**What is SQL?**

SQL (Structured Query Language) is a standard Database language designed for managing and manipulating relational databases. It allows users to create, retrieve, update, and delete data within a database. SQL is highly versatile and used across different database systems, including MySQL, PostgreSQL, Oracle, SQL Server, and SQLite.

SQL works with tables, where data is stored in rows and columns. Its syntax is simple and English-like, making it easy to understand and use.

### What is a Database?

A **database** is an organized collection of structured information or data, typically stored electronically in a computer system. It enables efficient data management, retrieval, and manipulation. Databases are designed to handle large amounts of data while ensuring data integrity and security.

**Purpose of SQL**

The primary purpose of SQL is to interact with databases to perform various operations like:

1. **Data Querying**: Retrieve specific data using SELECT statements.
2. **Data Manipulation**: Insert, update, delete, and alter data in a database.
3. **Data Definition**: Create or modify the structure of database objects such as tables, indexes, and views.
4. **Access Control**: Manage access permissions and ensure data security using SQL commands.
5. **Transaction Control**: Manage transactions and ensure the integrity of data during multiple operations.

**Who Developed SQL and When Was It Developed?**

SQL (Structured Query Language) was developed in the early 1970s by **IBM** researchers **Donald D. Chamberlin** and **Raymond F. Boyce**. They were part of the team working on IBM’s pioneering database project known as **System R**, which was designed to implement a relational database management system (RDBMS).

SQL was initially called **SEQUEL** (Structured English Query Language), which was inspired by the **relational model of data** proposed by **Edgar F. Codd** in 1970. Codd’s work on the relational model revolutionized the way databases were designed and managed, enabling the structured organization of data using tables and relationships. SEQUEL was later renamed SQL due to trademark issues.

**Timeline of SQL Development:**

* **1970**: Edgar F. Codd publishes a paper introducing the relational model of data.
* **1973–1974**: IBM researchers Chamberlin and Boyce develop SEQUEL, which was later renamed SQL.
* **1979**: Oracle Corporation (then known as Relational Software Inc.) releases the first commercial SQL-based RDBMS, Oracle V2.
* **1986**: SQL becomes a standard when the **American National Standards Institute (ANSI)** adopts SQL as the standard relational database query language.
* **1987**: SQL is adopted as an international standard by the **International Organization for Standardization (ISO)**.

**Who Should Learn SQL?**

* **Developers**: Both backend and full-stack developers need SQL to interact with databases in web applications.
* **Data Analysts**: SQL is essential for querying data and extracting insights for business intelligence.
* **Database Administrators**: They need SQL to manage databases, ensure data integrity, and optimize performance.
* **Data Scientists**: SQL is often used to clean and prepare large datasets for analysis.
* **System Administrators**: For managing database servers and tuning performance.

In general, anyone who works with data or systems involving relational databases will benefit from learning SQL.

### ****What is DBMS (Database Management System)?****

A **Database Management System (DBMS)** is software that allows users to **create, manage, and interact with databases**. It provides a systematic and organized way to store, retrieve, update, and manage data. A DBMS ensures that the data is consistently organized and remains easily accessible.

The DBMS acts as an intermediary between the user and the database, ensuring that the data is stored safely, can be retrieved efficiently, and can be manipulated as needed by various applications. It also provides security, data integrity, and backup/recovery features.

#### **Key Functions of a DBMS:**

1. **Data Definition**: Helps define the structure of the data (schema) and the relationships between different data entities.
2. **Data Manipulation**: Allows users to query, update, and delete data from the database.
3. **Data Security**: Ensures that only authorized users can access or modify the database.
4. **Backup and Recovery**: Ensures data is safe from accidental loss or system failures, and provides a way to restore it.
5. **Concurrency Control**: Manages simultaneous data access to ensure that multiple users can interact with the database without conflicts.
6. **Data Integrity**: Ensures that the data remains accurate and consistent throughout its lifecycle.

### ****Types of DBMS****

DBMS can be categorized into various types based on the data models they use and the architecture they follow:

#### 1. **Hierarchical DBMS**

A **Hierarchical DBMS** organizes data in a **tree-like structure**, where each record has a single parent but can have multiple children. This model is good for representing hierarchical relationships like an organizational structure or a file system.

* **Example**: IBM's Information Management System (IMS).

**Advantages**:

* Simple and fast for hierarchical data.
* Efficient for one-to-many relationships.

**Disadvantages**:

* Limited flexibility (difficult to restructure or extend the hierarchy).
* Requires knowledge of the hierarchical path to access the data.

#### 2. **Network DBMS**

A **Network DBMS** organizes data in a **graph structure**, allowing each record to have multiple parent and child records (many-to-many relationships). This model is more flexible than the hierarchical model.

* **Example**: Integrated Data Store (IDS).

**Advantages**:

* Efficient for complex relationships.
* Supports many-to-many relationships.

**Disadvantages**:

* Complex structure makes it difficult to manage.
* Requires specialized knowledge for querying.

#### 3. **Relational DBMS (RDBMS)**

A **Relational DBMS** organizes data into **tables (relations)** that are made up of rows and columns. It uses **SQL (Structured Query Language)** to manage and query data. Data in relational databases is highly structured, and relationships between tables can be created using foreign keys.

* **Examples**: MySQL, PostgreSQL, Oracle, Microsoft SQL Server.

**Advantages**:

* Simple and intuitive structure (tables).
* Supports powerful querying using SQL.
* Enforces data integrity with primary and foreign keys.
* ACID compliance ensures data reliability.

**Disadvantages**:

* Performance can be affected when scaling up with large datasets.
* Not ideal for unstructured data (like documents, images, etc.).

#### 4. **Object-Oriented DBMS (OODBMS)**

An **Object-Oriented DBMS** stores data in the form of **objects**, similar to how data is handled in object-oriented programming languages like Java or C++. Each object can contain data (attributes) and methods (operations).

* **Examples**: ObjectDB, db4o.

**Advantages**:

* Suitable for applications that use complex data structures (e.g., multimedia, engineering, etc.).
* Better integration with object-oriented programming languages.

**Disadvantages**:

* Slower performance compared to RDBMS for simple queries.
* Less widespread than relational databases, resulting in fewer tools and support.

#### 5. **Document-Oriented DBMS (NoSQL)**

A **Document-Oriented DBMS** stores, retrieves, and manages data as **documents**, usually in formats like **JSON** or **XML**. It is a type of **NoSQL** database that is highly flexible and can handle unstructured or semi-structured data.

* **Examples**: MongoDB, CouchDB.

**Advantages**:

* Schema-less design, which allows for flexible data models.
* Handles unstructured data well (e.g., JSON documents).
* Scales easily horizontally across distributed servers.

**Disadvantages**:

* Lacks the strict consistency and ACID properties of RDBMS.
* Less suited for applications that need structured data and complex relationships.

#### 6. **Key-Value Stores (NoSQL)**

A **Key-Value Store DBMS** is a **simple NoSQL database** that stores data as **key-value pairs**. The key is used as a unique identifier, and the value can be any type of data.

* **Examples**: Redis, Amazon DynamoDB, Riak.

**Advantages**:

* Fast and simple.
* Highly scalable for handling large volumes of simple data.

**Disadvantages**:

* Limited querying capabilities (compared to SQL databases).
* Not suitable for complex data relationships.

#### 7. **Column-Oriented DBMS (NoSQL)**

A **Column-Oriented DBMS** stores data in columns rather than rows. This model is efficient for handling large amounts of data, especially for analytical queries where specific columns are queried frequently.

* **Examples**: Apache Cassandra, HBase.

**Advantages**:

* Efficient for read-heavy operations and analytics.
* Scales horizontally across distributed clusters.

**Disadvantages**:

* Less efficient for transactional data and writes.
* Complex to set up and maintain.

#### 8. **Graph DBMS**

A **Graph DBMS** stores data in the form of **nodes, edges, and properties**, making it ideal for representing relationships and connections. It is useful in applications where data is highly interconnected, such as social networks.

* **Examples**: Neo4j, Amazon Neptune.

**Advantages**:

* Excellent for handling complex relationships.
* Allows for fast querying of paths and connections in large datasets.

**Disadvantages**:

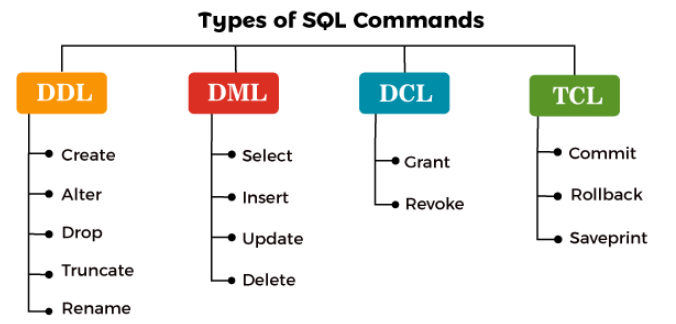
* Not suitable for simple, tabular data.
* Performance can degrade with a large number of nodes or connections.

**What Are the Subsets of SQL?**

SQL can be categorized into different subsets based on the type of operations performed:

1. **Data Definition Language (DDL)**: Defines the structure and schema of a database.
2. **Data Manipulation Language (DML)**: Deals with data manipulation within tables.
3. **Data Control Language (DCL)**: Manages access control to the database.
4. **Transaction Control Language (TCL)**: Controls the execution of transactions to maintain data integrity.
5. **Data Query Language(DQL)**: Used to fetch data from database

Subset of SQL:



**Data Definition Language (DDL)**

DDL is used to define or modify the structure of database objects like tables, indexes, and views. Key commands include:

* **CREATE**: Creates a new table, view, or database.

CREATE TABLE Students (

ID INT PRIMARY KEY,

Name VARCHAR(100),

Age INT

);

* **ALTER**: Modifies an existing table or database object (e.g., adding/removing columns).

