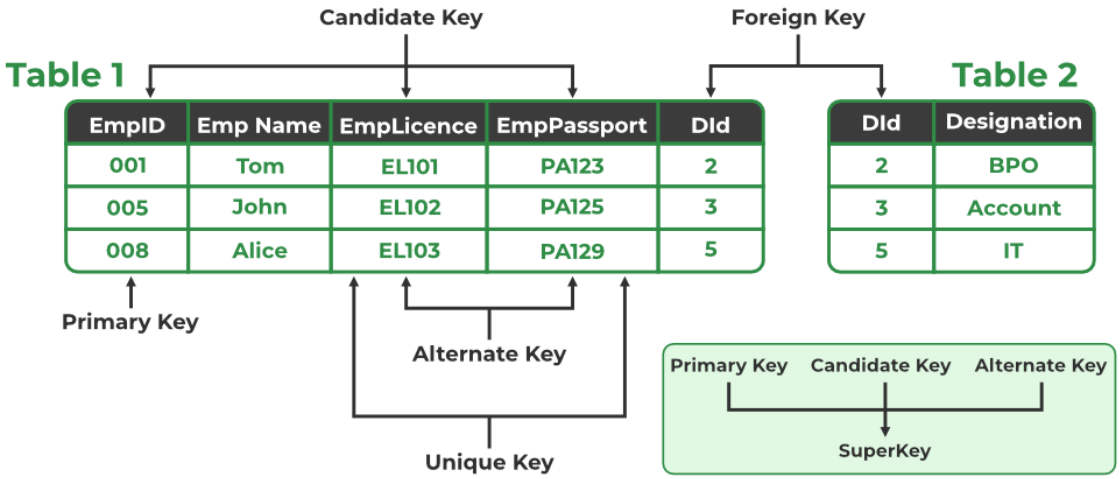
Composite Key :-

Sometimes, a table might not have a single column/attribute that uniquely identifies all the records of a table. To uniquely identify rows of a table, a combination of two or more columns/attributes can be used. It still can give duplicate values in rare cases. So, we need to find the optimal set of attributes that can uniquely identify rows in a table.

- It acts as a primary key if there is no primary key in a table
- Two or more attributes are used together to make a composite key.
- Different combinations of attributes may give different accuracy in terms of identifying the rows uniquely.

Example:

FULLNAME + DOB can be combined
together to access the details of a student.



What is Artificial Key?

Artificial Keys are the keys that are used when no attributes contain all the properties of the Primary Key or if the Primary key is very large and complex.

Difference between Primary and Candidate Key:

S.NO	Primary Key	Candidate Key
1.	Primary key is a minimal super key. So there is one and only one primary key in a relation.	While in a relation there can be more than one candidate key.
2.	Any attribute of Primary key can not contain NULL value.	While in Candidate key any attribute can contain NULL value.
3.	Primary key can be optional to specify any relation.	But without candidate key there can't be specified any relation.
4.	Primary key specifies the important attribute for the relation.	Candidate specifies the key which can qualify for primary key.
5.	Its confirmed that a primary key is a candidate key.	But Its not confirmed that a candidate key can be a primary key.

Difference between Primary Key and Foreign Key

PRIMARY KEY	FOREIGN KEY
A primary key is used to ensure data in the specific column is unique.	A foreign key is a column or group of columns in a relational database table that provides a link between data in two tables.
It uniquely identifies a record in the relational database table.	It refers to the field in a table which is the primary key of another table.
Only one primary key is allowed in a table.	Whereas more than one foreign key is allowed in a table.
It is a combination of UNIQUE and Not Null constraints.	It can contain duplicate values and a table in a relational database.
It does not allow NULL values.	It can also contain NULL values.
Its value cannot be deleted from the parent table.	Its value can be deleted from the child table.
It constraint can be implicitly defined on the temporary tables.	It constraint cannot be defined on the local or global temporary tables.

Clauses

- **Where clause**
- **Distinct**
- **From**
- **Order By**
- **Group By**
- **Having**

Where clause :-

WHERE Clause is used with SELECT, INSERT, UPDATE and DELETE clause to filter the results. It specifies a specific position where you have to do the operation.

Syntax:

```
SELECT expressions
FROM tables
[WHERE conditions];
```

Table : - orders_data

cust_id	order_id	country	state	order_pay
1	100	USA	Seattle	454
2	101	INDIA	UP	542
2	103	INDIA	Bihar	870
4	108	USA	WDC	432
5	109	UK	London	934
4	110	USA	WDC	765
3	120	INDIA	AP	234
2	121	INDIA	Goa	700
1	131	USA	Seattle	800
6	142	USA	Seattle	650
7	150	USA	Seattle	760

select * from orders_data where state = 'WDC';

cust_id	order_id	country	state	order_pay
4	108	USA	WDC	432
4	110	USA	WDC	765

select * from orders_data where state = 'WDC' and order_id < 110;

cust_id	order_id	country	state	order_pay
4	108	USA	WDC	432

select * from orders_data where state = 'WDC' or state = 'London';

cust_id	order_id	country	state	order_pay
4	108	USA	WDC	432
5	109	UK	London	934
4	110	USA	WDC	765

Distinct Clause :-

MySQL DISTINCT clause is used to remove duplicate records from the table and fetch only the unique records. The DISTINCT clause is only used with the SELECT statement.

Syntax:

SELECT DISTINCT expressions
FROM tables
[**WHERE** conditions];

select distinct country from orders_data;

country
USA
INDIA
UK

FROM Clause :-

The MySQL FROM Clause is used to select some records from a table. It can also be used to retrieve records from multiple tables using JOIN condition.

select * from orders_data where cust_id = 2;

cust_id	order_id	country	state	order_pay
2	101	INDIA	UP	542
2	103	INDIA	Bihar	870
2	121	INDIA	Goa	700

ORDER BY Clause :-

ORDER BY Clause is used to sort the records in ascending or descending order.

Syntax:

```
SELECT expressions
FROM tables
[WHERE conditions]
ORDER BY expression [ ASC | DESC ];
```

select * from orders_data where country = 'INDIA' order by order_pay desc;

cust_id	order_id	country	state	order_pay
2	103	INDIA	Bihar	870
2	121	INDIA	Goa	700
2	101	INDIA	UP	542
3	120	INDIA	AP	234

GROUP BY Clause :-

The **Group By** statement is used for organizing similar data into groups. The data is further organized with the help of equivalent function. It means, if different rows in a precise column have the same values, it will arrange those rows in a group.

The SELECT statement is used with the **GROUP BY** clause in the SQL query.

WHERE clause is placed before the **GROUP BY** clause in **SQL**.

ORDER BY clause is placed after the **GROUP BY** clause in **SQL**.

- You can also use some aggregate functions like COUNT, SUM, MIN, MAX, AVG etc. on the grouped column.
- A part from group by columns, we can use other columns in aggregation functions. They can not be used in select statement.

Syntax:

```
SELECT expression1, expression2, ... expression_n,
aggregate_function (expression)
FROM tables
[WHERE conditions]
GROUP BY expression1, expression2, ... expression_n;
```

Table : - orders_data

cust_id	order_id	country	state	order_pay
1	100	USA	Seattle	454
2	101	INDIA	UP	542
2	103	INDIA	Bihar	870
4	108	USA	WDC	432
5	109	UK	London	934
4	110	USA	WDC	765
3	120	INDIA	AP	234
2	121	INDIA	Goa	700
1	131	USA	Seattle	800
6	142	USA	Seattle	650
7	150	USA	Seattle	760

Query :- Calculate total order placed country wise.

select country, count(*) as order_count_by_each_country from orders_data group by country;

country	order_count_by_each_country
USA	6
INDIA	4
UK	1

Query :- Write a query to find the total order pay by each state

select state, sum(order_pay) as total_order_pay_by_each_state from orders_data group by state;

state	total_order_pay_by_each_state
Seattle	2664
UP	542
Bihar	870
WDC	1197
London	934
AP	234
Goa	700

Query :- Calculate total aggregated matrices for state.

```
select state,
        max(order_pay) as max_order_pay_by_each_state,
        min(order_pay) as min_order_pay_by_each_state,
        avg(order_pay) as avg_order_pay_by_each_state
from orders_data group by state;
```

state	max_order_pay_by_ each_state	min_order_pay_by_ each_state	avg_order_pay_by_ each_state
Seattle	800	454	666
UP	542	542	542
Bihar	870	870	870
WDC	765	432	598.5
London	934	934	934
AP	234	234	234
Goa	700	700	700

HAVING Clause :-

MySQL HAVING Clause is used with GROUP BY clause. It always returns the rows where condition is TRUE.

Syntax:

```
SELECT expression1, expression2, ... expression_n,
aggregate_function (expression)
FROM tables
[WHERE conditions]
GROUP BY expression1, expression2, ... expression_n
HAVING condition;
```

Query :- Write a query to find the country where only 1 order was placed.

select country from orders_data group by country Having count(*)=1;

country
UK

How to use GROUP_CONCAT

Query :- Write a query to print DISTINCT states present in the dataset for each country.

SELECT country, GROUP_CONCAT(distinct state) as states_in_country from orders_data group by country;

country	states_in_country
INDIA	AP,Bihar,Goa,UP
UK	London
USA	Seattle,WDC

SELECT country, GROUP_CONCAT(distinct state order by state desc) as states_in_country from orders_data group by country;

country	states_in_country
INDIA	UP,Goa,Bihar,AP
UK	London
USA	WDC,Seattle

SELECT country, GROUP_CONCAT(distinct state order by state desc SEPARATOR '_') as states_in_country from orders_data group by country;

country	states_in_country
INDIA	UP_Goa_Bihar_AP
UK	London
USA	WDC_Seattle

Use of IN and NOT IN

Query :- Write a query to print all the orders which were placed in 'seattle' or 'Goa'

Wrong way -> select * from orders_data where state = 'Seattle' or state 'Goa';

select * from orders_data where state in ('Seattle', 'Goa');

cust_id	order_id	country	state	order_pay
1	100	USA	Seattle	454
2	121	INDIA	Goa	700
1	131	USA	Seattle	800
6	142	USA	Seattle	650
7	150	USA	Seattle	760

SET operation :-

1. **UNION**
2. **UNION ALL**
3. **INTERSECT**
4. **EXCEPT**

Union and Union All

The Union Clause is used to combine two separate select statements and produce the result set as a union of both select statements. It comes with a default feature that removes the **duplicate** rows from the result set.

NOTE:

1. The fields to be used in both the select statements must be in the same order, same number, and same data type.
2. The Union clause produces distinct values in the result set, to fetch the duplicate values too UNION ALL must be used instead of just UNION.

```
SELECT column_list FROM table1
UNION
SELECT column_list FROM table2;
```

Table 1	
Column 1	Column 2
A	1
A	2
A	3

∪

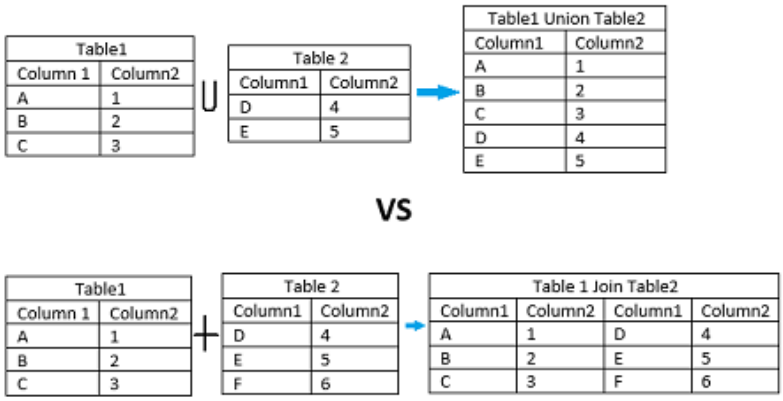
Table 2	
Column 1	Column 2
A	1
B	2
C	3

=

Table 1 Union Table 2	
Column 1	Column 2
A	1
A	2
A	3
B	2
C	3

Union vs. Join:-

The Union and Join clause are different because a union always combines the result set **vertically** while the join appends the output **horizontally**. We can understand it with the following visual representation:



JOIN	UNION
JOIN combines data from many tables based on a matched condition between them	SQL combines the result set of two or more SELECT statements.
It combines data into new columns.	It combines data into new rows
The number of columns selected from each table may not be the same.	The number of columns selected from each table should be the same.
Datatypes of corresponding columns selected from each table can be different.	The data types of corresponding columns selected from each table should be the same.
It may not return distinct columns.	It returns distinct rows.

Table :- student

stu_id	name	email	city
1	Mujahid	abc@gmail.com	Pilibhit
2	Raza	abc1@gmail.com	Aligarh
3	Amaan	abc2@gmail.com	Noida
4	Ahmad	abc3@gmail.com	Bareilly
5	Siddiqui	abc4@gmail.com	US Nagar

Table :- student2

stu_id	name	email	city
1	Mujahid	abc@gmail.com	Pilibhit
6	Ali	abc5@gmil.com	Delhi
8	Zeeshu	abc6@gmail.com	Mongli
10	zubair	abc7@gmail.com	Kudarki
5	Siddiqui	abc4@gmail.com	US Nagar

Query :- We are organising an tournament between college-1 and college-2, we need details of all students from both college.

```
select * from student
UNION
select * from student2;
```

stu_id	name	email	city
1	Mujahid	abc@gmail.com	Pilibhit
2	Raza	abc1@gmail.com	Aligarh
3	Amaan	abc2@gmail.com	Noida
4	Ahmad	abc3@gmail.com	Bareilly
5	Siddiqui	abc4@gmail.com	US Nagar
6	Ali	abc5@gmil.com	Delhi
8	Zeeshu	abc6@gmail.com	Mongli
10	zubair	abc7@gmail.com	Kudarki

```
select * from student
UNION ALL
select * from student2;
```

stu_id	name	email	city
1	Mujahid	abc@gmail.com	Pilibhit
2	Raza	abc1@gmail.com	Aligarh
3	Amaan	abc2@gmail.com	Noida
4	Ahmad	abc3@gmail.com	Bareilly
5	Siddiqui	abc4@gmail.com	US Nagar
1	Mujahid	abc@gmail.com	Pilibhit
6	Ali	abc5@gmil.com	Delhi
8	Zeeshu	abc6@gmail.com	Mongli
10	zubair	abc7@gmail.com	Kudarki
5	Siddiqui	abc4@gmail.com	US Nagar

--- Case 1 -

```
select stu_id, name from student
UNION
select name, stu_id from student2;
```

stu_id	name
1	Mujahid
2	Raza
3	Amaan
4	Ahmad
5	Siddiqui
Mujahid	1
Ali	6
Zeeshu	8
zubair	10
Siddiqui	5

--- Case 2 -

```
select stu_id, name from student
UNION
select stu_id, name from student2;
```

stu_id	name
1	Mujahid
2	Raza
3	Amaan
4	Ahmad
5	Siddiqui
6	Ali
8	Zeeshu
10	zubair

--- Case 3 -

```
select stu_id as stu_id_college1, name from student
UNION
select stu_id as stu_id_college2, name from student2;
```

stu_id_college1	name
1	Mujahid
2	Raza
3	Amaan
4	Ahmad
5	Siddiqui
6	Ali
8	Zeeshu
10	zubair

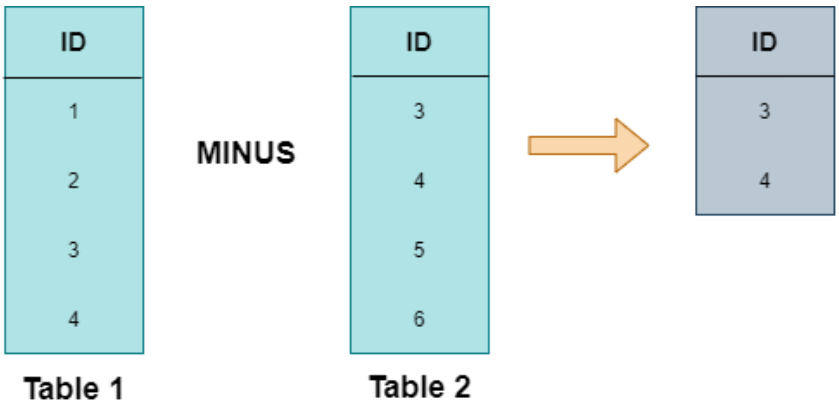
--- Case 4 -

```
select stu_id from student
UNION
select email from student2;
```

stu_id
1
2
3
4
5
abc@gmail.com
abc5@gmil.com
abc6@gmail.com
abc7@gmail.com
abc4@gmail.com

INTERSECT :-

The INTERSECT operator returns the distinct (common) elements in two sets or common records from two or more tables. In other words, it compares the result obtained by two queries and produces unique rows, which are the result returned by both queries.



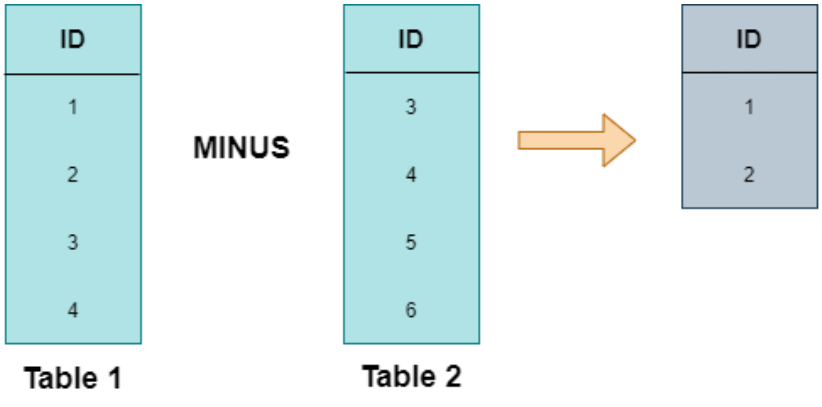

```
SELECT column_list FROM table1
INTERSECT
SELECT column_list FROM table2;
```

```
select * from student
INTERSECT
select * from student2;
```

stu_id	name	email	city
1	Mujahid	abc@gmail.com	Pilibhit
5	Siddiqui	abc4@gmail.com	US Nagar

EXCEPT :-

The EXCEPT operator returns the unique element from the first table/set, which is not found in the second table/set. In other words, it will compare the results of two queries and produces the resultant row from the result set obtained by the first query and not found in the result set obtained by the second query.



```
SELECT column_list FROM table1
EXCEPT
SELECT column_list FROM table2;
```

The following are the rules for a statement that uses the EXCEPT operator:

- The number and order of columns in all the SELECT statements must be the same.
- The data types of the corresponding columns in both SELECT statements must be the same or convertible

```
select * from student
EXCEPT
select * from student2;
```

stu_id	name	email	city
2	Raza	abc1@gmail.com	Aligarh
3	Amaan	abc2@gmail.com	Noida
4	Ahmad	abc3@gmail.com	Bareilly

JOINS

SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are as follows:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- NATURAL JOIN

Table :- orders

order_id	cust_id	ord_date	shipper_id
10308	2	15-09-2011	3
10309	30	16-09-2011	1
10310	41	19-09-2011	2

Table :- customers

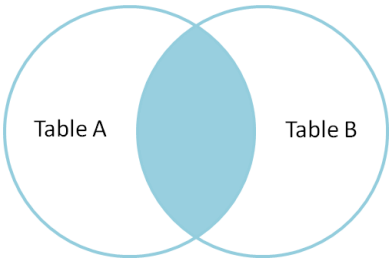
cust_id	cust_name	country
1	Shah	INDIA
2	Rukh	USA
3	Khan	UK

Table :- shippers

ship_id	ship_name
3	abc
1	xyz

INNER JOIN :-

The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.




```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

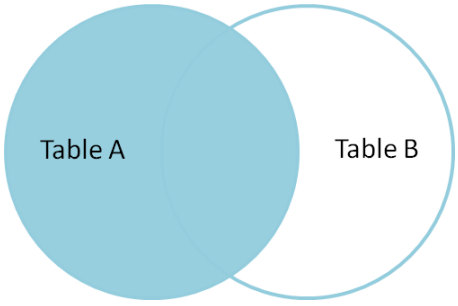
Query :- Get the customer information for each order , if the value of customer id present in the order table

```
select
o.*,c.*
from orders o
inner join customers c
on o.cust_id = c.cust_id;
```

order_id	cust_id	ord_date	shipper_id	cust_id	cust_name	country
10308	2	15-09-2011	3	2	Rukh	USA

LEFT JOIN :-

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.

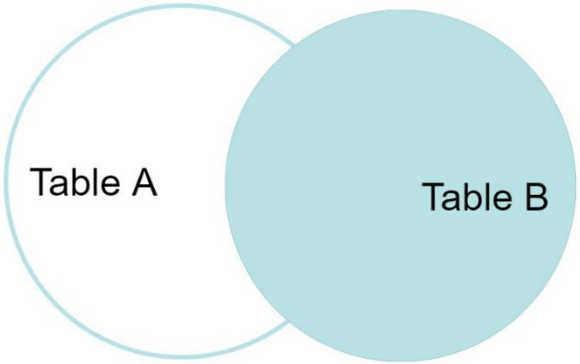



```
select
o.*,c.*
from orders o
left join customers c
on o.cust_id = c.cust_id;
```

order_id	cust_id	ord_date	shipper_id	cust_id	cust_name	country
10308	2	15-09-2011	3	2	Rukh	USA
10309	30	16-09-2011	1	NULL	NULL	NULL
10310	41	19-09-2011	2	NULL	NULL	NULL

RIGHT JOIN :-

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

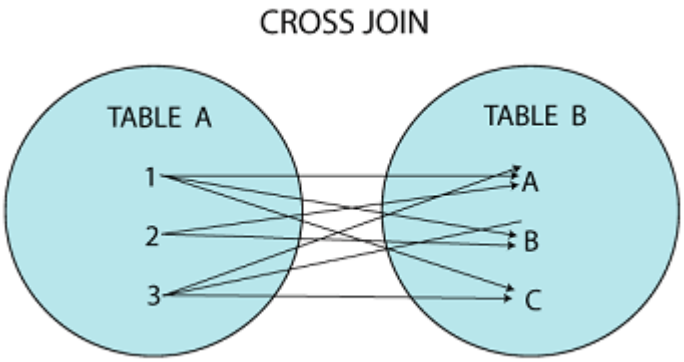



```
select
o.*,c.*
from orders o
Right join customers c
on o.cust_id = c.cust_id;
```

order_id	cust_id	ord_date	shipper_id	cust_id	cust_name	country
NULL	NULL	NULL	NULL	1	Shah	INDIA
10308	2	15-09-2011	3	2	Rukh	USA
NULL	NULL	NULL	NULL	3	Khan	UK

CROSS JOIN :-

MySQL CROSS JOIN is used to combine all possibilities of the two or more tables and returns the result that contains every row from all contributing tables. The CROSS JOIN is also known as CARTESIAN JOIN, which provides the Cartesian product of all associated tables. The Cartesian product can be explained as all rows present in the first table multiplied by all rows present in the second table. It is similar to the Inner Join, where the join condition is not available with this clause.



SELECT *

from orders o

cross join customers;

order_id	cust_id	ord_date	shipper_id	cust_id	cust_name	country
10310	41	19-09-2011	2	1	Shah	INDIA
10309	30	16-09-2011	1	1	Shah	INDIA
10308	2	15-09-2011	3	1	Shah	INDIA
10310	41	19-09-2011	2	2	Rukh	USA
10309	30	16-09-2011	1	2	Rukh	USA
10308	2	15-09-2011	3	2	Rukh	USA
10310	41	19-09-2011	2	3	Khan	UK
10309	30	16-09-2011	1	3	Khan	UK
10308	2	15-09-2011	3	3	Khan	UK

How to join more than 2 dataset.

QUERY:- Get the customer information for each order , if the value of customer id present in the order table. Also get the infornation of shipper name

```
SELECT

o.*, c.*, s.*

from orders o

inner join customers c on o.cust_id = c.cust_id

inner join shippers s on o.shipper_id = s.ship_id;
```

order_id	cust_id	ord_date	shipper_id	cust_id	cust_name	country	ship_id	ship_name
10308	2	15-09-2011	3	2	Rukh	USA	3	abc

Table :- orders

order_id	amount	cust_id
1	100	10
2	500	3
3	300	6
4	800	2
5	350	1

Table :- customers

id	name
1	Danish
2	Zubair
3	Ali
4	Raza

Query :- Get the orders information along with customers full details. if oders amount more than 400.

```
SELECT c.*, o.*
from orders o
inner join customers c on o.cust_id = c.id
where o.amount > 400;

SELECT c.*, o.*
from orders o
inner join customers c on o.cust_id = c.id and o.amount>400;
```

id	name	order_id	amount	cust_id
2	Zubair	4	800	2
3	Ali	2	500	3

Natural Join :-

A natural join is a type of join operation that creates an implicit join by combining tables based on columns with the same name and data type. It is similar to the INNER or LEFT JOIN, but we cannot use the ON or USING clause with natural join as we used in them.

Points to remember :-

- There is no need to specify the column names to join.
- The resultant table always contains unique columns.
- It is possible to perform a natural join on more than two tables.
- Do not use the ON clause.

Syntax:

The following is a basic syntax to illustrate the natural join:

```
SELECT column_names
FROM table_name1
NATURAL JOIN table_name2;
```

Table :- Employee

EMP_ID	EMP_NAME	DEPT_NAME
1	John	HR
2	Mark	IT
3	Naila	IT
4	BSD	SALES

Table :- Department

DEPT_NAME	MANAGER_NAME
IT	Khan
SALES	RAHUL
HR	Wood
FINANCE	Pandy

```
SELECT *
FROM employee
NATURAL JOIN department;
```

DEPT_NAME	EMP_ID	EMP_NAME	MANAGER_NAME
IT	3	Naila	Khan
IT	2	Mark	Khan
SALES	4	BSD	RAHUL
HR	1	John	Wood

SELECT employee.EMP_NAME, department.MANAGER_NAME

FROM employee

NATURAL JOIN department;

EMP_NAME	MANAGER_NAME
Naila	Khan
Mark	Khan
BSD	RAHUL
John	Wood

Difference between Natural Join and Inner Join : -

SN	Natural Join	Inner Join
1.	It joins the tables based on the same column names and their data types.	It joins the tables based on the column name specified in the ON clause explicitly.
2.	It always returns unique columns in the result set.	It returns all the attributes of both tables along with duplicate columns that match the ON clause condition.
3.	If we have not specified any condition in this join, it returns the records based on the common columns.	It returns only those rows that exist in both tables.

SELF JOIN :-

A SELF JOIN is a join that is used to join a table with **itself**. In the previous sections, we have learned about the joining of the table with the other tables using different JOINS, such as INNER, LEFT, RIGHT, and CROSS JOIN. However, there is a need to combine data with other data in the same table itself. In that case, we use Self Join.

We can perform Self Join using **table aliases**. The table aliases allow us not to use the same table name twice with a single statement. If we use the same table name more than one time in a single query without table aliases, it will throw an error.

The table aliases enable us to use the **temporary name** of the table that we are going to use in the query. Let us understand the table aliases with the following explanation.

Suppose we have a table named **"student"** that is going to use twice in the single query.

To aliases the student table, we can write it as:

```
Select ... FROM student AS S1
INNER JOIN student AS S2;
```

Syntax

```
SELECT s1.col_name, s2.col_name...
FROM table1 s1, table1 s2
WHERE s1.common_col_name = s2.common_col_name;
```

TABLE ;- Student

student_id	name	course_id	duration
1	Adam	1	3
2	Peter	2	4
1	Adam	2	4
3	Brian	3	2
2	Shane	3	5

```
SELECT  s1.student_id, s1.name
FROM student AS s1, student s2
WHERE s1.student_id=s2.student_id
AND s1.course_id<>s2.course_id;
```

student_id	name
1	Adam
2	Shane
1	Adam
2	Peter

Partitioning

Partitioning in MySQL is a database management technique used to divide large database tables into smaller, more manageable segments called partitions. Each partition is essentially a separate unit within the table, allowing data to be stored and accessed more efficiently. This can significantly improve the performance of certain types of queries and operations on large datasets.

The main objectives of partitioning are:

1. **Performance optimization :-** By dividing a large table into smaller partitions, it reduces the amount of data that needs to be scanned and processed when executing queries. This can lead to faster query performance and reduced response times.

2. **Manageability :-** Partitioning can make it easier to manage large tables. Operations such as data loading, backups, and maintenance can be performed on individual partitions rather than the entire table, making these tasks more manageable.

3. **Data organization :-** Partitions can be used to organize data based on certain criteria, such as ranges of values (e.g., date ranges) or specific categories. This can be particularly useful for time-series data or data with a clear hierarchical structure.

MySQL has mainly two forms of partitioning:

1. Horizontal Partitioning

This partitioning split the rows of a table into multiple tables based on our logic. In horizontal partitioning, the number of columns is the same in each table, but no need to keep the same number of rows. It physically divides the table but logically treated as a whole. Currently, MySQL supports this partitioning only.

2. Vertical Partitioning

This partitioning splits the table into multiple tables with fewer columns from the original table. It uses an additional table to store the remaining columns. Currently, MySQL does not provide supports for this partitioning.

Benefits of Partitioning

The following are the benefits of partitioning in MySQL:

- It optimizes the query performance. When we query on the table, it scans only the portion of a table that will satisfy the particular statement.
- It is possible to store extensive data in one table that can be held on a single disk or file system partition.
- It provides more control to manage the data in your database.

MySQL supports several types of partitioning methods:

1. **Range Partitioning** :- Data is divided into partitions based on a specified range of column values, such as date ranges or numeric ranges.

2. **List Partitioning :-** Data is divided into partitions based on matching specific values in a designated column.

3. **Hash Partitioning :-** Data is distributed across partitions based on a hash function applied to a column's value. This method can be useful for distributing data evenly across partitions.

4. **Key Partitioning :-** Data is partitioned based on the values of one or more columns that form the table's primary key.

5. **Subpartitioning :-** Each partition can be further divided into subpartitions using any of the above partitioning methods.