ALTER TABLE Students ADD Email VARCHAR(100);

* **DROP**: Deletes an existing table, database, or index.

DROP TABLE Students;

* **TRUNCATE**: Removes all rows from a table but keeps its structure.

TRUNCATE TABLE Students;

**Data Query Language(DQL)**

DQL Allows you to fetch data from database

* **SELECT**: Retrieves data from a table.

SELECT \* FROM Students;

**Data Manipulation Language (DML)**

DML allows you to manipulate data within tables. It includes commands such as:

* **INSERT**: Adds new rows to a table.

INSERT INTO Students (ID, Name, Age) VALUES (1, 'John', 20);

* **UPDATE**: Modifies existing data in a table.

UPDATE Students SET Age = 21 WHERE ID = 1;

* **DELETE**: Removes rows from a table.

DELETE FROM Students WHERE ID = 1;

**Data Control Language (DCL)**

DCL manages user access and permissions in a database. Common DCL commands are:

* **GRANT**: Provides specific privileges to users (e.g., SELECT, INSERT, UPDATE).

GRANT SELECT ON Students TO user1;

* **REVOKE**: Removes privileges from users.

REVOKE SELECT ON Students FROM user1;

### ****TCL (Transaction Control Language) in SQL****

**TCL** commands are used to manage **transactions** in a database. A transaction is a sequence of one or more SQL operations that are executed as a single unit of work. Transactions ensure **data integrity** by grouping operations, making it possible to commit or roll back multiple changes simultaneously.

TCL commands are crucial for managing transactions to ensure **ACID** properties (Atomicity, Consistency, Isolation, and Durability).

### ****Key TCL Commands****:

1. **COMMIT**
2. **ROLLBACK**
3. **SAVEPOINT**

### ****1. COMMIT****

The COMMIT command is used to **permanently save** all the changes made in a transaction. Once a transaction is committed, the changes cannot be undone by a ROLLBACK.

#### **Syntax**:

COMMIT;

#### **Example**:

BEGIN TRANSACTION;

UPDATE Employees SET Salary = 5000 WHERE EmpID = 1;

COMMIT;

In this example, the salary of the employee with EmpID = 1 is updated, and the changes are saved permanently using the COMMIT command.

### ****2. ROLLBACK****

The ROLLBACK command is used to **undo** all the changes made in a transaction before it has been committed. This ensures that if something goes wrong, the database can be restored to its previous state.

#### **Syntax**:

ROLLBACK;

#### **Example**:

BEGIN TRANSACTION;

UPDATE Employees SET Salary = 5000 WHERE EmpID = 1;

ROLLBACK;

In this example, the ROLLBACK command is used to undo the salary update, so no changes are saved to the database.

### ****3. SAVEPOINT****

The SAVEPOINT command sets a **point within a transaction** to which you can later roll back. It allows for partial rollbacks in a long transaction, giving more control over how and where to undo changes.

#### **Syntax**:

SAVEPOINT savepoint\_name;

#### **Example**:

BEGIN TRANSACTION;

UPDATE Employees SET Salary = 5000 WHERE EmpID = 1;

SAVEPOINT sp1;

UPDATE Employees SET Salary = 6000 WHERE EmpID = 2;

ROLLBACK TO sp1;

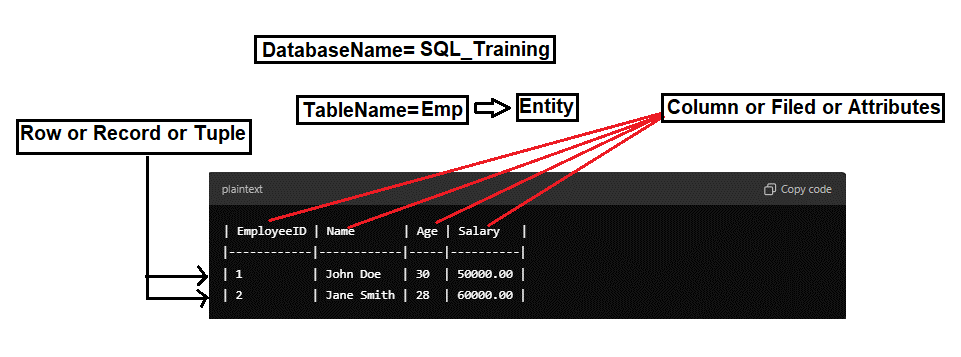
COMMIT;

Here, a SAVEPOINT named sp1 is created. If something goes wrong after the savepoint, the ROLLBACK TO sp1 undoes changes made after sp1, but keeps the updates before it. In this case, the salary change for EmpID = 1 is saved, while the change for EmpID = 2 is rolled back.

**Database Tables**

A **table** is a collection of related data entries in a database. It consists of:

* **Rows (Records, Tuple)**: Each row represents a single data entry or instance of the data structure.
* **Columns (Fields, Attributes)**: Each column represents a specific attribute of the data (e.g., name, age, salary).



**Table Records**

A **record** (or row) in a table is a single, structured data entry that consists of values for each of the table’s columns. Each record is unique and identifiable, often through a primary key, which is a unique identifier for each record.

**Example of Records**:

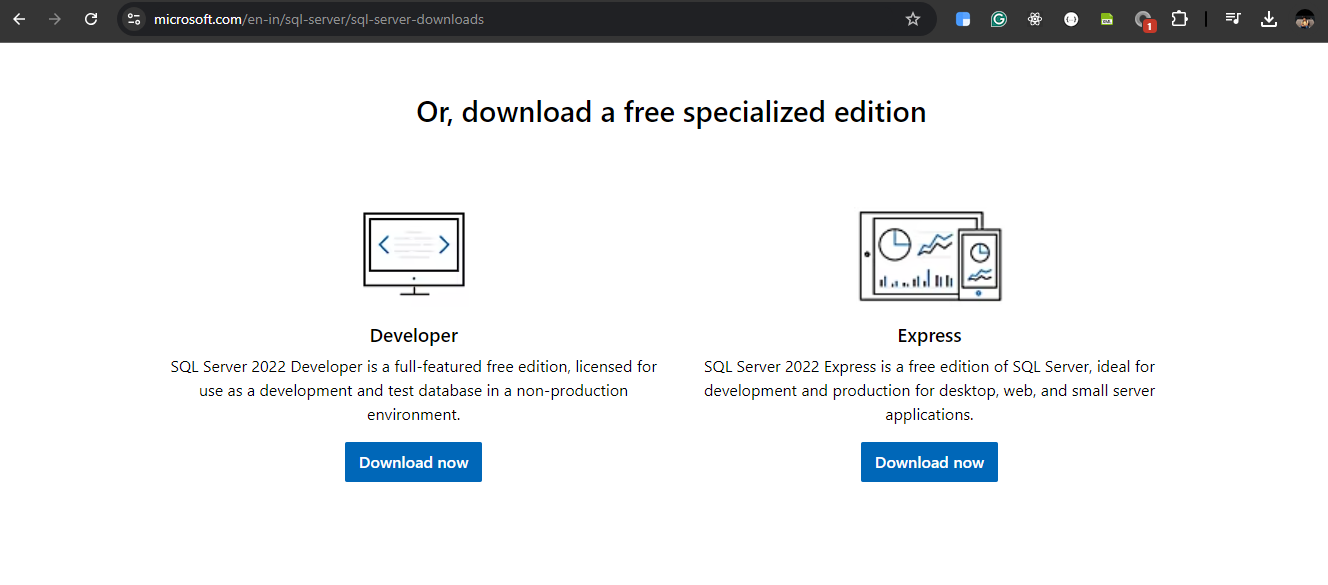
* The first record: 1, John Doe, 30, 50000.00
* The second record: 2, Jane Smith, 28, 60000.00

Steps to download & install SQL Server database:

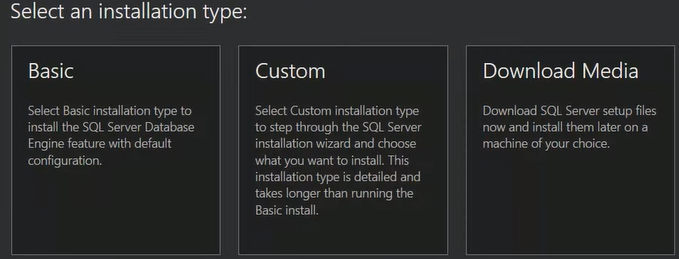
Step 1: Open chrome & paste below link

<https://www.microsoft.com/en-in/sql-server/sql-server-downloads>

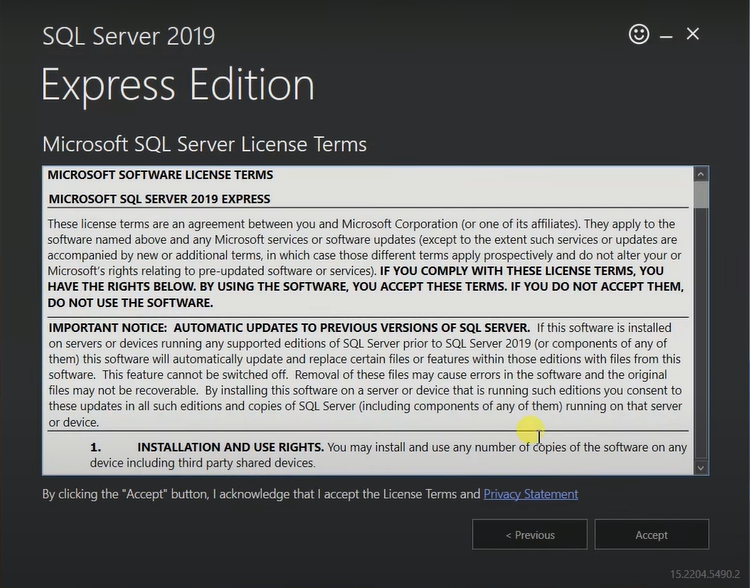
Step 2: Select Express one below snippet given