```
CREATE TABLE sales (
    customer_id INT AUTO_INCREMENT,
    cust_name VARCHAR(50),
    product_id VARCHAR(10),
    sales_year int,
    amount DECIMAL(10, 2),
    PRIMARY KEY (customer_id)
);
```

```
INSERT INTO sales (cust_name, product_id, sales_year, amount) VALUES
    ('Virat', 'A001', 2019, 342.76),
    ('Rohit', 'A051', 2019, 109.89),
    ('Rahul', 'A065', 2020, 312.65),
    ('Siraj', 'A078', 2020, 89.79),
    ('Ishan', 'A003', 2021, 222.01),
    ('Stokey', 'A333', 2022, 657.98),
    ('Miller', 'A243', 2022, 81.90),
    ('Singh', 'A055', 2021, 167.34),
    ('Root', 'A079', 2020, 814.73);

select * from sales;
```

customer_id	cust_name	product_id	sales_year	amount
1	Virat	A001	2019	342.76
2	Rohit	A051	2019	109.89
3	Rahul	A065	2020	312.65
4	Siraj	A078	2020	89.79
5	Ishan	A003	2021	222.01
6	Stokey	A333	2022	657.98
7	Miller	A243	2022	81.9
8	Singh	A055	2021	167.34
9	Root	A079	2020	814.73

Range Partitioning :-

```
CREATE TABLE sales1 (
    customer_id INT AUTO_INCREMENT,
    cust_name VARCHAR(50),
    product_id VARCHAR(10),
    sales_year int,
    amount DECIMAL(10, 2),
    PRIMARY KEY (customer_id, sales_year)
    )
    partition by range(sales_year) (
    partition p0 values less than (2019),
    partition p1 values less than (2020),
    partition p2 values less than (2021),
    partition p3 values less than (2022),
    partition p4 values less than maxvalue
);
```

select * from **sales** where sales_year = 2020; # Execution time is more

select * from **sales1** where sales_year = 2020; # Execution time is less

customer_id	cust_name	product_id	sales_year	amount
3	Rahul	A065	2020	312.65
4	Siraj	A078	2020	89.79
9	Root	A079	2020	814.73

List Partitioning :-

```
CREATE TABLE sales2 (
   customer_id INT AUTO_INCREMENT,
   cust_name VARCHAR(50),
   product_id VARCHAR(10),
   sales_year int,
   amount DECIMAL(10, 2),
   PRIMARY KEY (customer_id, sales_year)
   )
   partition by LIST (Customer_id) (
   partition p_Hr values IN (1,2,3),
   partition p_Software values IN (4,6,9),
   partition p_IT values IN (5,7,8)
);
INSERT INTO sales2 (cust_name, product_id, sales_year, amount) VALUES
   ('Virat', 'A001', 2019, 342.76),
   ('Rohit', 'A051', 2019, 109.89),
   ('Rahul', 'A065', 2020, 312.65),
   ('Siraj', 'A078', 2020, 89.79),
   ('Ishan', 'A003', 2021, 222.01),
   ('Stokey', 'A333', 2022, 657.98),
   ('Miller', 'A243', 2022, 81.90),
   ('Singh', 'A055', 2021, 167.34),
   ('Root', 'A079', 2020, 814.73);

   select * from sales2;
```

customer_id	cust_name	product_id	sales_year	amount
1	Virat	A001	2019	342.76
2	Rohit	A051	2019	109.89
3	Rahul	A065	2020	312.65
4	Siraj	A078	2020	89.79
6	Stokey	A333	2022	657.98
9	Root	A079	2020	814.73
5	Ishan	A003	2021	222.01
7	Miller	A243	2022	81.9
8	Singh	A055	2021	167.34

Hash Partitioning :-

```
CREATE TABLE sales3 (
   customer_id INT AUTO_INCREMENT,
   cust_name VARCHAR(50),
   product_id VARCHAR(10),
   sales_year int,
   amount DECIMAL(10, 2),
   PRIMARY KEY (customer_id, sales_year)
   )
   partition by HASH (Customer_id)
   partitions 4;
```

The data will be distributed across four partitions based on the hash value of the customer_id. The number of partitions (4 in this case) should be chosen based on the expected data distribution and performance requirements.

Key Partitioning :-

```
CREATE TABLE sales4 (
        id int auto_increment,
   customer_id INT,
   cust_name VARCHAR(50),
   product_id VARCHAR(10),
   sales_year int,
   amount DECIMAL(10, 2),
   PRIMARY KEY (id, customer_id, sales_year)
   )
   partition by KEY (id , Customer_id)
   partitions 6;
```

The data will be divided into six partitions based on the combination of id And customer_id. The key partitioning is useful when the primary key is a composite key, and it can help in distributing the data evenly across partitions.

Subpartitioning :-

It is a composite partitioning that further splits each partition in a partition table.

```
CREATE TABLE sales_table (
   BILL_NO INT,
   sale_date DATE,
```

```
   cust_code VARCHAR(15),
   AMOUNT DECIMAL(8,2),
   PRIMARY KEY (BILL_NO, sale_date)
)
PARTITION BY RANGE(YEAR(sale_date))
SUBPARTITION BY HASH(TO_DAYS(sale_date))
SUBPARTITIONS 4 (
   PARTITION p0 VALUES LESS THAN (1990),
   PARTITION p1 VALUES LESS THAN (2000),
   PARTITION p2 VALUES LESS THAN (2010),
   PARTITION p3 VALUES LESS THAN MAXVALUE
);
SELECT PARTITION_NAME, TABLE_ROWS FROM
INFORMATION_SCHEMA.PARTITIONS WHERE TABLE_NAME='sales_table'
```

PARTITION_NAME	TABLE_ROWS
p0	0
p0	0
p0	0
p0	0
p1	0
p1	0
p1	0
p1	0
p2	0
p2	0
p2	0
p2	0
p3	0
p3	0
p3	0
p3	0

In the above statement -

- The table has 4 RANGE partitions.
- Each of these partitions—p0, p1, p2 and p3—is further divided into 4 subpartitions.
- Therefore the entire table is divided into 4 * 4 = 16 partitions.

Window functions

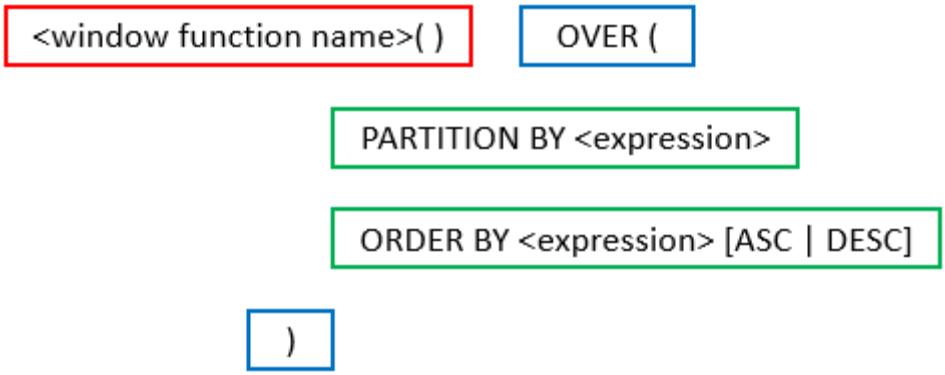
A window function in MySQL used to do a calculation across a set of rows that are related to the **current row**. The current row is that row for which function evaluation occurs. Window functions perform a calculation similar to a calculation done by using the aggregate functions. But, unlike aggregate functions that perform operations on an entire table, window functions do not produce a result to be grouped into one row. It means window functions perform operations on a set of rows and ***produces an aggregated value for each row***. Therefore each row maintains the unique identities.

Window functions applies aggregate and ranking functions over a particular window (set of rows). OVER clause is used with window functions to define that window. OVER clause does two things :

Partition Clause :- This clause is used to **divide or breaks** the rows into partitions, and the partition boundary separates these partitions.

ORDER BY Clause :- This clause is used to specify the order of the rows within a partition.

Partition :- Creating the window
Order by :- To arrange the data within the window.




```
SELECT coulmn_name1,
window_function(cloumn_name2)
OVER([PARTITION BY column_name1] [ORDER BY column_name3]) AS new_column
FROM table_name;
```

window_function	=	any aggregate or ranking function
column_name1	=	column to be selected
coulmn_name2	=	column on which window function is to be applied
column_name3	=	column on whose basis partition of rows is to be done
new_column	=	Name of new column
table_name	=	Name of table

Window Functions:

Name	Description
CUME_DIST ()	Cumulative distribution value
DENSE_RANK ()	Rank of current row within its partition, without gaps
FIRST_VALUE ()	Value of argument from first row of window frame
LAG ()	Value of argument from row lagging current row within partition
LAST_VALUE ()	Value of argument from last row of window frame
LEAD ()	Value of argument from row leading current row within partition
NTH_VALUE ()	Value of argument from N-th row of window frame
NTILE ()	Bucket number of current row within its partition.
PERCENT_RANK ()	Percentage rank value
RANK ()	Rank of current row within its partition, with gaps
ROW_NUMBER ()	Number of current row within its partition

```
create table shop_sales_data
(
sales_date date,
shop_id varchar(5),
sales_amount int
);

insert into shop_sales_data values('2022-02-14', 's1',200);
insert into shop_sales_data values('2022-02-15', 's1',300);
insert into shop_sales_data values('2022-02-14', 's2',600);
insert into shop_sales_data values('2022-02-15', 's3',500);
insert into shop_sales_data values('2022-02-18', 's1',400);
insert into shop_sales_data values('2022-02-17', 's2',250);
insert into shop_sales_data values('2022-02-20', 's3',300);

select * from shop_sales_data;
```

sales_date	shop_id	sales_amount
14-02-2022	s1	200
15-02-2022	s1	300
14-02-2022	s2	600
15-02-2022	s3	500
18-02-2022	s1	400
17-02-2022	s2	250
20-02-2022	s3	300

 # Total sum of sales for each shop using window FUNCTION

 # Working function - sum(), min(), max(), count(), avg()

```
select *, count(*) over
(partition by shop_id) as total_sales_count_by_shops
from shop_sales_data;
```

sales_date	shop_id	sales_amount	total_sales_count_by_shops
14-02-2022	s1	200	3
15-02-2022	s1	300	3
18-02-2022	s1	400	3
14-02-2022	s2	600	2
17-02-2022	s2	250	2
15-02-2022	s3	500	2
20-02-2022	s3	300	2

select shop_id, count(*) as total_sales_count_by_shop from shop_sales_data group by shop_id;

shop_id	total_sales_count_by_shop
s1	3
s2	2
s3	2

 # If we only use order by in over clause

```
select *,
    sum(sales_amount) over
    (order by sales_amount desc) as total_sum_of_sales
from shop_sales_data;
```

sales_date	shop_id	sales_amount	total_sum_of_sales
14-02-2022	s2	600	600
15-02-2022	s3	500	1100
18-02-2022	s1	400	1500
15-02-2022	s1	300	2100
20-02-2022	s3	300	2100
17-02-2022	s2	250	2350
14-02-2022	s1	200	2550

 # If we only use partition by in over clause

```
select *,
    sum(sales_amount) over
    (partition  by shop_id) as total_sum_of_sales
from shop_sales_data;
```

sales_date	shop_id	sales_amount	total_sum_of_sales
14-02-2022	s1	200	900
15-02-2022	s1	300	900
18-02-2022	s1	400	900
14-02-2022	s2	600	850
17-02-2022	s2	250	850
15-02-2022	s3	500	800
20-02-2022	s3	300	800

 # If we use partition by & order by together

```
select *,
     sum(sales_amount) over
     (partition  by shop_id order by sales_amount desc) as total_sum_of_sales
from shop_sales_data;
```

sales_date	shop_id	sales_amount	total_sum_of_sales
18-02-2022	s1	400	400
15-02-2022	s1	300	700
14-02-2022	s1	200	900
14-02-2022	s2	600	600
17-02-2022	s2	250	850
15-02-2022	s3	500	500
20-02-2022	s3	300	800

```
create table sales_data
(
sales_date date,
sales_amount int
);

insert into sales_data values('2022-08-21', 500);
insert into sales_data values('2022-08-22', 600);
insert into sales_data values('2022-08-19', 300);
insert into sales_data values('2022-08-18', 200);
insert into sales_data values('2022-08-25', 800);

select * from sales_data;
```

sales_date	sales_amount
21-08-2022	500
22-08-2022	600
19-08-2022	300
18-08-2022	200
25-08-2022	800

Aggregate Window Function :

Various aggregate functions such as SUM(), COUNT(), AVERAGE(), MAX(), MIN() applied over a particular window (set of rows) are called aggregate window functions.

 # Query -> Calculate the data wise Rolling Sum and Average of sales.

```
select *,
          sum(sales_amount) over(order by sales_date) as rolling_sum,
          avg(sales_amount) over(order by sales_date) as rolling_avg
from sales_data;
```

sales_date	sales_amount	rolling_sum	rolling_avg
18-08-2022	200	200	200
19-08-2022	300	500	250
21-08-2022	500	1000	333.3333
22-08-2022	600	1600	400
25-08-2022	800	2400	480

Ranking Window Functions :

Ranking functions are, RANK(), DENSE_RANK(), ROW_NUMBER()

- **RANK() –**
  As the name suggests, the rank function assigns rank to all the rows within every partition. Rank is assigned such that rank 1 given to the first row and rows having same value are assigned same rank. For the next rank after two same rank values, one rank value will be skipped.

- **DENSE_RANK() –**
  It assigns rank to each row within partition. Just like rank function first row is assigned rank 1 and rows having same value have same rank. The difference between RANK() and DENSE_RANK() is that in DENSE_RANK(), for the next rank after two same rank, consecutive integer is used, no rank is skipped.

- **ROW_NUMBER() –**
  It assigns consecutive integers to all the rows within partition. Within a partition, no two rows can have same row number.

```
insert into shop_sales_data values('2022-02-19', 's1',400);
insert into shop_sales_data values('2022-02-20', 's1',400);
insert into shop_sales_data values('2022-02-22', 's2',300);
insert into shop_sales_data values('2022-02-25', 's1',200);
insert into shop_sales_data values('2022-02-15', 's2',600);
insert into shop_sales_data values('2022-02-16', 's2',600);
insert into shop_sales_data values('2022-02-16', 's3',500);
insert into shop_sales_data values('2022-02-18', 's3',500);
insert into shop_sales_data values('2022-02-19', 's3',300);
```

```
select * from shop_sales_data;
```

sales_date	shop_id	sales_amount
14-02-2022	s1	200
15-02-2022	s1	300
14-02-2022	s2	600
15-02-2022	s3	500
18-02-2022	s1	400
17-02-2022	s2	250
20-02-2022	s3	300
19-02-2022	s1	400
20-02-2022	s1	400
22-02-2022	s2	300

25-02-2022	s1	200
15-02-2022	s2	600
16-02-2022	s2	600
16-02-2022	s3	500
18-02-2022	s3	500
19-02-2022	s3	300

```
select *,
    row_number() over(partition by shop_id order by sales_amount desc) as row_num,
    rank() over(partition by shop_id order by sales_amount desc) as ranl_val,
    DENSE_RANK() over(partition by shop_id order by sales_amount desc) as dense_rank_val
from shop_sales_data;
```

sales_date	shop_id	sales_amount	row_num	ran_val	dense_rank_val
18-02-2022	s1	400	1	1	1
19-02-2022	s1	400	2	1	1
20-02-2022	s1	400	3	1	1
15-02-2022	s1	300	4	4	2
14-02-2022	s1	200	5	5	3
25-02-2022	s1	200	6	5	3
14-02-2022	s2	600	1	1	1
15-02-2022	s2	600	2	1	1
16-02-2022	s2	600	3	1	1
22-02-2022	s2	300	4	4	2
17-02-2022	s2	250	5	5	3
15-02-2022	s3	500	1	1	1
16-02-2022	s3	500	2	1	1
18-02-2022	s3	500	3	1	1
20-02-2022	s3	300	4	4	2
19-02-2022	s3	300	5	4	2

```
create table employees
(
emp_is int,
salary int,
dept_name varchar(50)
);

insert into employees values
(1, 100000, 'software'),
(2, 110000, 'software'),
(3, 110000, 'software'),
(4, 110000, 'software'),
(5, 150000, 'finance'),
(6, 150000, 'finance'),
(7, 120000, 'IT'),
(8, 120000, 'HR'),
(9, 120000, 'HR'),
(10, 110000, 'HR');

select * from employees;
```

emp_is	salary	dept_name
1	100000	software
2	110000	software
3	110000	software
4	110000	software
5	150000	finance
6	150000	finance
7	120000	IT
8	120000	HR
9	120000	HR
10	110000	HR

Query --> Get one employee from each department who is getting maximum salary (employee can be random if salary is same)

```
select
tmp.*
from  (select *,
      row_number() over(partition by dept_name order by salary desc) as row_num
      from employees) tmp
where tmp.row_num = 1;
```

emp_is	salary	dept_name	row_num
5	150000	finance	1
8	120000	HR	1
7	120000	IT	1
2	110000	software	1

Query --> Get all employee from each department who is getting maximum salary

```
SELECT
tmp.*
from  (select *,
      rank() over(partition by dept_name order by salary desc) as rank_num
      from employees) tmp
where tmp.rank_num =1;
```

emp_is	salary	dept_name	rank_num
5	150000	finance	1
6	150000	finance	1
8	120000	HR	1
9	120000	HR	1
7	120000	IT	1
2	110000	software	1
3	110000	software	1
4	110000	software	1

Query --> Get all top 2 ranked employee from each department who is getting maximum salary.