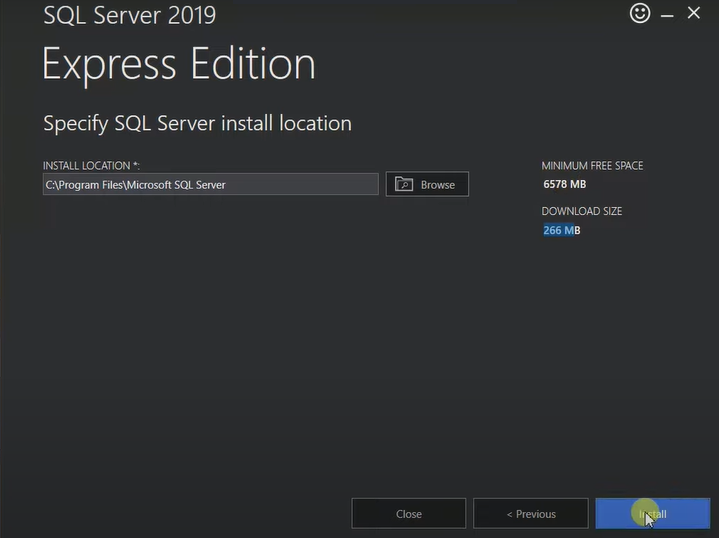
Step 3: Once Express is downloaded & while installing select Basic below is snippet



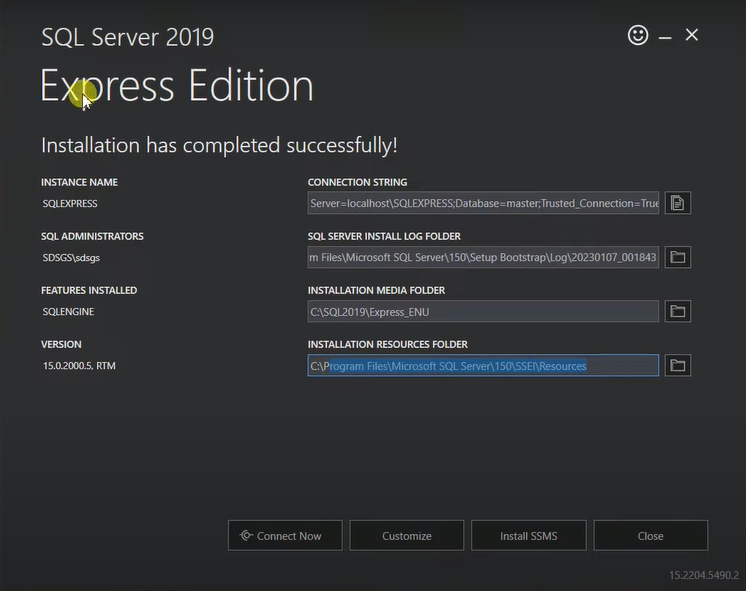
Step 4: Click in accept snippet given below:



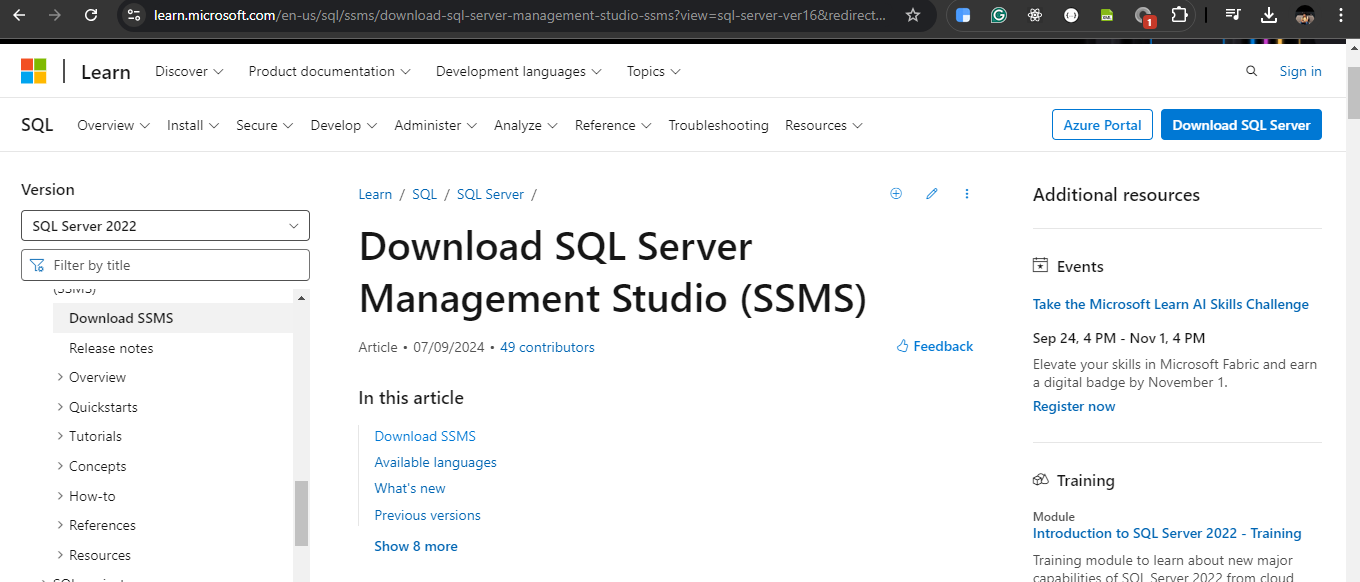
Step 5: Click on install & wait for few minutes



Step 6: Congratulation you SQL Server installed successfully below snippet you can see now click on install SSMS (SQL Server Management Studio) when you will click on install SSMS it will redirect to below link:



<https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16&redirectedfrom=MSDN>

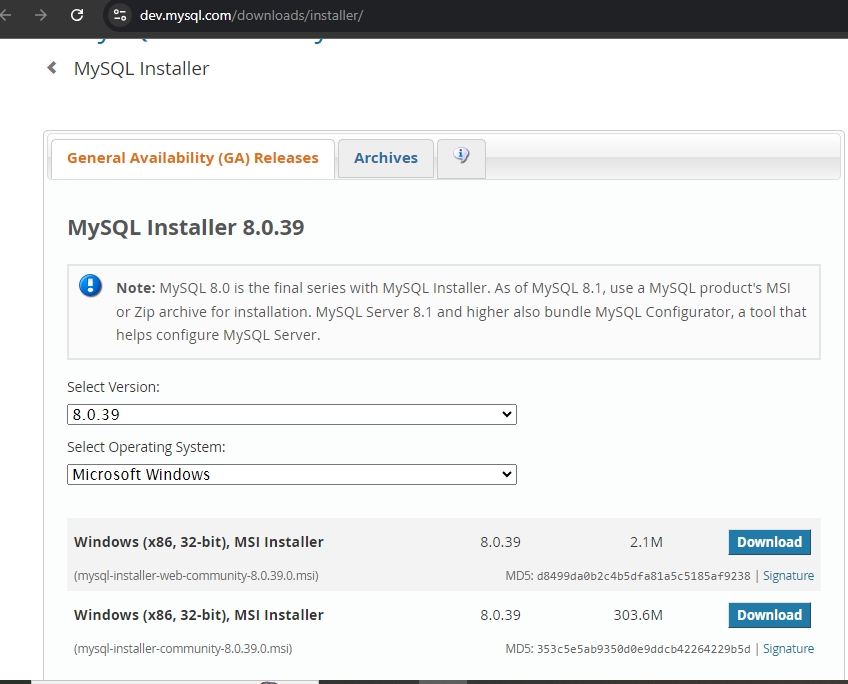


Step 7: Download SSMS and install it now you can you SQL Server Management Studio to write your SQL Query to interact with database

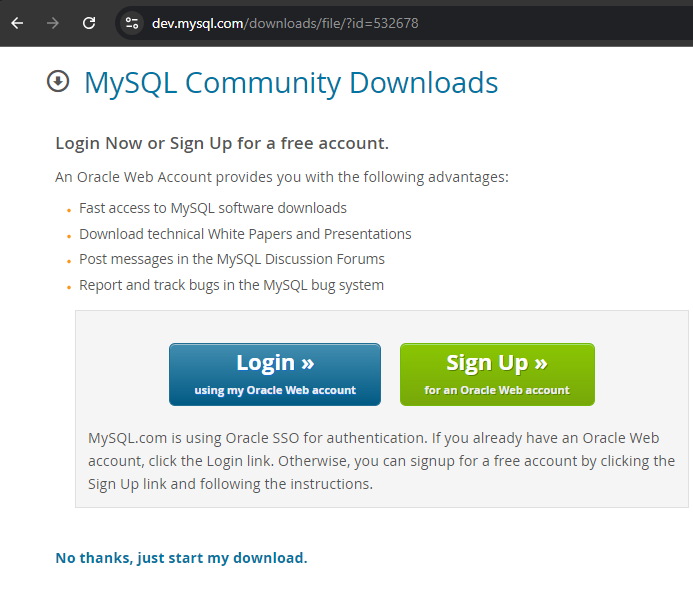
Download and Install MySql Database guide:-

Step 1: Open this link <https://dev.mysql.com/downloads/installer/>

Step 2: Select click on MySQL community Installer MSI below snippet given



Step 3: Click on No Thanks just start my download snippet given below:



Next steps refer from here <https://www.javatpoint.com/how-to-install-mysql>

**Difference Between ALTER & UPDATE**

**1. UPDATE**

* **Purpose**: Modifies the data within an existing table.
* **Used For**: Updating rows in a table.
* **Syntax**:

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

* **Example**:

UPDATE employees

SET salary = 60000

WHERE employee\_id = 5;

* + This updates the salary of the employee with employee\_id = 5.

**2. ALTER**

* **Purpose**: Modifies the structure of an existing table (e.g., adding, deleting, or modifying columns).
* **Used For**: Altering the schema or structure of a table.
* **Syntax**:
  + Add a new column:

ALTER TABLE table\_name

ADD column\_name datatype;

* + Modify an existing column:

ALTER TABLE table\_name

MODIFY column\_name datatype;

* + Drop a column:

ALTER TABLE table\_name

DROP COLUMN column\_name;

* **Example**:

ALTER TABLE employees

ADD phone\_number VARCHAR(15);

* + This adds a new column phone\_number to the employees table.

**Summary:**

* UPDATE changes the **data** inside the table.
* ALTER changes the **structure** of the table (like columns or constraints).

Top of Form

**Difference Between ALTER & RENAME**

**ALTER**

* **Purpose**: Modifies the **structure** of an existing table (e.g., adding, deleting, or modifying columns, changing constraints).
* **Used For**: Changing the table schema (columns, data types, constraints, etc.).
* **Syntax**:
  + Add a column:

ALTER TABLE table\_name

ADD column\_name datatype;

* + Modify a column:

ALTER TABLE table\_name

MODIFY column\_name datatype;

* + Drop a column:

ALTER TABLE table\_name

DROP COLUMN column\_name;

* + Add or drop constraints:

ALTER TABLE table\_name

ADD CONSTRAINT constraint\_name ...;

ALTER TABLE table\_name

DROP CONSTRAINT constraint\_name;

* **Example**:

ALTER TABLE employees

ADD phone\_number VARCHAR(15);

* + This adds a new column phone\_number to the employees table.

**2. RENAME**

* **Purpose**: Renames **database objects** such as tables or columns.
* **Used For**: Changing the name of tables, columns, or constraints.
* **Syntax**:
  + Rename a table:

RENAME TABLE old\_table\_name TO new\_table\_name;

* + Rename a column (syntax varies across databases):
    - In MySQL:

ALTER TABLE table\_name

RENAME COLUMN old\_column\_name TO new\_column\_name;

* + - In Oracle:

ALTER TABLE table\_name

RENAME COLUMN old\_column\_name TO new\_column\_name;

* **Example**:

RENAME TABLE employees TO staff;

* + This renames the employees table to staff.

**Key Differences:**

* **ALTER** changes the structure of a table (add/drop/modify columns, constraints).
* **RENAME** changes the name of the table or its columns, but does not affect the structure or data in the table.

**Summary:**

* Use ALTER to **modify the table structure**.
* Use RENAME to **change the name** of the table or columns.

Refer this resources for better understanding <https://prepinsta.com/dbms/alter/>

Bottom of Form

**Data Types in SQL:**

**1. INT (Integer)**

* **Definition**: Stores whole numbers (without decimals).
* **Example**:

employee\_id INT;

* **When to Use**: Use INT for storing numeric values that do not require decimals, such as IDs, age, quantity, and counts.

**2. DECIMAL (or NUMERIC)**

* **Definition**: Stores fixed-point numbers, useful for precise arithmetic calculations. It has two parts: precision (total number of digits) and scale (number of digits after the decimal point).
* **Example**:

salary DECIMAL(10, 2);

* **When to Use**: Use DECIMAL for financial data like prices, salaries, or any numbers that need exact precision. E.g., monetary values.

**3. VARCHAR (Variable-Length Character String)**

* **Definition**: Stores variable-length strings, where the length can vary up to a specified limit.
* **Example**:

email VARCHAR(255);

* **When to Use**: Use VARCHAR for text data where the length can vary, like names, addresses, or emails. Ideal for storing textual data that doesn't have a fixed length.

**4. CHAR (Fixed-Length Character String)**

* **Definition**: Stores fixed-length strings. If the data is shorter than the defined length, it is padded with spaces.
* **Example**:

status CHAR(1);

* **When to Use**: Use CHAR for fixed-length text fields, such as single-character status codes (like 'M' for Male or 'F' for Female), country codes, names, addresses, or emails. Ideal for storing textual data etc.

**5. DATE**

* **Definition**: Stores a date (without time), typically in the format YYYY-MM-DD.
* **Example**:

birthdate DATE;

* **When to Use**: Use DATE for storing only date information, such as birthdays, order dates, or registration dates.

**6. DATETIME**

* **Definition**: Stores both date and time, typically in the format YYYY-MM-DD HH:MM:SS.
* **Example**:

created\_at DATETIME;

* **When to Use**: Use DATETIME when you need to store both date and time information, like timestamps for when a record is created or modified.

**7. TIMESTAMP**

* **Definition**: Stores date and time. Automatically updates to the current time when the record is modified, depending on the SQL engine.
* **Example**:

last\_modified TIMESTAMP;

* **When to Use**: Use TIMESTAMP to store data that requires automatic tracking of changes, like last updated records or creation timestamps.

**8. FLOAT/REAL**

* **Definition**: Stores approximate floating-point numbers. Useful for scientific calculations but may lose precision.
* **Example**:

rating FLOAT;

* **When to Use**: Use FLOAT when you need to store large or small real numbers, such as scientific measurements or large decimal values that do not need absolute precision.

**9. BOOLEAN (or BIT)**

* **Definition**: Stores true/false values or 0/1 (depending on the database system).
* **Example**:

is\_active BOOLEAN;

* **When to Use**: Use BOOLEAN for true/false values, such as flags for user activity or subscription status.

**10. TEXT**

* **Definition**: Stores large amounts of variable-length text.
* **Example**:

description TEXT;

* **When to Use**: Use TEXT for storing long, unstructured textual data like product descriptions, user comments, or logs.

**11. BLOB (Binary Large Object)**

* **Definition**: Stores large binary data like images, audio, video, or other multimedia files.
* **Example**:

image BLOB;

* **When to Use**: Use BLOB for binary data like images, files, or any other large non-textual data.

**12. ENUM**

* **Definition**: Allows a predefined set of values for a column.
* **Example**:

gender ENUM('Male', 'Female', 'Other');

* **When to Use**: Use ENUM when a column must contain one of a set of predefined values, like gender, status, or categories.

**13. BIGINT**

* **Definition**: Stores large integer values, useful when INT is not large enough.
* **Example**:

transaction\_id BIGINT;

* **When to Use**: Use BIGINT when storing very large numbers, like transaction IDs or high-precision counters.

**14. SMALLINT**

* **Definition**: Stores small integer values.
* **Example**:

age SMALLINT;

* **When to Use**: Use SMALLINT when storing smaller whole numbers to save space, like small counts or ages.

**15. NVARCHAR**

**When to Use NVARCHAR**

* **Internationalization**: When your application needs to support multiple languages, use NVARCHAR to ensure proper storage of characters.
* **Special Characters**: If you expect to store text that includes special characters (like emojis, accented characters, or symbols), NVARCHAR is a good choice.

#### **Creating a Table with NVARCHAR Columns**

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

first\_name NVARCHAR(50),

last\_name NVARCHAR(50),

email NVARCHAR(100),

phone\_number NVARCHAR(20),

address NVARCHAR(255)

);

#### **Inserting Data into the NVARCHAR Columns**

-- Inserting data into the employees table

INSERT INTO employees (employee\_id, first\_name, last\_name, email, phone\_number, address)

VALUES

(1, N'Алексей', N'Иванов', N'aleksey@example.com', N'+7 123 456 7890', N'Москва, Россия'),

(2, N'John', N'Doe', N'john.doe@example.com', N'123-456-7890', N'123 Elm St, Springfield, IL'),

(3, N'张伟', N'李', N'zhang.wei@example.com', N'+86 123 456 7890', N'北京市, 中国');

**Summary of When to Use Each Data Type:**

* **INT, BIGINT, SMALLINT**: When storing numeric values without decimal points.
* **DECIMAL**: When storing precise decimal values (like money or financial data).
* **VARCHAR, CHAR**: When storing text (use VARCHAR for variable-length, CHAR for fixed-length).
* **DATE, DATETIME, TIMESTAMP**: When storing date or time information.
* **FLOAT**: For approximate values with a wide range but lower precision.
* **BOOLEAN**: For true/false or binary values.
* **TEXT, BLOB**: For large text or binary data.
* **ENUM**: For predefined choices in a column.

**VARCHAR vs NVARCHAR:**

**1. Character Storage:**

* **VARCHAR** (Variable Character) stores **non-Unicode** characters. Each character takes 1 byte of storage.
* **NVARCHAR** (National Variable Character) stores **Unicode** characters. Each character takes 2 bytes of storage.

**2. Supported Character Set:**

* **VARCHAR** supports characters defined by the database's collation, usually ASCII or extended ASCII characters (Latin alphabets).
* **NVARCHAR** supports **Unicode** characters, which can represent a wider range of characters, including non-Latin alphabets (such as Chinese, Japanese, Arabic, etc.).

**3. Maximum Length:**

* **VARCHAR** can store up to 8,000 characters.
* **NVARCHAR** can store up to 4,000 characters (because each Unicode character takes 2 bytes).

However, both can store up to **2 GB** of data if defined as VARCHAR(MAX) or NVARCHAR(MAX).

**4. Storage Efficiency:**

* **VARCHAR** is more storage-efficient for non-Unicode data since it uses only 1 byte per character.
* **NVARCHAR** uses more storage because each character requires 2 bytes, even if you're storing characters that could fit into a VARCHAR column.

**When to Use:**

* **Use VARCHAR** when:
  + You are storing **non-Unicode** data (e.g., English characters or other Latin-based alphabets).
  + Storage efficiency is a concern and you know that the data will not require Unicode characters.
* **Use NVARCHAR** when:
  + You need to store **Unicode** characters (e.g., data in multiple languages, special symbols, non-Latin alphabets).
  + You want to future-proof your application in case it needs to support internationalization.