```
SELECT
 tmp.*
from  (select *,
      dense_rank() over(partition by dept_name order by salary desc) as dense_rank_num
      from employees) tmp
where tmp.dense_rank_num <=2;
```

emp_is	salary	dept_name	dense_rank_num
5	150000	finance	1
6	150000	finance	1
8	120000	HR	1
9	120000	HR	1
10	110000	HR	2
7	120000	IT	1
2	110000	software	1
3	110000	software	1
4	110000	software	1
1	100000	software	2

LEAD() and LAG() Function:-

The LEAD() and LAG() function in MySQL are used to get preceding and succeeding value of any row within its partition. These functions are termed as nonaggregate Window functions.

- The LAG() function is used to get value from row that precedes the current row.
- The LEAD() function is used to get value from row that succeeds the current row.

Syntax:

For LEAD() function-

```
LEAD(expr, N, default)
          OVER (Window_specification | Window_name)
```

For LAG() function-

```
LAG(expr, N, default)
          OVER (Window_specification | Window_name)
```

Parameters used:

1. **expr:** It can be a column or any built-in function.
2. **N:** It is a positive value which determine number of rows preceding/succeeding the current row. If it is omitted in query then its default value is 1.
3. **default:** It is the default value return by function in-case no row precedes/succeedes the current row by **N** rows. If it is missing then it is by default NULL.
4. **OVER():** It defines how rows are partitioned into groups. If OVER() is empty then function compute result using all rows.
5. **Window_specification:** It consist of query partition clause which determines how the query rows are partitioned and ordered.
6. **Window_name:** If window is specified elsewhere in the query then it is referenced using this Window_name.

Table:- sales_data.

sales_date	sales_amount

21-08-2022	500
22-08-2022	600
19-08-2022	300
18-08-2022	200
25-08-2022	800

Lag :-

```
select *

    lag(sales_amount, 1) over(order by sales_date) as pre_day_sales
from sales_data;
```

sales_date	sales_amount	pre_day_sales
18-08-2022	200	NULL
19-08-2022	300	200
21-08-2022	500	300
22-08-2022	600	500
25-08-2022	800	600

Lead :-

```
select *,
    lead(sales_amount, 1) over(order by sales_date) as next_day_sales
from sales_data;
```

sales_date	sales_amount	next_day_sales
18-08-2022	200	300
19-08-2022	300	500
21-08-2022	500	600
22-08-2022	600	800
25-08-2022	800	NULL

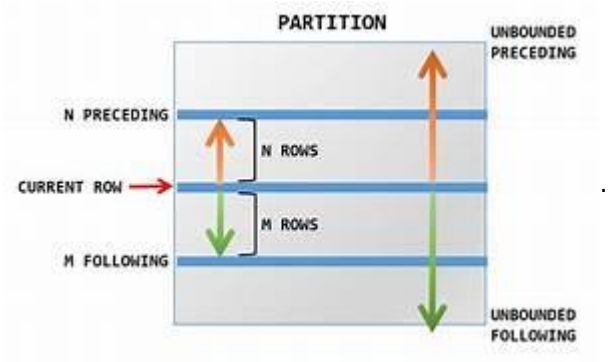
Frame clause in window functions:-

Frames are determined with respect to the current row, which enables a frame to move within a partition depending on the location of the current row within its partition. Examples:

- By defining a frame to be all rows from the partition start to the current row, you can compute running totals for each row.
- By defining a frame as extending ***N*** rows on either side of the current row, you can compute rolling averages

Rows Between :- Define upper bound & lower bound of row to be covered.

Range Between :- Define upper & lower range of value for a column to be covered.



CURRENT ROW :- It specifies the row of the recent calculation

N PRECEDING :- It is the physical N of rows before the first current row.

M FOLLOWING :- It is the physical M of rows after the first current row.

UNBOUNDED PRECEDING :- Here, the frame starts from the first row of a current partition.

UNBOUNDED FOLLOWING :- It specifies the end of the frame at the final row in the partition.

Table:- sales_data.

sales_date	sales_amount
21-08-2022	500
22-08-2022	600
19-08-2022	300
18-08-2022	200
25-08-2022	800

How to use Frame clause - Rows BETWEEN

```
select *,
    sum(sales_amount) over(order by sales_date rows BETWEEN 1 PRECEDING and 1
following) as
    pre_plus_next_sales_sum
from sales_data;
```

sales_date	sales_amount	pre_plus_next_sales_sum
18-08-2022	200	500
19-08-2022	300	1000
21-08-2022	500	1400
22-08-2022	600	1900
25-08-2022	800	1400

```
select *,
    sum(sales_amount) over(order by sales_date rows BETWEEN 1 PRECEDING and current
row) as
     pre_plus_current_sales_sum
from sales_data;
```

sales_date	sales_amount	pre_plus_current_sales_sum
18-08-2022	200	200
19-08-2022	300	500
21-08-2022	500	800
22-08-2022	600	1100
25-08-2022	800	1400

```
select *,
   sum(sales_amount) over(order by sales_date rows BETWEEN current row and 1 following) as
    current_plus_next_sales_sum
from sales_data;
```

sales_date	sales_amount	current_plus_next_sales_sum
18-08-2022	200	500
19-08-2022	300	800
21-08-2022	500	1100
22-08-2022	600	1400
25-08-2022	800	800

```
select *,
    sum(sales_amount) over(order by sales_date rows BETWEEN unbounded PRECEDING and
current
    row) as rolling_sum_from_upper
from sales_data;
```

sales_date	sales_amount	rolling_sum_from_upper
18-08-2022	200	200
19-08-2022	300	500
21-08-2022	500	1000
22-08-2022	600	1600
25-08-2022	800	2400

```
select *,
    sum(sales_amount) over(order by sales_date rows BETWEEN  current row and unbounded
    following) as rolling_sum_from_bottom
from sales_data;
```

sales_date	sales_amount	rolling_sum_from_bottom
18-08-2022	200	2400
19-08-2022	300	2200
21-08-2022	500	1900
22-08-2022	600	1400
25-08-2022	800	800

```
select *,
    sum(sales_amount) over(order by sales_date rows BETWEEN unbounded PRECEDING and
    unbounded following) as rows_sum
from sales_data;
```

sales_date	sales_amount	rows_sum
18-08-2022	200	2400
19-08-2022	300	2400
21-08-2022	500	2400

22-08-2022	600	2400
25-08-2022	800	2400

Range Between:-

Table:- sales_data.

sales_date	sales_amount
21-08-2022	500
22-08-2022	600
19-08-2022	300
18-08-2022	200
25-08-2022	800
20-03-2023	900
23-03-2023	200
25-03-2023	300
29-03-2023	250

Query -> Calculate the running sum for a week

```
select *,
    sum(sales_amount) over(order by sales_date range between interval '6' day preceding and
current
    row) as running_weekly_sum
from sales_data;
```

sales_date	sales_amount	running_weekly_sum
18-08-2022	200	200
19-08-2022	300	500
21-08-2022	500	1000
22-08-2022	600	1600
25-08-2022	800	2200
20-03-2023	900	900
23-03-2023	200	1100
25-03-2023	300	1400
29-03-2023	250	750

CTE (Common Table Expressions)

In MySQL, every statement or query produces a temporary result or relation. A common table expression or CTE is used to **name those temporary results set** that exist within the execution scope of that particular statement, such as CREATE, INSERT, SELECT, UPDATE, DELETE, etc.

- A CTE is defined using **WITH** clause.
- Using WITH clause we can define more than one CTEs in a single statement.
- A CTE can be referenced in the other CTEs that are part of same WITH clause but those CTEs should be defined earlier.
- The scope of every CTE exist within the statement in which it is defined.

```
WITH cte_name (column_names) AS (query)
SELECT * FROM cte_name;
```

CTE is a **subquery** that can be **self-referencing** using its own name. It is also known as **recursive CTE** and can also be referenced multiple times in the same query.

- The recursive CTEs are defined using **WITH RECURSIVE** clause.
- There should be a terminating condition to recursive CTE.
- The recursive CTEs are used for series generation and traversal of hierarchical or tree-structured data.

```
WITH RECURSIVE cte_name (column_names) AS ( subquery )
SELECT * FROM cte_name;
```

The recursive CTE consist of a non-recursive subquery followed by a recursive subquery-

- The first select statement is a non-recursive statement, which provides initial rows for result set.
- UNION [ALL, DISTINCT] is use to add additional rows to previous result set. Use of ALL and DISTINCT keyword are used to include or eliminate duplicate rows in the last result set.
- The second select statement is a recursive statement which produces result set iteratively until the condition provided in WHERE clause is true.
- The result set produced at each iteration take result set produced at previous iteration as the base table.
- The recursion ends when the recursive select statement doesn't produce any additional rows

Table :- employees

emp_id	emp_name	dep_id	salary
1	Mujahid	100	10000
2	Raza	100	20000
3	Ayyub	101	30000
4	Zeeshan	102	40000
5	Ali	101	60000
6	Danish	100	90000
7	Taimoor	101	80000

Table :- department

dept_id	dept_name
100	Software
101	HR
102	IT
103	Finance

Query - Write a query to print the name of department along with the total salary paid in each department.

--- Normal approach

```
select d.dept_name, tmp.total_salary
from (select dept_id, sum(salary) as total_salary from employees group by dept_id) tmp
inner join department d on tmp.dept_id = d.dept_id;
```

dept_name	total_salary
Software	120000
HR	170000
IT	40000

--- How to do it using with clause

```
with dept_wise_salary as (select dept_id, sum(salary) as total_salary from employees group by dept_id)
select * from dept_wise_salary;
```

dept_name	total_salary
Software	120000
HR	170000
IT	40000

Recursive :-

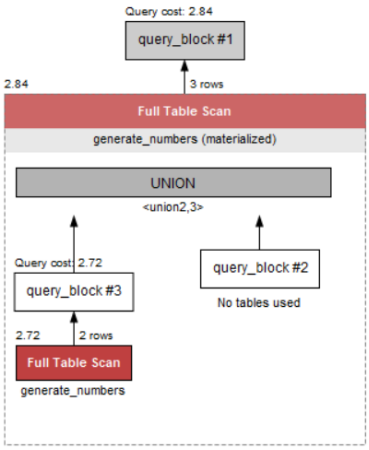
```
with recursive cte(n) as
(select 1 union all select n+1 from cte where n<5)
select * from cte;
```

n
1
2
3
4
5

Query - Write a query to generates numbers from 1 to 10 in SQL

```
with recursive generate_numbers as
(
   select 1 as n
   UNION
   select n+1 from generate_numbers where n<10
)
select * from generate_numbers;
```

n
1
2
3
4
5
6
7
8
9
10




```
with recursive cte  as (
select 1 as n , 1 as p, -1 as q
union all
select n+1, p+2, p+4 from cte where n<5)
select * from cte;
```

n	p	q
1	1	-1
2	3	3
3	5	7
4	7	11
5	9	15

Query - Generates a series of **first five odd numbers.**

```
WITH RECURSIVE
odd_num (id, n) AS
(
SELECT 1, 1
union all
SELECT id+1, n+2 from odd_num where id < 5
)
SELECT * FROM odd_num;
```

id	n
1	1
2	3
3	5
4	7
5	9

Table :- emp_mgr

id	name	manager_id	designation
1	Raza	NULL	CEO
2	Ali	5	SDE
3	Danish	5	DA
4	Zeeshu	5	DS
5	Azam	7	Manager
6	Aamir	7	Architect
7	Anna	1	CTO
8	Zubair	1	Manager

Query - For our CTO 'Anna' present her org chart

```
with RECURSIVE emp_hir as
( SELECT id, name, manager_id, designation from emp_mgr where name = 'Anna'
 UNION
 select em.id, em.name, em.manager_id, em.designation from emp_hir eh inner join
emp_mgr em on eh.id = em.manager_id)
select * from emp_hir;
```

id	name	manager_id	designation
7	Anna	1	CTO
5	Azam	7	Manager
6	Aamir	7	Architect
2	Ali	5	SDE
3	Danish	5	DA
4	Zeeshu	5	DS

Query- Print level of employees as well

```
with RECURSIVE emp_hir as
(
   SELECT id, name, manager_id, designation, 1 as lvl from emp_mgr where name = 'Anna'
   UNION
   select em.id, em.name, em.manager_id, em.designation, eh.levl + 1 as lvl from emp_hir eh
inner join emp_mgr em on eh.id = em.manager_id
)
select * from emp_hir;
```

id	name	manager_id	designation	lvl
7	Anna	1	CTO	1
5	Azam	7	Manager	2
6	Aamir	7	Architect	2
2	Ali	5	SDE	3
3	Danish	5	DA	3
4	Zeeshu	5	DS	3

Benefits of using CTE

- It provides better readability of the query.
- It increases the performance of the query.
- The CTE allows us to use it as an alternative to the VIEW concept
- It can also be used as chaining of CTE for simplifying the query.
- It can also be used to implement recursive queries easily.

Indexes

An index is a data structure that allows us to add indexes in the existing table. It enables you to improve the faster retrieval of records on a database table. It creates an **entry** for each value of the indexed columns. We use it to quickly find the record without searching each row in a database table whenever the table is accessed. We can create an index by using one or more **columns** of the table for efficient access to the records.

When a table is created with a primary key or unique key, it automatically creates a special index named **PRIMARY**. We called this index as a **clustered index**. All indexes other than **PRIMARY** indexes are known as a **non-clustered index** or secondary index.

Need for Indexing

Suppose we have a contact book that contains names and mobile numbers of the user. In this contact book, we want to find the mobile number of Martin Stoke. If the contact book is an unordered format means the name of the contact book is not sorted alphabetically, we need to go over all pages and read every name until we will not find the desired name that we are looking for. This type of searching name is known as sequential searching.

To find the name and contact of the user from table **contactbooks**, generally, we used to execute the following query:

```
SELECT mobile_number FROM contactbooks WHERE first_name = 'Martin' AND last_name = 'Stoke';
```

This query is very simple and easy. Although it finds the phone number and name of the user fast, the database searches entire rows of the table until it will not find the rows that you want. Assume, the contact books table contains **millions** of rows, then, without an index, the data retrieval takes a lot of time to find the result. In that case, the database indexing plays an important role in returning the desired result and improves the overall performance of the query.

CREATE INDEX

Generally, we create an index at the time of table creation in the database. The following statement creates a table with an index that contains two columns col2 and col3.