**Example:**

* If you're storing names in English, VARCHAR would be more appropriate:

CREATE TABLE Users (

Name VARCHAR(100)

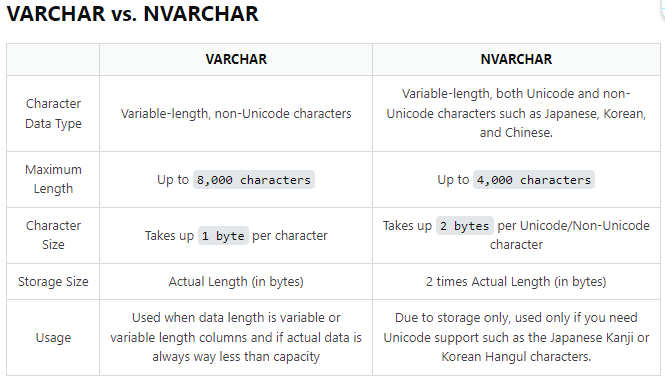
);

* If you're storing names in multiple languages (e.g., English, Chinese, Arabic), NVARCHAR would be the better choice:

CREATE TABLE Users (

Name NVARCHAR(100)

);



**For more information read below link** [**https://stackoverflow.com/questions/144283/what-is-the-difference-between-varchar-and-nvarchar**](https://stackoverflow.com/questions/144283/what-is-the-difference-between-varchar-and-nvarchar)

**For more information can refer this resource** [**https://www.w3schools.com/sql/sql\_datatypes.asp**](https://www.w3schools.com/sql/sql_datatypes.asp)

**Operators in SQL:**

### SQL Arithmetic Operators

SQL provides several arithmetic operators that can be used to perform mathematical operations on numeric data types. These operators allow you to manipulate data effectively within your SQL queries.

#### 1. Addition (+)

* **Description:** Adds two numeric values.
* **Example:**

SELECT 10 + 5 AS Sum; -- Result: 15

#### 2. Subtraction (-)

* **Description:** Subtracts one numeric value from another.
* **Example:**

SELECT 10 - 5 AS Difference; -- Result: 5

#### 3. Multiplication (\*)

* **Description:** Multiplies two numeric values.
* **Example:**

SELECT 10 \* 5 AS Product; -- Result: 50

#### 4. Division (/)

* **Description:** Divides one numeric value by another. Note that dividing by zero will result in an error.
* **Example:**

SELECT 10 / 5 AS Quotient; -- Result: 2

#### 5. Modulus (%)

* **Description:** Returns the remainder of a division operation.
* **Example:**

SELECT 10 % 3 AS Remainder; -- Result: 1

### SQL Comparison Operators

SQL comparison operators are used to compare two values. They return a boolean result (TRUE, FALSE, or UNKNOWN) based on the comparison. These operators are essential for filtering records in SQL queries.

#### Common Comparison Operators

1. **Equal to (=)**
   * **Description:** Checks if two values are equal.
   * **Example:**

SELECT \* FROM Employees WHERE Salary = 50000;

1. **Not equal to (!= or <>)**
   * **Description:** Checks if two values are not equal.
   * **Example:**

SELECT \* FROM Employees WHERE Salary <> 50000; -- Using <>

-- OR

SELECT \* FROM Employees WHERE Salary != 50000; -- Using !=

1. **Greater than (>)**
   * **Description:** Checks if the left value is greater than the right value.
   * **Example:**

SELECT \* FROM Employees WHERE Salary > 50000;

1. **Less than (<)**
   * **Description:** Checks if the left value is less than the right value.
   * **Example:**

SELECT \* FROM Employees WHERE Salary < 50000;

1. **Greater than or equal to (>=)**
   * **Description:** Checks if the left value is greater than or equal to the right value.
   * **Example:**

SELECT \* FROM Employees WHERE Salary >= 50000;

1. **Less than or equal to (<=)**
   * **Description:** Checks if the left value is less than or equal to the right value.
   * **Example:**

SELECT \* FROM Employees WHERE Salary <= 50000;

1. **BETWEEN**
   * **Description:** Checks if a value falls within a specified range.
   * **Example:**

SELECT \* FROM Employees WHERE Salary BETWEEN 40000 AND 60000;

1. **LIKE**
   * **Description:** Used to search for a specified pattern in a column (often with wildcard characters).
   * **Example:**

SELECT \* FROM Employees WHERE FirstName LIKE 'J%'; -- Names starting with 'J'

1. **IN**
   * **Description:** Checks if a value matches any value in a specified list.
   * **Example:**

SELECT \* FROM Employees WHERE EmployeeID IN (1, 2, 3);

1. **IS NULL**
   * **Description:** Checks if a value is NULL.
   * **Example:**

SELECT \* FROM Employees WHERE Bonus IS NULL;

1. **IS NOT NULL**
   * **Description:** Checks if a value is NOT NULL.
   * **Example:**

SELECT \* FROM Employees WHERE Bonus IS NOT NULL;

### Using Comparison Operators in Queries

Comparison operators are widely used in SQL queries to filter and retrieve specific records based on conditions. Below are some examples demonstrating their usage:

#### Example Table: Employees

| **EmployeeID** | **FirstName** | **LastName** | **Salary** |
| --- | --- | --- | --- |
| 1 | John | Doe | 50000 |
| 2 | Jane | Smith | 60000 |
| 3 | Alice | Johnson | 55000 |
| 4 | Bob | Brown | 45000 |
| 5 | Charlie | Davis | NULL |

#### Examples of Queries Using Comparison Operators

1. **Select Employees with Salary Greater than 50000**

SELECT \* FROM Employees WHERE Salary > 50000;

1. **Select Employees with Salary Less than or Equal to 50000**

SELECT \* FROM Employees WHERE Salary <= 50000;

1. **Select Employees whose First Names Start with 'A'**

SELECT \* FROM Employees WHERE FirstName LIKE 'A%';

1. **Select Employees with Salary in a Specific Range**

SELECT \* FROM Employees WHERE Salary BETWEEN 45000 AND 60000;

1. **Select Employees with a Specific EmployeeID**

SELECT \* FROM Employees WHERE EmployeeID IN (1, 3);

1. **Select Employees with NULL Bonus Values**

SELECT \* FROM Employees WHERE Bonus IS NULL;

### SQL Logical Operators

SQL logical operators are used to combine multiple conditions in a query. They allow you to construct complex queries by evaluating multiple expressions. The primary logical operators in SQL are AND, OR, and NOT.

#### 1. AND

* **Description:** The AND operator combines two or more conditions and returns TRUE only if all conditions are true.
* **Syntax:**

SELECT column1, column2, ...

FROM table\_name

WHERE condition1 AND condition2;

* **Example:**

SELECT \* FROM Employees

WHERE Salary > 50000 AND Department = 'Sales';

This query retrieves employees with a salary greater than 50,000 who work in the Sales department.

#### 2. OR

* **Description:** The OR operator combines two or more conditions and returns TRUE if at least one condition is true.
* **Syntax:**

SELECT column1, column2, ...

FROM table\_name

WHERE condition1 OR condition2;

* **Example:**

SELECT \* FROM Employees

WHERE Salary < 40000 OR Department = 'HR';

This query retrieves employees who have a salary less than 40,000 or work in the HR department.

#### 3. NOT

* **Description:** The NOT operator negates a condition, returning TRUE if the condition is false.
* **Syntax:**

SELECT column1, column2, ...

FROM table\_name

WHERE NOT condition;

* **Example:**

SELECT \* FROM Employees

WHERE NOT Department = 'Finance';

This query retrieves employees who do not work in the Finance department.

### Combining Logical Operators

You can combine logical operators to create more complex conditions using parentheses to group conditions as needed.

#### Example of Combined Conditions

SELECT \* FROM Employees

WHERE (Salary > 50000 AND Department = 'Sales') OR (Department = 'HR' AND NOT Bonus IS NULL);

This query retrieves employees who either have a salary greater than 50,000 and work in Sales, or work in HR and have a non-null Bonus.

### Practical Examples

#### Example Table: Employees

| **EmployeeID** | **FirstName** | **LastName** | **Salary** | **Department** | **Bonus** |
| --- | --- | --- | --- | --- | --- |
| 1 | John | Doe | 50000 | Sales | 5000 |
| 2 | Jane | Smith | 60000 | HR | NULL |
| 3 | Alice | Johnson | 55000 | IT | 3000 |
| 4 | Bob | Brown | 45000 | Finance | 2000 |
| 5 | Charlie | Davis | 40000 | Sales | NULL |

#### Examples of Queries Using Logical Operators

1. **Select Employees in Sales or IT with Salary Greater than 50000**

SELECT \* FROM Employees

WHERE (Department = 'Sales' OR Department = 'IT') AND Salary > 50000;

1. **Select Employees who have a Bonus or are in HR**

SELECT \* FROM Employees

WHERE Bonus IS NOT NULL OR Department = 'HR';

1. **Select Employees whose Salary is less than 50000 and do not work in Finance**

SELECT \* FROM Employees

WHERE Salary < 50000 AND NOT Department = 'Finance';

**Types of SQL Constraints**

1. **NOT NULL**
2. **UNIQUE**
3. **PRIMARY KEY**
4. **FOREIGN KEY**
5. **CHECK**
6. **DEFAULT**
7. **INDEX** (not a constraint in all contexts but often mentioned)

**1. NOT NULL**

* **Definition:** Ensures that a column cannot have a NULL value.
* **Usage:** When you want to make sure that every row in the table has a value for that column.
* **Example:**

CREATE TABLE Employees (

EmployeeID INT NOT NULL,

FirstName VARCHAR(50) NOT NULL,

LastName VARCHAR(50),

Age INT

);

*In this example, EmployeeID and FirstName must have values for each employee.*

**2. UNIQUE**

* **Definition:** Ensures that all values in a column are different.
* **Usage:** When you need to maintain unique values for a column but do not want to make it a primary key.
* **Example:**

CREATE TABLE Employees (

EmployeeID INT NOT NULL,

Email VARCHAR(100) UNIQUE,

FirstName VARCHAR(50),

LastName VARCHAR(50)

);

*In this example, no two employees can have the same email address.*

**3. PRIMARY KEY**

* **Definition:** A combination of NOT NULL and UNIQUE. It uniquely identifies each record in a table.
* **Usage:** When you need a unique identifier for each row in the table.
* **Example:**

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50)

);

*Here, EmployeeID serves as the unique identifier for each employee.*

**4. FOREIGN KEY**

* **Definition:** A key used to link two tables together. It ensures referential integrity between the two tables.
* **Usage:** When you want to enforce a relationship between two tables.
* **Example:**

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(50)

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

DepartmentID INT,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)

);

*In this example, DepartmentID in the Employees table references the DepartmentID in the Departments table, ensuring that each employee belongs to a valid department.*

**5. CHECK**

* **Definition:** Ensures that all values in a column satisfy a specific condition.
* **Usage:** When you want to enforce a certain range of values or a specific condition for a column.
* **Example:**

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Age INT CHECK (Age >= 18)

);

*This example ensures that all employees must be at least 18 years old.*

**6. DEFAULT**

* **Definition:** Provides a default value for a column when none is specified during the insertion of a new record.
* **Usage:** When you want to set a default value for a column.
* **Example:**

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

HireDate DATE DEFAULT CURRENT\_DATE

);

*Here, if no HireDate is specified, it will default to the current date.*

**7. INDEX**

* **Definition:** An index improves the speed of data retrieval operations on a database table at the cost of additional space and decreased performance on data modification.
* **Usage:** When you want to improve query performance.
* **Example:**

CREATE INDEX idx\_lastname ON Employees (LastName);

*This creates an index on the LastName column to speed up queries that filter by last name.*

**Summary**

SQL constraints are essential for ensuring data integrity and enforcing rules at the database level. Here’s a quick overview of their functionalities:

* **NOT NULL:** Prevents NULL values.
* **UNIQUE:** Ensures all values in a column are unique.
* **PRIMARY KEY:** Combines NOT NULL and UNIQUE; uniquely identifies a row.
* **FOREIGN KEY:** Ensures referential integrity between two tables.
* **CHECK:** Validates data against a condition.
* **DEFAULT:** Sets a default value for a column.
* **INDEX:** Enhances query performance.

**SQL Functions:**

SQL functions are essential for performing operations on data stored in a database. They can be categorized into two main types: **Scalar Functions** and **Aggregate Functions**. Below is a comprehensive overview of SQL functions, including their definitions, examples, and usage.

**Types of SQL Functions**

1. **Scalar Functions**
2. **Aggregate Functions**
3. **String Functions**
4. **Numeric Functions**
5. **Date Functions**
6. **Conversion Functions**
7. **User-Defined Functions (UDF)**

**1. Scalar Functions**

**Definition:** Scalar functions return a single value for each row processed and operate on a single value. Examples include string manipulation, mathematical calculations, and date/time operations.

**Examples:**

* **UPPER()**: Converts a string to uppercase.

SELECT UPPER(FirstName) AS UpperFirstName FROM Employees;

* **LOWER()**: Converts a string to lowercase.

SELECT LOWER(LastName) AS LowerLastName FROM Employees;

* **ROUND()**: Rounds a numeric value to a specified number of decimal places.

SELECT ROUND(Salary, 2) AS RoundedSalary FROM Employees;

* **LENGTH()**: Returns the length of a string.

SELECT LENGTH(FirstName) AS FirstNameLength FROM Employees;

**2. Aggregate Functions**

**Definition:** Aggregate functions perform calculations on a set of values and return a single value. These are often used with the GROUP BY clause.

**Examples:**

* **COUNT()**: Returns the number of rows that match a specified criterion.

SELECT COUNT(\*) AS TotalEmployees FROM Employees;

* **SUM()**: Returns the total sum of a numeric column.

SELECT SUM(Salary) AS TotalSalary FROM Employees;

* **AVG()**: Returns the average value of a numeric column.

SELECT AVG(Salary) AS AverageSalary FROM Employees;

* **MIN()**: Returns the minimum value in a set.

SELECT MIN(Age) AS YoungestEmployee FROM Employees;

* **MAX()**: Returns the maximum value in a set.

SELECT MAX(Salary) AS HighestSalary FROM Employees;

**3. String Functions**

String functions are a subset of scalar functions specifically designed for manipulating string data.

**Examples:**

* **CONCAT()**: Concatenates two or more strings.

SELECT CONCAT(FirstName, ' ', LastName) AS FullName FROM Employees;

* **SUBSTRING()**: Extracts a substring from a string.

SELECT SUBSTRING(Email, 1, 5) AS EmailPrefix FROM Employees;

* **TRIM()**: Removes leading and trailing spaces from a string.

SELECT TRIM(FirstName) AS TrimmedFirstName FROM Employees;

**4. Numeric Functions**

Numeric functions perform mathematical calculations and operations.

**Examples:**

* **ABS()**: Returns the absolute value of a number.

SELECT ABS(-25) AS AbsoluteValue;

* **CEIL()**: Returns the smallest integer value that is greater than or equal to a number.

SELECT CEIL(4.3) AS CeilingValue; -- Returns 5

* **FLOOR()**: Returns the largest integer value that is less than or equal to a number.

SELECT FLOOR(4.8) AS FloorValue; -- Returns 4

**5. Date Functions**

Date functions are used to manipulate and extract information from date and time values.

**Examples:**

* **NOW()**: Returns the current date and time.

SELECT NOW() AS CurrentDateTime;

* **DATEADD()**: Adds a specified interval to a date.

SELECT DATEADD(YEAR, 1, HireDate) AS NextYearHireDate FROM Employees;

* **DATEDIFF()**: Returns the difference between two dates.

SELECT DATEDIFF(NOW(), HireDate) AS DaysSinceHired FROM Employees;

**6. Conversion Functions**

Conversion functions are used to convert data from one type to another.

**Examples:**

* **CAST()**: Converts a value to a specified data type.

SELECT CAST(Salary AS VARCHAR(10)) AS SalaryAsString FROM Employees;

* **CONVERT()**: Converts a value from one type to another (specific to some SQL databases).

SELECT CONVERT(DATE, HireDate) AS ConvertedHireDate FROM Employees;

**7. User-Defined Functions (UDF)**

**Definition:** User-defined functions allow you to create your own functions to perform specific calculations or operations that can be reused.

**Example in SQL Server:**

* Creating a simple user-defined function that returns the full name of an employee.

CREATE FUNCTION GetFullName(@FirstName VARCHAR(50), @LastName VARCHAR(50))

RETURNS VARCHAR(100)

AS

BEGIN

RETURN CONCAT(@FirstName, ' ', @LastName);

END;

* Using the function:

SELECT dbo.GetFullName(FirstName, LastName) AS FullName FROM Employees;

### MySQL Syntax for Creating a Function

1. **Define the Function**: Use CREATE FUNCTION without the @ symbol for parameters.
2. **Use DELIMITER**: Change the delimiter to allow for multiple statements.

**Example in MYSQL:**

Here's how to create a GetFullName function in MySQL:

DELIMITER //

CREATE FUNCTION GetFullName(FirstName VARCHAR(50), LastName VARCHAR(50))

RETURNS VARCHAR(100)

BEGIN

RETURN CONCAT(FirstName, ' ', LastName);

END //

DELIMITER ;

### Explanation

* **DELIMITER //**: Changes the default statement delimiter from ; to // to avoid premature termination of the function definition.
* **Parameters**: You don’t use @ for parameters in MySQL functions; just specify the name and type.
* **CONCAT**: The CONCAT function is used to concatenate the first name and last name with a space in between.
* **RETURN**: The function returns the full name as a concatenated string.

### Example Usage

After creating the function, you can call it like this:

SELECT GetFullName('John', 'Doe') AS FullName;

### Expected Output

This query will return:

+------------+

| FullName |

+------------+

| John Doe |

+------------+

**Difference Between Function and Stored Procedure**

### 1. ****Using a Stored Procedure**** to Select Top 5 Rows

Stored procedures allow you to execute complex SQL statements and return full result sets.

#### Step-by-Step:

1. **Create the stored procedure**:
   * This procedure selects the top 5 rows from a table based on the ordering of a specific column (e.g., id or created\_date).

DELIMITER //

CREATE PROCEDURE SelectTop5()

BEGIN

SELECT \*

FROM your\_table\_name

ORDER BY some\_column

LIMIT 5;

END //

DELIMITER ;

* **your\_table\_name**: Replace with your actual table name.
* **some\_column**: Replace with the column name based on which you want to order the rows (e.g., id, created\_date).

1. **Call the stored procedure**:
   * Use this to execute the procedure and get the top 5 rows.

CALL SelectTop5();

This will output the top 5 rows from the specified table.

#### Example:

Suppose we have a table called employees with columns like id, first\_name, and salary. The following procedure selects the top 5 employees with the highest salary:

DELIMITER //

CREATE PROCEDURE SelectTop5Employees()

BEGIN

SELECT id, first\_name, salary

FROM employees

ORDER BY salary DESC

LIMIT 5;

END //

DELIMITER ;

To call this procedure:

CALL SelectTop5Employees();

### 2. ****Using a Function**** to Return a Scalar Value (Top 5 IDs)

While functions in MySQL cannot return a full result set like stored procedures, they can return scalar values or aggregated data. Here’s an example of a function that returns the top 5 IDs as a concatenated string.

#### Step-by-Step:

1. **Create the function**:
   * This function uses the GROUP\_CONCAT function to concatenate the top 5 id values from the table.

DELIMITER //

CREATE FUNCTION GetTop5IDs()

RETURNS VARCHAR(255)

DETERMINISTIC

BEGIN

DECLARE result VARCHAR(255) DEFAULT '';

SELECT GROUP\_CONCAT(id ORDER BY id LIMIT 5) INTO result

FROM your\_table\_name;

RETURN result;

END //

DELIMITER ;

### Query Write Explanations:

DELIMITER //

* **Explanation**: By default, MySQL uses ; to terminate statements. However, when defining stored procedures or functions, we need to temporarily change the statement delimiter because the function itself contains semicolons (;). So, we use // as a custom delimiter for the duration of the function definition.