```
CREATE TABLE t_index(
  col1 INT PRIMARY KEY,
  col2 INT NOT NULL,
  col3 INT NOT NULL,
  col4 VARCHAR(20),
  INDEX (col2,col3)
);
```

If we want to add index in table, we will use the CREATE INDEX statement as follows:

CREATE INDEX [index_name] **ON** [table_name] (**column** names)

When should indexes be created:

- A column contains a wide range of values.
- A column does not contain a large number of null values.
- One or more columns are frequently used together in a where clause or a join condition.

When should indexes be avoided:

- The table is small
- The columns are not often used as a condition in the query
- The column is updated frequently
-

Drop Index

MySQL allows a DROP INDEX statement to remove the existing index from the table. To delete an index from a table, we can use the following query:

DROP INDEX index_name **ON** table_name [algorithm_option | lock_option];

If we want to delete an index, it requires two things:

- First, we have to specify the name of the index that we want to remove.
- Second, name of the table from which your index belongs.

Algorithm Option-

Algorithm [=] {**DEFAULT** | INPLACE | COPY}

COPY:- This algorithm allows us to copy one table into another new table row by row and then DROP Index statement performed on this new table. On this table, we cannot perform an INSERT and UPDATE statement for data manipulation.

INPLACE:- This algorithm allows us to rebuild a table instead of copy the original table. We can perform all data manipulation operations on this table. On this table, MySQL issues an exclusive metadata lock during the index removal.

Lock Option

This clause enables us to control the level of concurrent reads and writes during the index removal.

LOCK [=] {**DEFAULT**|NONE|SHARED|EXCLUSIVE}

SHARED:- This mode supports only concurrent reads, not concurrent writes. When the concurrent reads are not supported, it gives an error.

DEFAULT:- This mode can have the maximum level of concurrency for a specified algorithm. It will enable concurrent reads and writes if supported otherwise enforces exclusive mode.

NONE:- You have concurrent read and write if this mode is supported. Otherwise, it gives an error.

EXCLUSIVE:- This mode enforces exclusive access.

Show Indexes

SHOW INDEXES **FROM** table_name;

SHOW INDEXES IN table_name **FROM** database_name;

OR

SHOW KEYS **FROM** table_name IN database_name;

The SHOW INDEX query returns the following fields/information:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
employees	1	emp_id	1	emp_id	A	0	NULL	NULL	YES	BTREE			YES	NULL

Table: It contains the name of the table.

Non_unique: It returns 1 if the index contains duplicates. Otherwise, it returns 0.

Key_name: It is the name of an index. If the table contains a primary key, the index name is always PRIMARY.

Seq_in_index: It is the sequence number of the column in the index that starts from 1.

Column_name: It contains the name of a column.

Collation: It gives information about how the column is sorted in the index. It contains values where **A** represents ascending, **D** represents descending, and **Null** represents not sorted.

Cardinality: It gives an estimated number of unique values in the index table where the higher cardinality represents a greater chance of using indexes by MySQL.

Sub_part: It is a prefix of the index. It has a NULL value if all the column of the table is indexed. When the column is partially indexed, it will return the number of indexed characters.

Packed: It tells how the key is packed. Otherwise, it returns NULL.

NULL: It contains **blank** if the column does not have NULL value; otherwise, it returns YES.

Index_type: It contains the name of the index method like BTREE, HASH, RTREE, FULLTEXT, etc.

Comment: It contains the index information when they are not described in its column. For example, when the index is disabled, it returns disabled.

Index_column: When you create an index with **comment** attributes, it contains the comment for the specified index.

Visible: It contains YES if the index is visible to the query optimizer, and if not, it contains NO.

Expression:- MySQL 8.0 supports **functional key parts** that affect both **expression** and **column_name** columns. We can understand it more clearly with the below points:

- For functional parts, the expression column represents expression for the key part, and column_name represents NULL.
- For the non-functional part, the expression represents NULL, and column_name represents the column indexed by the key part.

UNIQUE INDEX

Generally, we use the primary key constraint to enforce the uniqueness value of one or more columns. But, we can use only one primary key for each table. So if we want to make multiple sets of columns with unique values, the primary key constraint will not be used.

MySQL allows another constraint called the **UNIQUE INDEX** to enforce the uniqueness of values in one or more columns. We can create more than one UNIQUE index in a single table, which is not possible with the primary key constraint

Unique indexes are used for the maintenance of the integrity of the data present in the table as well as for fast performance, it does not allow multiple values to enter into the table.

```
CREATE UNIQUE INDEX index_name
ON table_name (index_column1, index_column2,...);
```

Clustered VS Non-Clustered Index

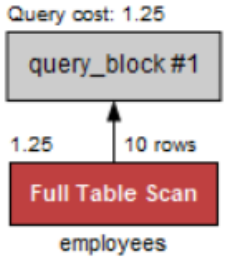
Parameter	Clustered Index	Non-Clustered Index
Definition	A clustered index is a table where the data for the rows are stored. In a relational database, if the table column contains a primary key, MySQL automatically creates a clustered index named **PRIMARY**.	The indexes other than PRIMARY indexes (clustered indexes) called a non-clustered index. The non-clustered indexes are also known as secondary indexes.
Use for	It can be used to sort the record and store the index in physical memory.	It creates a logical ordering of data rows and uses pointers for accessing the physical data files.
Size	Its size is large.	Its size is small in comparison to a clustered index.
Data Accessing	It accesses the data very fast.	It has slower accessing power in comparison to the clustered index.
Storing Method	It stores records in the leaf node of an index.	It does not store records in the leaf node of an index that means it takes extra space for data.
Additional Disk Space	It does not require additional reports.	It requires an additional space to store the index separately.
Type of Key	It uses the primary key as a clustered index.	It can work with unique constraints that act as a composite key.
Contains in Table	A table can only one clustered index.	A table can contain one or more than a non-clustered index.
Index Id	A clustered index always contains an index id of 0.	A non-clustered index always contains an index id>0.

```
create table employees
(
emp_id int,
emp_name varchar(50),
emp_dept varchar(50),
emp_company Varchar(50)
);


insert into employees values
(102, 'mujahid', 'IT', 'ABC'),
(103, 'raza', 'HR', 'tech'),
(104, 'ayyub', 'IT', 'RNA'),
(105, 'amaan', 'Software', 'YWC'),
(106, 'ahmad', 'Hr', 'NFX'),
(107, 'ali', 'software', 'TATA'),
(108, 'zeeshu', 'IT', 'RNA'),
(109, 'danish', 'IT', 'tech'),
(110, 'hiru', 'HR', 'ABC'),
(111, 'mehka', 'IT', 'mahindra');

select * from employees;
```

emp_id	emp_name	emp_dept	emp_company
102	mujahid	IT	ABC
103	raza	HR	tech
104	ayyub	IT	RNA
105	amaan	Software	YWC
106	ahmad	Hr	NFX
107	ali	software	TATA
108	zeeshu	IT	RNA
109	danish	IT	tech
110	hiru	HR	ABC
111	mehka	IT	mahindra

```
select * from employees where emp_id = 106;
```

emp_id	emp_name	emp_dept	emp_company
106	ahmad	Hr	NFX



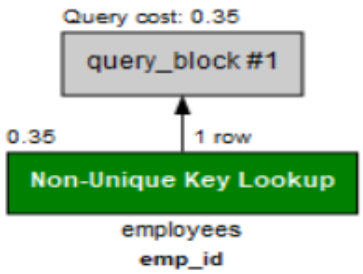
When we use Indexing:-

```
create table employees
(
emp_id int,
emp_name varchar(50),
emp_dept varchar(50),
emp_company Varchar(50),
index(emp_id)
);
```

emp_id	emp_name	emp_dept	emp_company
102	mujahid	IT	ABC
103	raza	HR	tech
104	ayyub	IT	RNA
105	amaan	Software	YWC
106	ahmad	Hr	NFX
107	ali	software	TATA
108	zeeshu	IT	RNA
109	danish	IT	tech
110	hiru	HR	ABC
111	mehka	IT	mahindra

```
select * from employees where emp_id = 106;
```

emp_id	emp_name	emp_dept	emp_company
106	ahmad	Hr	NFX



Views :-

A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own. The View and table have one main difference that the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the View also.

- They don't occupy any space in the memory.

Syntax -

```
CREATE [OR REPLACE] VIEW view_name AS
SELECT columns
FROM tables
[WHERE conditions];
```

Parameters:

OR REPLACE : It is optional. It is used when a VIEW already exists. If you do not specify this clause and the VIEW already exists, the CREATE VIEW statement will return an error.

view_name: It specifies the name of the VIEW that you want to create in MySQL.

WHERE conditions: It is also optional. It specifies the conditions that must be met for the records to be included in the VIEW.

Advantages :-

Simplify complex query :- It allows the user to simplify complex queries. If we are using the complex query, we can create a view based on it to use a simple SELECT statement instead of typing the complex query again.

Increases the Re-usability :- We know that View simplifies the complex queries and converts them into a single line of code to use VIEWS. Such type of code makes it easier to integrate with our application. This will eliminate the chances of repeatedly writing the same formula in every query, making the code reusable and more readable.

Help in Data Security :- It also allows us to show only authorized information to the users and hide essential data like personal and banking information. We can limit which information users can access by authoring only the necessary data to them.

Enable Backward Compatibility :- A view can also enable the backward compatibility in legacy systems. Suppose we want to split a large table into many smaller ones without affecting the current applications that reference the table. In this case, we will create a view with the same name as the real table so that the current applications can reference the view as if it were a table.

```
create table employees
(
emp_id int,
emp_name varchar(20),
mobile BIGINT,
dept_name varchar(20),
salary int
);

insert into employees values
(1, 'Mujahid' , 9838834543 , 'Software' , 10000),
(2, 'Raza' , 4568834576 , 'IT' , 20000),
(3, 'Ayyub' , 8756345432 , 'HR' , 50000),
(4, 'Hasan' , 7687453425 , 'IT' , 40000),
(5, 'Amaan' , 6875644543 , 'Software' , 30000);

select * from employees;
```

emp_id	emp_name	mobile	dept_name	salary
1	Mujahid	9838834543	Software	10000
2	Raza	4568834576	IT	20000
3	Ayyub	8756345432	HR	50000
4	Hasan	7687453425	IT	40000
5	Amaan	6875644543	Software	30000

```
create view employee_data_for_finance as select emp_id, emp_name, salary from employees;

select * from employee_data_for_finance;
```

emp_id	emp_name	salary
1	Mujahid	10000
2	Raza	20000
3	Ayyub	50000
4	Hasan	40000
5	Amaan	30000

Create logic for department wise salary sum.

```
create view dept_wise_salary as select dept_name, sum(salary) from employees group by
dept_name;

select * from dept_wise_salary;
```

dept_name	sum(salary)
Software	40000
IT	60000
HR	50000

```
drop view dept_wise_salary;

create view specify_salary as select emp_id, emp_name, salary from employees where salary > 20000;

select * from specify_salary;
```

emp_id	emp_name	salary
3	Ayyub	50000
4	Hasan	40000
5	Amaan	30000

Creating a view with join :-

Table :- employees

emp_id	emp_name	salary
1	Mujahid	10000
2	Raza	20000
3	Ayyub	50000
4	Hasan	40000
5	Amaan	30000

Table :- other_data

emp_id	email
1	md@gmail.com
2	ra@gmail.com
3	ab@gmail.com
4	hsn@gmail.com
5	amn@gmail.com

```
create view full_details as select
e.emp_name, e.salary, o.email
from other_data o
inner join employees e on o.emp_id = e.emp_id;

select * from full_details;
```

emp_name	salary	email
Mujahid	10000	md@gmail.com
Raza	20000	ra@gmail.com
Ayyub	50000	ab@gmail.com
Hasan	40000	hsn@gmail.com
Amaan	30000	amn@gmail.com

Subquery

In SQL a Subquery can be simply defined as a query within another query. In other words we can say that a Subquery is a query that is embedded in WHERE clause of another SQL query. Important rules for Subqueries:

- Subqueries can be used with SELECT, UPDATE, INSERT, DELETE statements along with expression operator. It could be equality operator or comparison operator such as =, >, =, <= and Like operator.
- If the main query does not have multiple columns for subquery, then a subquery can have only one column in the SELECT command.
- A subquery is a query within another query. The outer query is called as **main query** and inner query is called as **subquery**.
- The subquery generally executes first when the subquery doesn't have any **co-relation** with the **main query**, when there is a co-relation the parser takes the decision **on the fly** on which query to execute on **precedence** and uses the output of the subquery accordingly.
- Subquery must be enclosed in parentheses.
- Subqueries are on the right side of the comparison operator.
- We cannot use the **ORDER BY** clause in a subquery, although it can be used inside the main query.
- Use single-row operators with single row Subqueries. Use multiple-row operators with multiple-row Subqueries.

The following are the advantages of using subqueries:

- The subqueries make the queries in a structured form that allows us to isolate each part of a statement.
- The subqueries provide alternative ways to query the data from the table; otherwise, we need to use complex joins and unions.
- The subqueries are more readable than complex join or union statements.

Table :- employee

id	name	age	hiring_date	salary	city
1	Alex	24	12-08-2021	10000	aligarh
2	Raza	27	12-08-2021	20000	pilibhit
3	Smith	26	17-08-2021	11000	agra
4	Anna	28	17-08-2021	12000	bareilly
5	Maya	29	19-08-2021	17000	delhi

Query :- Write a query to print all those employee records who are getting more salary then "Anna"

 # Wrong way -> select salary from employee where salary > 12000;

```
select * from employee where salary > (select salary from employee
                                       where name = 'Anna');
```

id	name	age	hiring_date	salary	city
2	Raza	27	12-08-2021	20000	pilibhit
5	Maya	29	19-08-2021	17000	delhi

```
select name, city, salary from employee where salary = ( select max(salary)
                                                         from employee);
```

name	city	salary
Raza	pilibhit	20000

Table :- customer_order_data

order_id	cust_id	supplier_id	cust_country
101	200	300	USA
102	201	301	INDIA
103	202	302	USA
104	203	303	UK

Table :- supplier_data

supplier_id	sup_country
300	USA
303	UK

Query :- Write a query to find all customer order data where all customers are from

same countries as tha supplier.

```
select * from customer_order_data where cust_country in (select distinct
                                                    sup_country from supplier_data);
```

order_id	cust_id	supplier_id	cust_country
101	200	300	USA
103	202	302	USA
104	203	303	UK

Function and Procedure

MySQL Functions

Creating a function

The **CREATE FUNCTION** statement is used for creating a stored function and user-defined functions. A stored function is a set of SQL statements that perform some operation and return a single value. The function can be used in SQL queries.

```
CREATE FUNCTION function_name [ (fun_parameter1   fun_parameter2 ..) ]
RETURNS datatype  [ characteristics ]
BEGIN
Declaration_section
Executable_section
END;
```

Parameters used:

1. **function_name :-**

   It is the name by which stored function is called. The name should not be same as native(built_in) function. In order to associate routine explicitly with a specific database function name should be given as *database_name.func_name*.

2. **func_parameter :-**

   It is the argument whose value is used by the function inside its body. You can't specify to these parameters **IN, OUT, INOUT**. The parameter declaration inside parenthesis is provided as *func_parameter type*. Here, type represents a valid Mysql datatype.

3. **Datatype :-**

   It is datatype of value returned by function.

4. **Characteristics :-**

   The CREATE FUNCTION statement is accepted only if at least one of the characteristics { DETERMINISTIC, NO SQL, or READS SQL DATA } is specified in its declaration.

5. ***Func_body*** : - is the set of Mysql statements that perform operation.

   declaration_section : all variables are declared.

   executable_section : code for the function is written here.

   The function body must contain one RETURN statement.

```
create database sales_DB;
use sales_DB;
```

Note :- If we have large dataset file(csv) , in which columns numbers is more so its very difficult to create table manually. So how to insert a table in MySQL, the following steps are required-

- Open Anaconda prompt
- Go to the drive where file id located **(eg. >\D:)**
- Show file **(D:\>dir)**
- **Pip install csvkit**
- **csvsql --dialect mysql --snifflimit 100000 sales.csv > output_sales.sql**
- The output file in generated in the disk drive.
- Open output file in notepad and copy the table schema.

```
CREATE TABLE sales (
    order_id VARCHAR(15) NOT NULL,
    customer_name VARCHAR(16) NOT NULL,
    region VARCHAR(14) NOT NULL,
    sales DECIMAL(38, 0) NOT NULL,
    quantity DECIMAL(38, 0) NOT NULL,
    discount DECIMAL(38, 2) NOT NULL,
    profit DECIMAL(38, 4) NOT NULL
      );
```

```
set session sql_mode = ' '

load data infile
'D:\\sales.csv'
into table sales
fields terminated by ','
enclosed by '"'
lines terminated by '\n'
ignore 1 rows;
```

order_id	customer_name	region	sales	quantity	discount	profit
IN-2011-47883	Annie Thurman	Africa	408	2	0	106.14
IN-2011-47883	Eugene Moren	Oceania	120	3	0.1	36.036
CA-2011-1510	Joseph Holt	EMEA	66	4	0	29.64
IN-2011-79397	Joseph Holt	North	45	3	0.5	-26.055
ID-2011-80230	Magdelene	Oceania	114	5	0.1	37.77
IZ-2011-4680	Kean Nguyen	Oceania	55	2	0.1	15.342
IN-2011-65159	Ken Lonsdale	Canada	314	1	0	3.12
IN-2011-65159	Lindsay Williams	Oceania	276	1	0.1	110.412
ES-2011-4869686	Larry Blacks	Oceania	912	4	0.4	-319.464
IN-2011-33652	Larry Blacks	EMEA	667	4	0	253.32
ID-2011-80230	Dorothy	Southeast Asia	338	3	0.45	-122.801
MX-2011-160234	Dennis Pardue	Southeast Asia	211	1	0.55	-70.3995
IR-2011-770	Ken Lonsdale	North	854	7	0	290.43
ID-2011-80230	Stewart Visinsky	Southeast Asia	193	1	0	50.13
ID-2011-80230	Jas O'Carroll	Oceania	159	2	0.4	-95.676
ID-2011-12596	Ken Lonsdale	Central	195	4	0	44.88
IN-2011-79397	Ken Lonsdale	EMEA	123	2	0	42.9
IR-2011-7690	Chris McAfee	Oceania	69	2	0.4	3.42
IR-2011-770	Kean Nguyen	Oceania	69	2	0.4	-26.412
TZ-2011-7370	Nat Gilpin	Southeast Asia	135	2	0.47	-45.9018
IZ-2011-4680	Jas O'Carroll	Oceania	36	3	0.1	4.743

Query :-

```
DELIMITER $$
CREATE FUNCTION add_to_col(a int)
returns int
deterministic
begin
    declare b int;
    set b = a + 10;
    return b;
end $$
DELIMITER ;

select add_to_col(15);
```

add_to_col(15)
25

```
select quantity, add_to_col(quantity) from sales;
```

quantity	add_to_col(quantity)
2	12
3	13
4	14
3	13
5	15
2	12
.....	.....

Query :-

```
DELIMITER $$
CREATE FUNCTION final_profit(profit int, discount int)
returns int
deterministic
begin
   declare final_profit int ;
   set final_profit = profit - discount ;
   return final_profit;
end $$
delimiter ;

select profit, discount, final_profit(profit, discount) from sales;
```

profit	discount	final_profit(profit, discount)
106.14	0	106
36.036	0.1	36
29.64	0	30
-26.055	0.5	-27
37.77	0.1	38
15.342	0.1	15
….	…..	…..

Query :-

```
DELIMITER $$
CREATE FUNCTION final_profit_real(profit decimal(20,6), discount decimal(20,6), sales
decimal(20,6))
returns int
deterministic
begin
    declare final_profit int ;
    set final_profit = profit - sales * discount ;
    return final_profit;
end $$
DELIMITER ;

select profit, discount, sales, final_profit_real(profit , discount, sales) from sales;
```

profit	discount	sales	final_profit_real(profit , discount, sales)
106.14	0	408	106
36.036	0.1	120	24
29.64	0	66	30
-26.055	0.5	45	-49
37.77	0.1	114	26
15.342	0.1	55	10
3.12	0	314	3
….	…..	….	…….

Query :-

```
DELIMITER $$
CREATE FUNCTION int_to_str(a int)
returns varchar(30)
deterministic
begin
   declare b varchar(30) ;
   set b = a ;
   return b ;
end $$
DELIMITER ;

select max(sales), min(sales) from sales;
```

max(sales)	min(sales)
912	36

```
 select quantity, int_to_str(quantity) from sales;
```

quantity	int_to_str(quantity)
2	2
3	3
4	4
3	3
5	5
2	2
....	.....

Query :-

```
if sales 1 - 100 = Super affordable product
if sales 100 - 300 = Affordable
if sales 300 - 600 = Moderate
if sales 600 above = Expensive

DELIMITER $$
create function mark_sales(sales int)
returns varchar(30)
deterministic
begin
   declare flag_sales varchar(30);
```

```
    if sales <= 100 then
            set flag_sales = "Super affordable product";
    elseif sales >= 100 and sales <= 300 then
            set flag_sales = "Affordable";
    elseif sales >= 300 and sales <= 600 then
            set flag_sales = "Moderate" ;
    else
            set flag_sales = "Expensive" ;
end if ;
return flag_sales;
end $$
DELIMITER ;

select mark_sales(30);
```

mark_sales(30)
Super affordable product

```
select mark_sales(430);
```

mark_sales(430)
Moderate

```
 select sales, mark_sales(sales) from sales;
```

sales	mark_sales(sales)
408	Moderate
120	Affordable
66	Super affordable product
45	Super affordable product
114	Affordable
55	Super affordable product
314	Moderate
276	Affordable
912	Expensive
667	Expensive
...	.....

MySQL Stored Procedure

A procedure (often called a stored procedure) is a **collection of pre-compiled SQL statements** stored inside the database. It is a subroutine or a subprogram in the regular computing language. **A procedure always contains a name, parameter lists, and SQL statements**.

Stored procedures are prepared SQL code that you save so you can reuse it over and over again. So if you have an SQL query that you write over and over again, save it as a stored procedure and call it to run it. You can also pass parameters to stored procedures so that the stored procedure can act on the passed parameter values.

Stored Procedures are created to perform one or more **DML** operations on Database. It is nothing but the group of SQL statements that accepts some input in the form of parameters and performs some task and may or may not return a value.

A procedure is called a **recursive stored procedure** when it calls itself. Most database systems support recursive stored procedures. But, it is not supported well in MySQL.

Stored Procedure Features

- Stored Procedure increases the performance of the applications. Once stored procedures are created, they are compiled and stored in the database.
- Stored procedure reduces the traffic between application and database server. Because the application has to send only the stored procedure's name and parameters instead of sending multiple SQL statements.
- Stored procedures are reusable and transparent to any applications.
- A procedure is always secure. The database administrator can grant permissions to applications that access stored procedures in the database without giving any permissions on the database tables.

```
DELIMITER &&
CREATE PROCEDURE procedure_name [[IN | OUT | INOUT] parameter_name datatype [, parameter datatype]) ]
BEGIN
   Declaration_section
   Executable_section
END &&
DELIMITER ;
CALL procedure_name ( parameter(s))
```

Parameter Explanations

Parameter Name	Descriptions
procedure_name	It represents the name of the stored procedure.
parameter	It represents the number of parameters. It can be one or more than one.
Declaration_section	It represents the declarations of all variables.
Executable_section	It represents the code for the function execution.

MySQL procedure parameter has one of three modes:

IN parameter

It is the default mode. It takes a parameter as input, such as an attribute. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected.

OUT parameters

It is used to pass a parameter as output. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.

INOUT parameters

It is a combination of IN and OUT parameters. It means the calling program can pass the argument, and the procedure can modify the INOUT parameter, and then passes the new value back to the calling program.

We can use the **CALL statement** to call a stored procedure. This statement returns the values to its caller through its parameters (IN, OUT, or INOUT).

Difference between Function and Procedure

Function

In a programming language, function is said to be a set of instructions that take some input and execute some tasks. A function can either be predefined or user-defined.

The function contains the set of programming statements enclosed by {}. It provides code reusability and modularity to the program.

Procedure

The procedure is an important and frequently used programming construct for a compiler. It is used to generate good code for procedure calls and returns.

It does not deal with an expression. It is defined as the set of commands that are executed in order.

Based on	Function	Procedure
Basic	It calculates the result based on given inputs.	It performs certain tasks in order.
SQL Query	It can be called in a query.	It cannot be called in a query.
try-catch	It does not support try-catch blocks.	It supports try-catch blocks.
SELECT	There can be a function call in a SELECT statement.	There is no procedure call in a SELECT statement.
Return	It returns the resulting value or control to the calling function or code.	On the other hand, it returns control but does not return a value to the code or calling function.
DML statements	DML statements such as (**Insert, Delete,** and **Update)** cannot be used in a function.	In a procedure, DML statements can be used.
Call	We can call a function via the procedure.	Whereas it is not possible to call a procedure via function.
Expression	It has to deal with an expression.	It does not have to deal with an expression.
Compilation	Functions are compiled whenever they are called.	Procedures need to be compiled once, and if necessary, we can call

		them repeatedly without compiling them every time.
Explicit transaction	There is no explicit transaction handling in a function.	It can use explicit transaction handling.

Query :-

```
create table loop_table(var int)

DELIMITER $$
create procedure insert_data()
begin
      set @var = 10;
  generate_data : loop
  insert into loop_table values(@var);
  set @var = @var + 1;
  if @var = 100 then
       leave generate_data;
  end if;
       end loop generate_data;
end $$
DELIMITER ;

call insert_data()

select * from loop_table;
```

var
10
11
12
13
14
15
.....

Table :- student_info

stud_id	stud_roll	stud_name	subject	marks	phone
1	101	mark	Physics	78	9827546328
2	102	wood	Physics	67	6745546328
3	103	rahul	Maths	88	3457546328
4	104	shami	Maths	73	9857646328
5	105	allen	Biology	71	2827546988
6	106	stoke	Physics	80	4888546328
7	107	smith	Language	60	8797876756
8	108	kane	English	76	9997546328
9	109	alex	Hindi	88	8976543428
10	110	john	Biology	98	904546328

IN parameter example

The following example creates a stored procedure that finds all student_info according to subject specified by the input parameter subjectName.

```
DELIMITER &&
CREATE PROCEDURE student_records( IN subjectName varchar(30) )
BEGIN
   SELECT * FROM student_info WHERE `subject` = subjectName;
END &&
DELIMITER ;
```

In this example, the **subjectName** is the IN parameter of the stored procedure.

Suppose that you want to find students enrolled in the Physics, you need to pass an argument **('Physics')** to the stored procedure as shown in the following query:

```
call student_records('Physics');
```

stud_id	stud_roll	stud_name	subject	marks	phone
1	101	mark	Physics	78	9827546328
2	102	wood	Physics	67	6745546328
6	106	stoke	Physics	80	4888546328

OUT parameter example

Now, with the help of the following query, we will create a stored procedure with OUT parameter which will count the total of a particular subject by providing the subject name as the parameter.

```
DELIMITER &&
CREATE PROCEDURE subjects (
                    IN stud_subject varchar(30),
                    out total int )
BEGIN
   SELECT COUNT(subject) into total FROM student_info
   where subject = stud_subject;
END &&
DELIMITER ;
```

'stud-Subject' is the IN parameter that is the number of subjects we want to count and 'total' is the OUT parameter that stores the number of subjects for a particular subject.

```
call subjects('Maths', @total);
select @total;
```

@total
2

INOUT parameter example

```
DELIMITER &&
CREATE PROCEDURE counter ( INOUT count int,
                           IN increment int)
BEGIN
   set count = count + increment;
END &&
DELIMITER ;
```

Here, 'count' is the INOUT parameter, which can store and return values and 'increment' is the IN parameter, which accepts the values from user.

```
set @counter =0;

call counter(@counter ,1);
select @counter;
```

@counter
1

```
call counter(@counter ,7);
select @counter;
```

@counter
8

Trigger

Trigger is a statement that a system executes automatically when there is any modification to the database.

A trigger in MySQL is a set of SQL statements that reside in a system catalog. **It is a special type of stored procedure that is invoked automatically in response to an event**. Each trigger is associated with a table, which is activated on any DML statement such as **INSERT, UPDATE**, or **DELETE**.

A trigger is called a special procedure because it cannot be called directly like a stored procedure. The main difference between the trigger and procedure is that a trigger is called automatically when a data modification event is made against a table. In contrast, a stored procedure must be called explicitly.

Generally, **triggers are of two types** according to the SQL standard: row-level triggers and statement-level triggers.

Row-Level Trigger: It is a trigger, which is activated for each row by a triggering statement such as insert, update, or delete. For example, if a table has inserted, updated, or deleted multiple rows, the row trigger is fired automatically for each row affected by the insert, update, or delete statement.

Statement-Level Trigger: It is a trigger, which is fired once for each event that occurs on a table regardless of how many rows are inserted, updated, or deleted.

Why we need/use triggers in MySQL

We need/use triggers in MySQL due to the following features:

- Triggers help us to enforce business rules.
- Triggers help us to validate data even before they are inserted or updated.
- Triggers help us to keep a log of records like maintaining audit trails in tables.
- SQL triggers provide an alternative way to check the integrity of data.
- Triggers provide an alternative way to run the scheduled task.
- Triggers increases the performance of SQL queries because it does not need to compile each time the query is executed.
- Triggers reduce the client-side code that saves time and effort.
- Triggers help us to scale our application across different platforms.
- Triggers are easy to maintain.

Types of Triggers in MySQL

We can define the maximum six types of actions or events in the form of triggers:

1. **Before Insert:** It is activated before the insertion of data into the table.
2. **After Insert:** It is activated after the insertion of data into the table.
3. **Before Update:** It is activated before the update of data in the table.
4. **After Update:** It is activated after the update of the data in the table.
5. **Before Delete:** It is activated before the data is removed from the table.
6. **After Delete:** It is activated after the deletion of data from the table.

> When we use a statement that does not use INSERT, UPDATE or DELETE query to change the data in a table, the triggers associated with the trigger will not be invoked.

Benefits of using triggers in business:

- Faster application development. Because the database stores triggers, you do not have to code the trigger actions into each database application.
- Global enforcement of business rules. Define a trigger once and then reuse it for any application that uses the database.
- Easier maintenance. If a business policy changes, you need to change only the corresponding trigger program instead of each application program.
- Improve performance in client/server environment. All rules run on the server before the result returns.

Limitations of Using Triggers in MySQL

- MySQL triggers do not allow to use of all validations; they only provide extended validations. **For example**, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
- Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
- Triggers may increase the overhead of the database server.

Naming Conventions

Naming conventions are the set of rules that we follow to give appropriate unique names. It saves our time to keep the work organize and understandable. Therefore, **we must use a unique name for each trigger associated with a table**. However, it is a good practice to have the same trigger name defined for different tables.

The following naming convention should be used to name the trigger in MySQL:

(BEFOR | AFTER) table_name (INSERT | UPDATE | DELETE)

Thus,

Trigger Activation Time: BEFORE | AFTER

Trigger Event: INSERT | UPDATE | DELETE

Syntax : -

```
Delimiter //
CREATE TRIGGER trigger_name
(AFTER | BEFORE) (INSERT | UPDATE | DELETE)
ON table_name FOR EACH ROW
    BEGIN
     --variable declarations
     --trigger code
     END;  //
```

We assume that you are habituated with "MySQL Stored Procedures". You can use the following statements of MySQL procedure in triggers:

- Compound statements (BEGIN / END)
- Variable declaration (DECLARE) and assignment (SET)
- Flow-of-control statements (IF, CASE, WHILE, LOOP, WHILE, REPEAT, LEAVE, ITERATE)
- Condition declarations
- Handler declarations

Trigger	Event	Table	Statement	Timing	Created	sql_mode	Definer	character_set_	collation_conne	Database Collation

before_insert_empworkinghou	INSERT	employee	BEGIN
			IF NEW.working_hours < 0 THEN SET
			NEW.working_hours = 0;
			END IF;
			END
				BEFORE	2020-11-13
					14:49:05.83
						STRICT_TRANS_TABLES,NC	root@localhost	cp850	cp850_general_	utf8mb4_0900

The show trigger statement contains several columns in the result set. Let us explain each column in detail.

- **Trigger :-** It is the name of the trigger that we want to create and must be unique within the schema.
- **Event :-** It is the type of operation name that invokes the trigger. It can be either INSERT, UPDATE, or DELETE operation.
- **Table :-** It is the name of the table to which the trigger belongs.
- **Statement :-** It is the body of the trigger that contains logic for the trigger when it activates.
- **Timing :-** It is the activation time of the trigger, either BEFORE or AFTER. It indicates that the trigger will be invoked before or after each row of modifications occurs on the table.
- **Created :-** It represents the time and date when the trigger is created.
- **sql_mode :-** It displays the SQL_MODE when the trigger is executed.
- **Definer :-** It is the name of a user account that created the trigger and should be in the 'user_name'@'host_name' format.
- **Character_set_client :-** It was the session value of the character_set_client system variable when the trigger was created.
- **Collation_connection :-** It was the session value of the character_set_client system variable when the trigger was created.
- **Database Collation :-** It determines the rules that compare and order the character string. It is the collation of the database with which the trigger belongs.