CREATE FUNCTION GetTop5IDs()

RETURNS VARCHAR(255)

DETERMINISTIC

* **CREATE FUNCTION GetTop5IDs()**: This line defines a function named GetTop5IDs. It takes no parameters and returns a value.
* **RETURNS VARCHAR(255)**: Specifies that the function will return a VARCHAR type string, with a maximum length of 255 characters. The returned value will be the result of the query, which in this case will be a concatenated string of IDs.
* **DETERMINISTIC**: Declares that the function always returns the same result when given the same inputs, meaning that it doesn’t depend on external factors like the state of the database. This allows for better optimization.

BEGIN

* **Explanation**: The BEGIN keyword marks the start of the block of code that defines the logic of the function.

DECLARE result VARCHAR(255) DEFAULT '';

* **DECLARE result**: This declares a local variable result within the function. It's a VARCHAR type with a maximum length of 255 characters.
* **DEFAULT ''**: Initializes the result variable with an empty string ('') by default, so the variable has a value even before any operations.

SELECT GROUP\_CONCAT(id ORDER BY id LIMIT 5) INTO result

FROM your\_table\_name;

* **SELECT GROUP\_CONCAT(id ORDER BY id LIMIT 5) INTO result**: This line performs a SELECT query.
  + **GROUP\_CONCAT(id)**: Aggregates (concatenates) the values from the id column of the table into a single string, separated by commas.
  + **ORDER BY id**: Orders the id values in ascending order (smallest to largest).
  + **LIMIT 5**: Restricts the query to the top 5 rows only.
  + **INTO result**: Assigns the concatenated result (i.e., the top 5 ids) to the local result variable.
* **FROM your\_table\_name**: Specifies the table from which to retrieve the data. You need to replace your\_table\_name with the actual name of your table.

RETURN result;

* **RETURN result**: This line returns the value stored in the result variable. In this case, the returned value is a comma-separated string of the top 5 id values.

END //

* **Explanation**: The END keyword marks the end of the function's logic. The // delimiter is used to terminate the function definition (as set earlier with DELIMITER //).

DELIMITER ;

* **Explanation**: This restores the default delimiter (;) for MySQL, so after defining the function, regular SQL statements will again use ; as the statement terminator.

1. **Use the function**:
   * Call this function like a normal SQL function:

SELECT GetTop5IDs();

This will return a string with the top 5 id values concatenated, separated by commas.

#### Example:

For the employees table, the function can be used as follows:

DELIMITER //

CREATE FUNCTION GetTop5EmployeeIDs()

RETURNS VARCHAR(255)

DETERMINISTIC

BEGIN

DECLARE result VARCHAR(255) DEFAULT '';

SELECT GROUP\_CONCAT(id ORDER BY salary DESC LIMIT 5) INTO result

FROM employees;

RETURN result;

END //

DELIMITER ;

To get the top 5 employee IDs:

SELECT GetTop5EmployeeIDs();

This will return a result like: 1, 3, 5, 7, 8 based on their salary.

### Summary of Differences:

* **Stored Procedure**:
  + Can return full result sets (multiple rows and columns).
  + Use CALL procedure\_name(); to invoke.
  + Best for complex operations like selecting, inserting, or updating multiple rows.
* **Function**:
  + Can return scalar values (like strings, integers, or single values).
  + Use SELECT function\_name(); to invoke.
  + Best for calculations, transformations, and returning a single value.

### Conclusion

* Use a **stored procedure** when you want to return full result sets like the top 5 rows.
* Use a **function** when you want to return a single value, such as a concatenation or an aggregated result of the top 5 rows.

## **Data Control Language (DCL) in MySQL:**

DCL commands are used to control the access and privileges in a database. The primary DCL commands are GRANT and REVOKE, which control user permissions on the database objects.

### ****1. Common DCL Commands:****

* **GRANT**: Gives privileges to a user on the database or specific tables.
* **REVOKE**: Removes previously granted privileges from a user.

### ****Pre-requisites****

Before performing any DCL operations, ensure that:

1. You have MySQL installed and running.
2. You have access to the root user or any admin-level user with sufficient privileges.

## **Practical Examples with DCL Commands**

### ****Step 1: View Existing Users****

You can view all users present in your MySQL database by running the following query:

SELECT user FROM mysql.user;

This will display a list of users who have access to the MySQL server.

### ****Step 2: Check Privileges for a User****

To check which privileges a specific user (like the root user) has, use the SHOW GRANTS command:

SHOW GRANTS FOR 'root'@'localhost';

This will display all the roles and privileges assigned to the root user.

### ****Step 3: Create a New User****

Now, let's create a new user in MySQL. In this case, we'll create a user named user1 with the password root:

CREATE USER 'user1'@'localhost' IDENTIFIED BY 'root';

This command creates a new user user1 with the password root, but as of now, this user has no privileges to perform any actions in the database.

### ****Step 4: Grant Privileges to the User****

You can grant specific or all privileges to a user on a specific database or table. Here's how:

#### **Grant All Privileges on All Databases**

Grant all privileges to user1 on all databases and tables:

GRANT ALL PRIVILEGES ON \*.\* TO 'user1'@'localhost';

This command grants all privileges (such as SELECT, INSERT, UPDATE, DELETE, ALTER, etc.) to user1 on every database and table.

#### **Verify the Privileges**

To verify the privileges of user1, run:

SHOW GRANTS FOR 'user1'@'localhost';

### ****Step 5: Revoke Privileges from the User****

If you want to revoke all privileges from user1, use the following command:

REVOKE ALL PRIVILEGES ON \*.\* FROM 'user1'@'localhost';

This command removes all privileges granted to user1. After this, user1 will not be able to perform any actions unless new privileges are granted.

### ****Step 6: Grant Specific Privileges on a Specific Database****

You can grant specific privileges to a user on a particular database or table. Here are some examples:

#### **Grant SELECT Privilege**

Grant the SELECT privilege to user1 on a database called Amarjeet:

GRANT SELECT ON Amarjeet.\* TO 'user1'@'localhost';

Now, user1 can only perform SELECT queries on the Amarjeet database, but no other operations (such as INSERT, UPDATE, or DELETE) are allowed.

#### **Grant ALTER Privilege**

Grant the ALTER privilege to user1 on the Amarjeet database:

GRANT ALTER ON Amarjeet.\* TO 'user1'@'localhost';

This allows user1 to modify the structure of tables within the Amarjeet database (e.g., adding or deleting columns).

#### **Grant Multiple Privileges**

You can also grant multiple specific privileges at once. For example:

GRANT SELECT, INSERT, UPDATE ON Amarjeet.\* TO 'user1'@'localhost';

This command gives user1 the ability to select, insert, and update data in all tables in the Amarjeet database.

### ****Step 7: Revoke Specific Privileges from a User****

Just like granting privileges, you can revoke specific privileges from a user. For example, to revoke the INSERT privilege from user1 on the Amarjeet database:

REVOKE INSERT ON Amarjeet.\* FROM 'user1'@'localhost';

After this, user1 will no longer be able to insert data into any table in the Amarjeet database.

### ****Step 8: Practical Testing****

Once you have granted or revoked privileges, you should test the access by logging in as user1 to see what operations can be performed.

#### **Login as user1**

Open a terminal or command prompt and login as user1:

mysql -u user1 -p

You will be prompted for the password. Enter the password (in this case, root).

#### **Test SELECT Query**

If you granted the SELECT privilege, you can run a query like this:

SELECT \* FROM Amarjeet.some\_table;

This should work if the SELECT privilege was granted.

#### **Test INSERT Query**

If you revoked the INSERT privilege and try to insert data, it should fail:

INSERT INTO Amarjeet.some\_table (column1, column2) VALUES ('value1', 'value2');

You should get a permission error because the INSERT privilege has been revoked.

### ****Step 9: Removing a User****

If you need to remove a user, you can do so with the following command:

DROP USER 'user1'@'localhost';

This will delete the user1 user from the MySQL system.

## **Summary of DCL Commands**

| **Command** | **Description** |
| --- | --- |
| CREATE USER 'user1'@'localhost' | Creates a new user user1 in the MySQL system. |
| GRANT ALL PRIVILEGES ON \*.\* | Grants all privileges to the user on all databases. |
| GRANT SELECT ON db.\* | Grants only the SELECT privilege on a specific database or table. |
| REVOKE ALL PRIVILEGES | Revokes all privileges from a user, removing any ability to perform operations. |
| SHOW GRANTS FOR 'user1'@'localhost' | Displays the privileges assigned to a specific user. |
| DROP USER 'user1'@'localhost' | Deletes a user from the MySQL system. |

## **TCL (Transaction Control Language) in MySQL: Practical with Notes**

TCL commands are used to manage transactions in the database. Transactions allow you to execute a series of SQL operations as a single unit. The main TCL commands are COMMIT, ROLLBACK, and SAVEPOINT.

### ****TCL Commands: Overview****

* **START TRANSACTION**: Begins a transaction. All subsequent queries will be part of this transaction until it is committed or rolled back.
* **COMMIT**: Saves all the changes made in the transaction permanently to the database.
* **ROLLBACK**: Undoes all the changes made in the transaction, reverting the database to its previous state.
* **SAVEPOINT**: Sets a point within a transaction to which you can later roll back.
* **ROLLBACK TO SAVEPOINT**: Rolls back the transaction to the specified savepoint.

**Step 1: Creating a Database and Table**

First, we create a new database Transaction\_DB (if it doesn’t already exist) and a table called employees inside this database. The table includes columns for employee ID, name, and salary.