```
create table course(
course_id int,
course_desc varchar(50),
course_mentor varchar(50),
course_price int,
course_discount int,
create_date date
);

delimiter //
create trigger course_before_insert
 before insert
 on course for each row
  begin
  set new.create_date = sysdate();
  end; //

  insert into course (course_id, course_desc, course_mentor, course_price, course_discount)
values
  (101, 'DA', 'MJ', 5000, 1000),
  (104, 'DS', 'HR', 5500, 700),
  (178, 'BD', 'HS', 7000, 2500);

 select * from course;
```

course_id	course_desc	course_mentor	course_price	course_discount	create_date
101	DA	MJ	5000	1000	28-08-2023
104	DS	HR	5500	700	28-08-2023
178	BD	HS	7000	2500	28-08-2023

```
create table course1(
course_id int,
course_desc varchar(50),
course_mentor varchar(50),
course_price int,
course_discount int,
create_date date,
user_info varchar(50)
);
```

```
delimiter //
create trigger course_before_insert1
 before insert
 on course1 for each row
  begin
  declare user_val varchar(50);
  set new.create_date = sysdate();
  select user() into user_val;
  set new.user_info = user_val;
  end; //

  insert into course1 (course_id, course_desc, course_mentor, course_price, course_discount)
values
  (101, 'DA', 'MJ', 5000, 1000),
  (104, 'DS', 'HR', 5500, 700),
  (178, 'BD', 'HS', 7000, 2500);

  select * from course1;
```

course_id	course_desc	course_mentor	course_price	course_discount	create_date	user_info
101	DA	MJ	5000	1000	28-08-2023	root@localhost
104	DS	HR	5500	700	28-08-2023	root@localhost
178	BD	HS	7000	2500	28-08-2023	root@localhost

```
  create table course2(
  course_id int,
  course_desc varchar(50),
  course_mentor varchar(50),
  course_price int,
  course_discount int,
  create_date date,
  user_info varchar(50)
  );

   create table ref_course(
   record_insert_date date,
   record_insert_user varchar(50)
   );
```

```
delimiter //
create trigger course_before_insert2
 before insert
 on course2 for each row
  begin
  declare user_val varchar(50);
  set new.create_date = sysdate();
  select user() into user_val;
  set new.user_info = user_val;
  insert into ref_course values(sysdate(), user_val);
  end; //

 insert into course2 (course_id, course_desc, course_mentor, course_price, course_discount)
values
  (101, 'DA', 'MJ', 5000, 1000),
  (104, 'DS', 'HR', 5500, 700),
  (178, 'BD', 'HS', 7000, 2500);

 select * from ref_course;
```

record_insert_date	record_insert_user
28-08-2023	root@localhost
28-08-2023	root@localhost
28-08-2023	root@localhost

```
create table test1
 (
 c1 varchar(50),
 c2 date,
 c3 int
 );

  create table test2
 (
 c1 varchar(50),
 c2 date,
 c3 int
 );
```

```
 create table test3
(
c1 varchar(50),
c2 date,
c3 int
);

insert into test1 values
('John', sysdate(), 109803);

delimiter //
create trigger to_update_others
before insert
on test1 for each row
begin
    insert into test2 values('MRA', sysdate(), 78786);
    insert into test3 values('MRA', sysdate(), 78786);
end; //
```

```
select * from test1;
```

c1	c2	c3
John	28-08-2023	109803
John	28-08-2023	109803

```
select * from test2;
```

c1	c2	c3
MRA	28-08-2023	78786

```
select * from test3;
```

c1	c2	c3
MRA	28-08-2023	78786

Update and delete :-

```
delimiter //
 create trigger to_update_others_table
 before insert
 on test1 for each row
```

```
begin
    update test2 set c1 = 'abc' where c1 = 'MRA';
    delete from test3 where c1 = 'MRA';
end; //
insert into test1 values('wood' ,sysdate(), 7728);
 select * from test1;
select * from test2;
select * from test3;
```

AFTER DELETE

The AFTER DELETE Trigger in MySQL is invoked automatically whenever a delete event is fired on the table.

Restrictions

- We can access the OLD rows but cannot update them in the AFTER DELETE trigger.
- We cannot access the NEW rows. It is because there are no NEW row exists.
- We cannot create an AFTER DELETE trigger on a VIEW.

```
delimiter //
 create trigger after_delete_trigger
 after delete
 on test1 for each row
 begin
     insert into test3 values('After_delete', sysdate(), 57547);
 end; //
 select * from test1;
 delete from test1 where c1 = 'wood';
 select * from test1;
 select * from test3;
```

c1	c2	c3
After_delete	28-08-2023	57547

BEFORE DELETE

BEFORE DELETE Trigger in MySQL is invoked automatically whenever a delete operation is fired on the table.

```
delimiter //
 create trigger before_delete_trigger
 before delete
 on test1 for each row
 begin
      insert into test3 values('After_delete', sysdate(), 57547);
 end; //
  delete from test1 where c1 = 'john';
  select * from test3;
```

Observation :- Before Delete

```
create table test11
 (
 c1 varchar(50),
 c2 date,
 c3 int
 );

   create table test12
 (
 c1 varchar(50),
 c2 date,
 c3 int
 );

 delimiter //
 create trigger before_delete_trigger_observation1
 before delete
 on test11 for each row
 begin
      insert into test12(c1,c2,c3) values(old.c1,old.c2, old.c3);
 end; //
 insert into test11 values('Aligarh' ,sysdate(), 5346);
```

```
insert into test11 values('Delhi' ,sysdate(), 3528);
insert into test11 values('Agra' ,sysdate(), 7756);
insert into test11 values('Noida' ,sysdate(), 8928);


select * from test11;
```

c1	c2	c3
Aligarh	28-08-2023	5346
Delhi	28-08-2023	3528
Agra	28-08-2023	7756
Noida	28-08-2023	8928

```
delete from test11 where c1 = 'Agra';

select * from test12;
```

c1	c2	c3
Agra	28-08-2023	7756

Observation :- After Delete

```
delimiter //
 create trigger after_delete_trigger_observation2
 after delete
 on test11 for each row
 begin
     insert into test12(c1,c2,c3) values(old.c1,old.c2, old.c3);
 end; //

 delete from test11 where c1 = 'Agra';
 select * from test12;
```

c1	c2	c3
Agra	28-08-2023	7756

```
 delete from test11 where c1 = 'Aligarh';
 select * from test12;
```

c1	c2	c3
Agra	28-08-2023	7756
Aligarh	28-08-2023	5346
Aligarh	28-08-2023	5346

AFTER UPDATE TRIGGER

The AFTER UPDATE trigger in MySQL is invoked automatically whenever an UPDATE event is fired on the table associated with the triggers.

```
delimiter //
create trigger after_update_trigger_
after update
on test11 for each row
begin
    insert into test12(c1,c2,c3) values(old.c1,old.c2, old.c3);
end; //
```

```
select * from test11;
```

c1	c2	c3
Delhi	28-08-2023	3528
Noida	28-08-2023	8928

```
update test11 set c1 = 'New York' where c1 = 'Delhi';
```

```
select * from test11;
```

c1	c2	c3
New York	28-08-2023	3528
Noida	28-08-2023	8928

```
select * from test12;
```

c1	c2	c3
Agra	28-08-2023	7756
Aligarh	28-08-2023	5346
Aligarh	28-08-2023	5346

BEFORE UPDATE

BEFORE UPDATE Trigger in MySQL is invoked automatically whenever an update operation is fired on the table associated with the trigger

```
delimiter //
 create trigger before_update_trigger_
 before update
 on test11 for each row
 begin
     insert into test12(c1,c2,c3) values(new.c1,new.c2, new.c3);
 end; //

 select * from test11;

 update test11 set c1 = 'Rome' where c1 = 'Noida';


select * from test11;
```

c1	c2	c3
New York	28-08-2023	3528
Rome	28-08-2023	8928

```
select * from test12;
```

c1	c2	c3
Agra	28-08-2023	7756
Aligarh	28-08-2023	5346
Aligarh	28-08-2023	5346
Rome	28-08-2023	8928

CASE Statement

CASE statement in MySQL is used to find a value by passing over conditions whenever any condition satisfies the given statement otherwise it returns the statement in an else part. However, when a condition is satisfied it stops reading further and returns the output.

MySQL CASE statement is a part of the control flow function that provides us to write an **if-else or if-then-else** logic to a query. This expression can be used anywhere that uses a valid program or query, such as SELECT, WHERE, ORDER BY clause, etc.

Features:

- This returns the statement in the else part if none of the stated conditions are true.
- If there is no ELSE part and no conditions are true, it returns NULL.
- This function comes under Advanced Functions.
- This accepts two parameters namely conditions and results.

CASE Syntax :-

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END;
```

Return Type

The CASE expression returns the result depending on the context where it is used. For example:

- If it is used in the string context, it returns the string result.
- If it is used in a numeric context, it returns the integer, float, decimal value

We can use the CASE statement in two ways, which are as follows:

1. Simple CASE statement:

The first method is to take a value and matches it with the given statement, as shown below.

Syntax

```
CASE value
  WHEN [compare_value] THEN result
  [WHEN [compare_value] THEN result ...]
  [ELSE result]
END
```

It returns the result when the first **compare_value** comparison becomes true. Otherwise, it will return the else clause.

2. Searched CASE statement:

The second method is to consider a **search_condition** in the **WHEN** clauses, and if it finds, return the result in the corresponding THEN clause. Otherwise, it will return the else clause. If else clause is not specified, it will return a NULL value.

Syntax

```
CASE
  WHEN [condition] THEN result
  [WHEN [condition] THEN result ...]
  [ELSE result]
END
```

Some important points about CASE statements:

1. There should always be a SELECT in the case statement.
2. END. ELSE is an optional component but WHEN THEN these cases must be included in the CASE statement.
3. We can make any conditional statement using any conditional operator (like WHERE) between WHEN and THEN. This includes stringing together multiple conditional statements using AND and OR.
4. We can include multiple WHEN statements and an ELSE statement to counter with unaddressed conditions.

Prob :-

```
CREATE TABLE Customer(
   CustomerID INT PRIMARY KEY,
   CustomerName VARCHAR(50),
   LastName VARCHAR(50),
   Country VARCHAR(50),
   Age int,
  Phone bigint
);
```

CustomerID	CustomerName	LastName	Country	Age	Phone
1	Almaas	Raza	India	15	9089765434
2	Jonh	Albert	UK	23	9467257845
3	Shubham	Thakur	India	32	9045248569
4	Naveen	Tulasi	India	25	9089764785
5	Mark	Wood	UK	32	2154765434
6	Kane	Neil	USA	34	1459765454
7	Stom	Neig	France	45	3214569875

Adding Multiple Conditions to a CASE statement

```
select customerName, Age,
case
    when age > 32 then 'The Age is greater than 32'
    when Age = 32 then 'The Age is 32'
    else 'The Age is below 30'
end as Age_cat
from custome
```

customerName	Age	Age_cat
Almaas	15	The Age is below 30
Jonh	23	The Age is below 30
Shubham	32	The Age is 32
Naveen	25	The Age is below 30
Mark	32	The Age is 32
Kane	34	The Age is greater than 32
Stom	45	The Age is greater than 32

CASE Statement With ORDER BY Clause

```
select CustomerName, Country
from customer
order by
(case
    when Country = 'India' then Country
    else Country
end);
```

CustomerName	Country
Stom	France
Almaas	India
Shubham	India
Naveen	India
Jonh	UK
Mark	UK
Kane	USA

Prob :-

```
create table student(
Stud_id int,
Stud_Name varchar(50),
Class varchar(10),
Age int
);
```

Stud_id	Stud_Name	Class	Age
101	Khan	CS	22
121	Khan	CS	22
111	Kumar	ME	24
184	Ghoni	CV	19
151	Wood	EE	26
167	Peter	ME	20

```
select Stud_id, Stud_Name,
case class
    when 'CS' then 'Computer Science'
    when 'CV' then 'Civil'
    when 'EE' then 'Electrical'
    else 'Mechanical'
end as department
from student;
```

Stud_id	Stud_Name	department
101	Khan	Computer Science
121	Khan	Computer Science
111	Kumar	Mechanical
184	Ghoni	Civil
151	Wood	Electrical
167	Peter	Mechanical

Prob :-

```
create table course_detail(
course_name varchar(50),
course_id int,
course_title varchar(10),
course_fee int,
course_launch_year int
);
```

course_name	course_id	course_title	course_fee	course_launch_year
Computer Science	101	CS	20000	2002
Electrical Engineering	104	EE	30000	2001
Mechanical Engineering	100	ME	50000	2000
Civil Engineering	107	CV	20000	2005
CVS	111	DS	20000	2014
NLP	161	DS	29000	2022
nlpe	176	NL	90000	2012
fsda	198	DA	27000	2020
Wdpt	191	WD	70000	2018

```
select * ,
case
    when course_name = 'Mechanical Engineering' then 'This is my course'
    else 'This is not my course'
end as statement
from course_detail;
```

course_name	course_id	course_title	course_fee	course_launch_year	statement
Computer Science	101	CS	20000	2002	This is not my course
Electrical Engineering	104	EE	30000	2001	This is not my course
Mechanical Engineering	100	ME	50000	2000	This is my course
Civil Engineering	107	CV	20000	2005	This is not my course
CVS	111	DS	20000	2014	This is not my course
NLP	161	DS	29000	2022	This is not my course
nlpe	176	NL	90000	2012	This is not my course
fsda	198	DA	27000	2020	This is not my course
Wdpt	191	WD	70000	2018	This is not my course

```
select * ,
case
    when length(course_name) = '4' then 'len 4'
    when length(course_name) = '3' then 'len 3'
    else 'other length'
end as length_desc
from course_detail;
```

course_name	course_id	course_title	course_fee	course_launch_year	length_desc
Computer Science	101	CS	20000	2002	other length
Electrical Engineering	104	EE	30000	2001	other length
Mechanical Engineering	100	ME	50000	2000	other length
Civil Engineering	107	CV	20000	2005	other length
CVS	111	DS	20000	2014	len 3
NLP	161	DS	29000	2022	len 3
nlpe	176	NL	90000	2012	len 4
fsda	198	DA	27000	2020	len 4
Wdpt	191	WD	70000	2018	len 4

```
UPDATE course_detail
SET course_name = CASE
   WHEN course_name = 'NLP' THEN 'Natural Language Processing'
   WHEN course_name = 'fsda' THEN 'Full Stack Data Analytics'
END
WHERE course_name IN ('NLP', 'fsda');
select * from course_detail;
```

course_name	course_id	course_title	course_fee	course_launch_year
Computer Science	101	CS	20000	2002
Electrical Engineering	104	EE	30000	2001
Mechanical Engineering	100	ME	50000	2000
Civil Engineering	107	CV	20000	2005
CVS	111	DS	20000	2014
Natural Language Processing	161	DS	29000	2022
nlpe	176	NL	90000	2012
Full Stack Data Analytics	198	DA	27000	2020
Wdpt	191	WD	70000	2018

Date Functions

The most difficult part when working with dates is to be sure that the format of the date you are trying to insert, matches the format of the date column in the database.

As long as your data contains only the date portion, your queries will work as expected. However, if a time portion is involved, it gets complicated.

Before talking about the complications of querying for dates, we will look at the most important built-infunctions for working with dates.

DATE() in MySQL

The **DATE()** function is used to extract the date part from a DateTime expression. This function is an inbuilt function in MySQL. The DATE() function in MySQL can be used to know the date for a given date or a DateTime. The DATE() function takes a date value as an argument and returns the date. The date argument represents the valid date or DateTime.

Syntax :

select date('Expression');

- **Parameter :** It accepts only one parameter.
- **Expression :** It represents the date/datetime value.
- **Return Value :** It returns the date and returns NULL if the expression is not a date.

MySQL Date Functions

The following table lists the most important built-in date functions in MySQL:

Function	Description
NOW()	Returns the current date and time
CURDATE()	Returns the current date
CURTIME()	Returns the current time
DATE()	Extracts the date part of a date or date/time expression
EXTRACT()	Returns a single part of a date/time

DATE_ADD()	Adds a specified time interval to a date
DATE_SUB()	Subtracts a specified time interval from a date
DATEDIFF()	Returns the number of days between two dates
DATE_FORMAT()	Displays date/time data in different formats

SQL Server Date Functions

The following table lists the most important built-in date functions in SQL Server:

Function	Description
GETDATE()	Returns the current date and time
DATEPART()	Returns a single part of a date/time
DATEADD()	Adds or subtracts a specified time interval from a date
DATEDIFF()	Returns the time between two dates
CONVERT()	Displays date/time data in different formats

SQL Date Data Types

MySQL comes with the following data types for storing a date or a date/time value in the database:

- DATE - format YYYY-MM-DD
- DATETIME - format: YYYY-MM-DD HH:MM:SS
- TIMESTAMP - format: YYYY-MM-DD HH:MM:SS
- YEAR - format YYYY or YY

SQL Server comes with the following data types for storing a date or a date/time value in the database:

- DATE - format YYYY-MM-DD
- DATETIME - format: YYYY-MM-DD HH:MM:SS
- SMALLDATETIME - format: YYYY-MM-DD HH:MM:SS
- TIMESTAMP - format: a unique number

```
create table orders (

order_id int,
product_name varchar(30),
```

```
orderDate date
);

insert into orders values
(101, 'Sweety Supari', '2022-12-05'),
(102, 'Chutki', '2022-12-15'),
(103, 'Gagan', '2022-12-22'),
(101, 'Pass-Pass', '2022-12-25');

select * from orders;
```

order_id	product_name	orderDate
101	Sweety Supari	05-12-2022
102	Chutki	15-12-2022
103	Gagan	22-12-2022
101	Pass-Pass	25-12-2022

Query :- Now you want to select the record with an Orderdate of "2022-12-15" from the above table.

```
select * from orders where orderDate = '2022-12-15';
```

order_id	product_name	orderDate
102	Chutki	15-12-2022

```
create table orders (

order_id int,
product_name varchar(30),
orderDate datetime
);

insert into orders values
(101, 'Sweety Supari', '2022-12-05 13:23:44'),
(102, 'Chutki', '2022-12-15 15:45:21'),
(103, 'Gagan', '2022-12-05 11:34:01'),
(101, 'Pass-Pass', '2022-12-25 09:32:43');
select * from orders;
```

order_id	product_name	orderDate
101	Sweety Supari	05-12-2022 13:23
102	Chutki	15-12-2022 15:45

103	Gagan	05-12-2022 11:34
101	Pass-Pass	25-12-2022 09:32

select Now(), curdate(),curtime();

Now()	curdate()	curtime()
25-09-2023 15:51	25-09-2023	15:51:41

DATE() :-

Extracts the date part of a date or date/time expression.

select product_name, date(orderDate) from orders;

product_name	date(orderDate)
Sweety Supari	05-12-2022
Chutki	15-12-2022
Gagan	05-12-2022
Pass-Pass	25-12-2022

EXTRACT() :-

Returns a single part of a date/time.

Syntax –

EXTRACT(unit FROM date);

Several units can be considered but only some are used such as **MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, etc.** And 'date' is a valid date expression.

Unit Value
MICROSECOND
SECOND
MINUTE
HOUR
DAY
WEEK
MONTH
QUARTER
YEAR
SECOND_MICROSECOND
MINUTE_MICROSECOND
MINUTE_SECOND
HOUR_MICROSECOND
HOUR_SECOND
62
HOUR_MINUTE
DAY_MICROSECOND
DAY_SECOND

DAY_MINUTE
DAY_HOUR
YEAR_MONTH

select product_name, Extract(Day from orderDate) as orderDay from orders;

product_name	orderDay
Sweety Supari	5
Chutki	15
Gagan	5
Pass-Pass	25

select product_name, Extract(SECOND from orderDate) as orderSecond from orders;

product_name	orderSecond
Sweety Supari	44
Chutki	21
Gagan	1
Pass-Pass	43

DATE_ADD() :-

Adds a specified time interval to a date.

Syntax:

DATE_ADD(date, INTERVAL expr type);

Where, date – valid date expression, and expr is the number of intervals we want to add. and type can be one of the following: MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, etc.

select product_name, date_add(orderDate, interval 2 year) as orderDateModified from orders;

product_name	orderDateModified
Sweety Supari	05-12-2024 13:23
Chutki	15-12-2024 15:45
Gagan	05-12-2024 11:34
Pass-Pass	25-12-2024 09:32

DATE_SUB() :-

Subtracts a specified time interval from a date. The syntax for DATE_SUB is the same as DATE_ADD just the difference is that DATE_SUB is used to subtract a given interval of date.

select order_id, date_sub(orderDate , interval 5 day) as subDate from orders;

order_id	subDate
101	30-11-2022 13:23
102	10-12-2022 15:45
103	30-11-2022 11:34
101	20-12-2022 09:32

DATEDIFF() :-

Returns the number of days between two dates.

Syntax:

DATEDIFF(date1, date2);

date1 & date2- date/time expression

select datediff('2022-12-05', '2022-12-25') as datediff;

datediff
-20

DATE_FORMAT() :-
Displays date/time data in different formats.

Syntax:
DATE_FORMAT(date,format);

The date is a valid date and the format specifies the output format for the date/time.
The formats that can be used are:

%a	Abbreviated weekday name (Sun-Sat)
%b	Abbreviated month name (Jan-Dec)
%c	Month, numeric (0-12)

%D	Day of month with English suffix (0th, 1st, 2nd, 3rd)
%d	Day of the month, numeric (00-31)
%e	Day of the month, numeric (0-31)
%f	Microseconds (000000-999999)
%H	Hour (00-23)
%h	Hour (01-12)
%I	Hour (01-12)
%i	Minutes, numeric (00-59)
%j	Day of the year (001-366)
%k	Hour (0-23)
%l	Hour (1-12)
%M	Month name (January-December)
%m	Month, numeric (00-12)
%p	AM or PM
%r	Time, 12-hour (hh:mm: ss followed by AM or PM)
%S	Seconds (00-59)
%s	Seconds (00-59)
%T	Time, 24-hour (hh:mm: ss)
%U	Week (00-53) where Sunday is the first day of the week
%u	Week (00-53) where Monday is the first day of the week
%V	Week (01-53) where Sunday is the first day of the week, used with %X
%v	Week (01-53) where Monday is the first day of the week, used with %x
%W	Weekday name (Sunday-Saturday)
%w	Day of the week (0=Sunday, 6=Saturday)
%X	Year for the week where Sunday is the first day of the week, four digits, used with %V
%x	Year for the week where Monday is the first day of the week, four digits, used with %v
%Y	Year, numeric, four digits
%y	Year, numeric, two digits

Query :
Getting a formatted year as “2020” from the specified date “2020-11-23”.
SELECT DATE_FORMAT("2020-11-23", "%Y");

Output :
2020

Query :
Getting a formatted month name as “November” from the specified date “2020-11-23”.
SELECT DATE_FORMAT("2020-11-23", "%M");

Output :
November

Query :
Getting a day of the month as a numeric value as “23rd” from the specified date “2020-11-23”.
SELECT DATE_FORMAT("2020-11-23", "%D");

Output :
23rd

Query :
Getting month day and year as “November 23 2020” from the specified date “2020-11-23”.
SELECT DATE_FORMAT("2020-11-23", "%M %d %Y");

Output :
November 23 2020

Query :
Getting hour and minute as “12 09” from the specified date and time “2020-11-23 12:09:23”.
SELECT DATE_FORMAT("2020-11-23 12:09:23", "%H %i");

Output :
12 09

DATEPART() :-

The **DATEPART()** function is used to return a single part of a date/time, such as year, month, day, hour, minute, etc.

Syntax

DATEPART(datepart,date)

Where date is a valid date expression and datepart can be one of the following:

datepart	Abbreviation
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw, w
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns

The SQL **datepart function** extracts parts of a datetime datatype for filtering or aggregating table fields in a SQL database. However**, it is not directly available in MySQL**. There are two ways to achieve a similar result as datepart in MySQL.

The Datepart Function Alternatives in MySQL

1. Extract Parts of a Datetime With the **Extract(datetime-part FROM datetime)** Method in MySQL
2. Extract Parts of a Datetime With the **datetime-part(datetime)** Method in MySQL

Normalization

Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table. A functional dependency is denoted by an arrow (→). If an attribute A functionally determines B, then it is written as A → B.

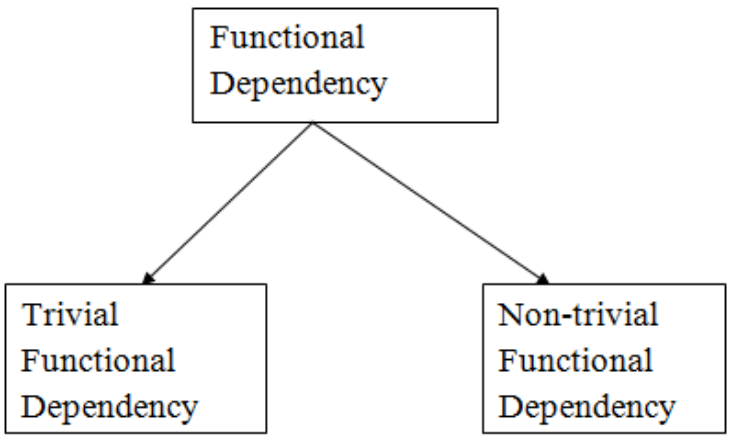
For example, **employee_id → name** means employee_id functionally determines the name of the employee

What does functionally dependent mean?

A function dependency A → B means for all instances of a particular value of A, there is the same value of B. For example in the below table A → B is true, but B → A is not true as there are different values of A for B = 3.

A	B
1	3
2	3
4	0
1	3
4	0

Types of Functional dependency



Trivial Functional Dependency

A → B has trivial functional dependency if B is a subset of A.

Example-

ABC → AB

ABC → A

ABC → ABC

Non - Trivial Functional Dependency

A → B has a non-trivial functional dependency if B is not a subset of A.

When A intersection B is NULL, then A → B is called as complete non-trivial.

Example:

ID → Name,
Name → DOB

Advantages of Functional Dependency-

- The database's data quality is maintained using it.
- It communicates the database design's facts.
- It aids in precisely outlining the limitations and implications of databases.
- It is useful to recognize poor designs.
- Finding the potential keys in the relationship is the first step in the normalization procedure. Identifying potential keys and normalizing the database without functional dependencies is impossible.

Normalization :-

A large database defined as a single relation may result in data duplication. This repetition of data may result in:

- Making relations very large.

- It isn't easy to maintain and update data as it would involve searching many records in relation.
- Wastage and poor utilization of disk space and resources.
- The likelihood of errors and inconsistencies increases.

So to handle these problems, we should analyze and decompose the relations with redundant data into smaller, simpler, and well-structured relations that are satisfy desirable properties. Normalization is a process of decomposing the relations into relations with fewer attributes.

What is Normalization?

- Normalization is the process of organizing the attributes of the database to reduce or eliminate **data redundancy (having the same data but at different places)**.
- It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- Normalization divides the larger table into smaller and links them using relationships.
- The normal form is used to reduce redundancy from the database table.

Data modification anomalies can be categorized into three types:

- **Insertion Anomaly:** Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
- **Deletion Anomaly:** The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
- **Updatation Anomaly:** The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

The features of database normalization are as follows:

Elimination of Data Redundancy: One of the main features of normalization is to eliminate the data redundancy that can occur in a database. Data redundancy refers to the repetition of data in different parts of the database. Normalization helps in reducing or eliminating this redundancy, which can improve the efficiency and consistency of the database.

Ensuring Data Consistency: Normalization helps in ensuring that the data in the database is consistent and accurate. By eliminating redundancy, normalization helps in

preventing inconsistencies and contradictions that can arise due to different versions of the same data.

Simplification of Data Management: Normalization simplifies the process of managing data in a database. By breaking down a complex data structure into simpler tables, normalization makes it easier to manage the data, update it, and retrieve it.

Improved Database Design: Normalization helps in improving the overall design of the database. By organizing the data in a structured and systematic way, normalization makes it easier to design and maintain the database. It also makes the database more flexible and adaptable to changing business needs.

Avoiding Update Anomalies: Normalization helps in avoiding update anomalies, which can occur when updating a single record in a table affects multiple records in other tables. Normalization ensures that each table contains only one type of data and that the relationships between the tables are clearly defined, which helps in avoiding such anomalies.

Standardization: Normalization helps in standardizing the data in the database. By organizing the data into tables and defining relationships between them, normalization helps in ensuring that the data is stored in a consistent and uniform manner.
Normalization is an important process in database design that helps in improving the efficiency, consistency, and accuracy of the database.

Advantages of Normalization

- Normalization helps to minimize data redundancy.
- Greater overall database organization.
- Data consistency within the database.
- Much more flexible database design.
- Enforces the concept of relational integrity.

Disadvantages of Normalization

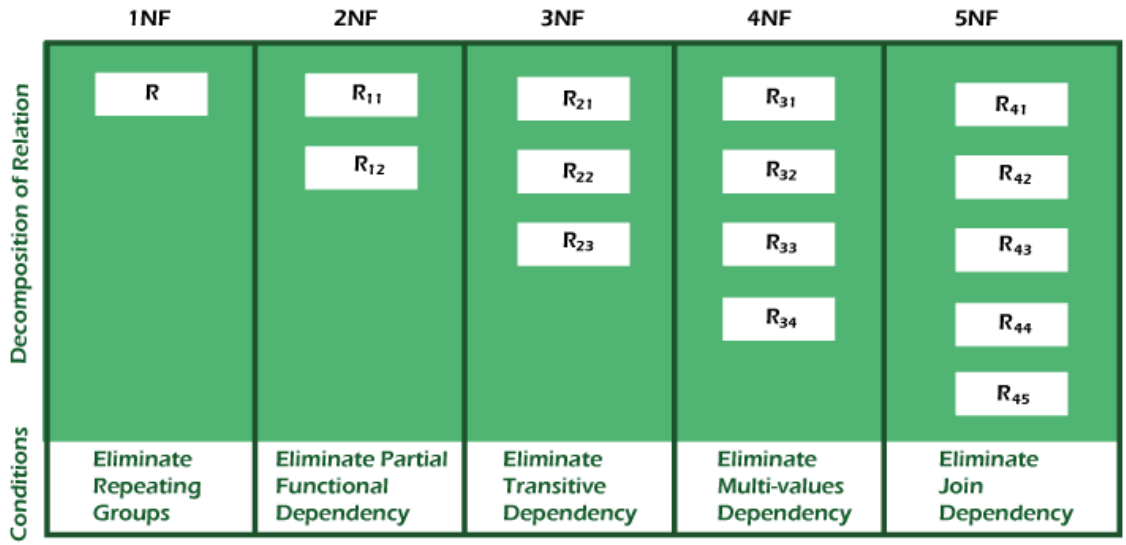
- You cannot start building the database before knowing what the user needs.
- The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
- It is very time-consuming and difficult to normalize relations of a higher degree.

- Careless decomposition may lead to a bad database design, leading to serious problems.

Types of Normal Forms:

Normalization works through a series of stages called Normal forms. The normal forms apply to individual relations. The relation is said to be in particular normal form if it satisfies constraints.

Following are the various types of Normal forms:



Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
BCNF	A stronger definition of 3NF is known as Boyce Codd's normal form.
4NF	A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency.
5NF	A relation is in 5NF. If it is in 4NF and does not contain any join dependency, joining should be lossless.

First Normal Form (1NF) :-

If a relation contain composite or multi-valued attribute, it violates first normal form or a relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is **singled valued attribute**.

- All the record in the table should be unique.
- Each table cell should contain only one value.

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

Second Normal Form (2NF) :-

- In the 2NF, relational must be in 1NF.
- In the second normal form, all non-key attributes are fully functional dependent on the primary key

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Math
83	Computer

Third Normal Form (3NF) :-

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
- 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

Example:

EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

{EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....

Candidate key: {EMP_ID}

Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Boyce Codd normal form (BCNF) :-

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Example: Let's assume there is a company where employees work in more than one department.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

EMP_ID → EMP_COUNTRY

EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

Fourth normal form (4NF) :-

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.

- For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

Fifth normal form (5NF):-

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.