CREATE DATABASE IF NOT EXISTS Transaction\_DB;

USE Transaction\_DB;

CREATE TABLE employees (

employee\_id INT AUTO\_INCREMENT PRIMARY KEY,

employee\_name VARCHAR(50),

employee\_salary DECIMAL(10, 2)

);

* **Explanation**:
  + CREATE DATABASE IF NOT EXISTS Transaction\_DB; creates the Transaction\_DB database.
  + USE Transaction\_DB; selects the database for further operations.
  + CREATE TABLE employees creates a table with three columns: employee\_id (auto-incrementing primary key), employee\_name, and employee\_salary.

**Step 2: Inserting Initial Data into the Table**

Next, we populate the employees table with some initial data:

INSERT INTO employees (employee\_name, employee\_salary) VALUES

('Alice', 60000),

('Bob', 55000),

('Charlie', 70000);

SELECT \* FROM employees;

* **Explanation**:
  + This step inserts three employee records into the table with the names Alice, Bob, and Charlie and their respective salaries.
  + SELECT \* FROM employees; retrieves and displays the current records in the employees table.

**Step 3: Starting a Transaction**

Now, we begin a transaction to group several operations together. If something goes wrong, we can roll back any changes before committing them.

-- Start a new transaction

START TRANSACTION;

* **Explanation**:
  + START TRANSACTION; begins a transaction. All operations performed after this will be part of this transaction, and you can either commit or rollback the changes later.

**Step 4: Insert New Records and Create SAVEPOINTs**

We proceed with inserting more employees and creating SAVEPOINTs at different stages in the transaction.

-- Insert a new employee

INSERT INTO employees (employee\_name, employee\_salary) VALUES ('amarjeet', 65000);

-- Create a SAVEPOINT after inserting 'amarjeet'

SAVEPOINT savepoint\_amarjeet;

-- Insert another employee

INSERT INTO employees (employee\_name, employee\_salary) VALUES ('sanjeet', 58000);

-- Create another SAVEPOINT after inserting 'sanjeet'

SAVEPOINT savepoint\_sanjeet;

-- Insert one more employee

INSERT INTO employees (employee\_name, employee\_salary) VALUES ('hira', 72000);

* **Explanation**:
  + INSERT INTO employees... adds new employee records ('amarjeet', 'sanjeet', and 'hira').
  + SAVEPOINT savepoint\_amarjeet; creates a savepoint after inserting 'amarjeet'. This allows us to rollback the transaction to this point if needed.
  + SAVEPOINT savepoint\_sanjeet; creates another savepoint after inserting 'sanjeet'.
  + The final insertion adds an employee 'hira' with a salary of 72000.

**Step 5: Viewing the Table After Inserting New Records**

We can now check the table to see the current state after these insertions.

-- View the table after these operations

SELECT \* FROM employees;

* **Explanation**:
  + This command retrieves the current records in the employees table, showing the full list of employees after the three recent insertions.

**Step 6: Rolling Back to a SAVEPOINT**

If something goes wrong or we decide to undo specific changes, we can use ROLLBACK TO to revert to a specific savepoint.

-- Rollback to savepoint\_sanjeet (undoes the insertion of 'hira')

ROLLBACK TO savepoint\_sanjeet;

-- View the table after rollback

SELECT \* FROM employees;

* **Explanation**:
  + ROLLBACK TO savepoint\_sanjeet; undoes all changes made after savepoint\_sanjeet. This means the insertion of 'hira' is undone, but 'sanjeet' remains in the table.
  + SELECT \* FROM employees; displays the records after the rollback. The table will no longer have the employee 'hira'.

**Step 7: Committing the Final Transaction**

Once you are satisfied with the state of the transaction, you can commit the changes to make them permanent.

-- Commit the transaction

COMMIT;

-- View the final state of the table

SELECT \* FROM employees;

* **Explanation**:
  + COMMIT; finalizes the transaction, making all changes permanent up to the most recent savepoint or operation.
  + SELECT \* FROM employees; shows the final state of the table after the commit.

**Summary of SAVEPOINT Workflow**

1. **Start a Transaction**: Begin a transaction with START TRANSACTION;.
2. **Perform Operations**: Execute SQL statements (inserts, updates, etc.).
3. **Create SAVEPOINT**: Use SAVEPOINT savepoint\_name; to mark a specific point in the transaction.
4. **Roll Back if Needed**: If something goes wrong, rollback to a savepoint with ROLLBACK TO savepoint\_name;.
5. **Commit Changes**: Once satisfied, finalize the transaction with COMMIT;.

**Example Transaction Flow with SAVEPOINTs**

| **Operation** | **Action** |
| --- | --- |
| START TRANSACTION; | Begin a new transaction. |
| INSERT INTO... | Add records to the table. |
| SAVEPOINT savepoint\_name; | Create a savepoint after inserting certain records. |
| ROLLBACK TO savepoint\_name; | Undo changes back to a specific savepoint. |
| COMMIT; | Finalize the transaction, making all changes permanent. |

**Conclusion**

* **SAVEPOINT** provides finer control over a transaction, allowing you to rollback to specific points without discarding the entire transaction.
* It's particularly useful in long or complex transactions where some operations may need to be undone without affecting earlier successful operations.
* By using START TRANSACTION, SAVEPOINT, ROLLBACK TO, and COMMIT, you can ensure that your database remains consistent while handling potential errors during large transactions.

**Difference Between ROLLBACK and SAVEPOINT**

In SQL transactions, both ROLLBACK and SAVEPOINT are important tools for managing and controlling the flow of transactions. However, they serve different purposes and provide different levels of control over transaction management. Here's a detailed comparison:

| **Aspect** | **ROLLBACK** | **SAVEPOINT** |
| --- | --- | --- |
| **Definition** | ROLLBACK undoes the entire transaction, discarding all changes made during the transaction since it started. | SAVEPOINT marks a specific point in a transaction to which you can later rollback without affecting earlier parts of the transaction. |
| **Scope** | A complete rollback of the entire transaction. | A partial rollback to a specific point in the transaction. |
| **Use Case** | Used when you want to discard **all** the changes made during a transaction. | Used when you want to rollback only **a portion** of the transaction, while preserving earlier changes. |
| **Effect on Transaction** | Ends the transaction when called, and all changes are undone. | Does not end the transaction; only reverses changes made after the SAVEPOINT was created. |
| **Granularity** | A broad rollback that affects the entire transaction. | Provides finer control by allowing selective rollback to specific points within a transaction. |
| **Example** | If you insert 3 records and call ROLLBACK, all 3 inserts will be undone. | If you insert 3 records and create SAVEPOINTs after each, you can rollback to any savepoint, undoing only specific changes. |
| **Syntax** | ROLLBACK; | SAVEPOINT savepoint\_name;  ROLLBACK TO savepoint\_name; |
| **Undoing Partial Changes** | Cannot selectively undo changes; it will undo **everything** from the start of the transaction. | Allows partial rollback, undoing changes only after the savepoint. |
| **Commit** | You cannot ROLLBACK after committing the transaction. | SAVEPOINT can be used **before** committing to mark the current state. You can still COMMIT after rolling back to a savepoint. |
| **Performance** | Potentially slower since it undoes all changes in the transaction. | More efficient when only specific changes need to be reverted, leaving earlier changes intact. |

**Practical Example of ROLLBACK**

* **Scenario**: You start a transaction and make several changes, but later realize there’s a mistake and want to undo **all changes**.

START TRANSACTION;

INSERT INTO employees (employee\_name, employee\_salary) VALUES ('Alice', 60000);

INSERT INTO employees (employee\_name, employee\_salary) VALUES ('Bob', 55000);

-- Realize there's an error, so undo everything

ROLLBACK;

-- Now, both 'Alice' and 'Bob' will not be in the table

SELECT \* FROM employees;

**Practical Example of SAVEPOINT**

* **Scenario**: You start a transaction, insert multiple records, and create savepoints after each insert. If something goes wrong, you can rollback to a specific savepoint, undoing only part of the changes.

START TRANSACTION;

-- Insert two records

INSERT INTO employees (employee\_name, employee\_salary) VALUES ('Charlie', 70000);

SAVEPOINT sp\_charlie;

INSERT INTO employees (employee\_name, employee\_salary) VALUES ('David', 65000);

SAVEPOINT sp\_david;

-- Realize there's an error in David's data, rollback only to the savepoint after 'Charlie'

ROLLBACK TO sp\_charlie;

-- 'David' is removed, but 'Charlie' remains in the table

SELECT \* FROM employees;

-- Commit the transaction to finalize the changes

COMMIT;

**Key Takeaways:**

1. **ROLLBACK** undoes **everything** from the start of the transaction, while **SAVEPOINT** allows you to rollback to specific points within the transaction.
2. Use **SAVEPOINT** for more fine-grained control during complex transactions, where you may want to reverse only some changes while preserving others.
3. **ROLLBACK** is useful when the entire transaction needs to be discarded due to a critical error or bad data.

These two tools are part of transaction management and help ensure data integrity by controlling how changes are applied or reversed in a database.

Top of Form

SQL Keys:

Bottom of Form

**1. Primary Key**

* **Definition**: A primary key is a column (or a combination of columns) that uniquely identifies each row in a table.
* **Characteristics**:
  + Must contain unique values.
  + Cannot contain NULL values.
  + Each table can have only one primary key.
* **Example**:

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(100)

);

**2. Foreign Key**

* **Definition**: A foreign key is a column (or a combination of columns) that creates a link between two tables. It refers to the primary key of another table.
* **Characteristics**:
  + Used to enforce referential integrity.
  + A foreign key can accept duplicate values and can contain NULLs.
* **Example**:

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(100)

);

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(100),

department\_id INT,

FOREIGN KEY (department\_id) REFERENCES departments(department\_id)

);

**3. Unique Key**

* **Definition**: A unique key is a constraint that ensures all values in a column (or a group of columns) are unique across the table, but it can allow NULL values.
* **Characteristics**:
  + Each table can have multiple unique keys.
  + Unlike primary keys, unique keys can accept NULL values.
* **Example**:

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

employee\_email VARCHAR(100) UNIQUE

);

**4. Composite Key**

* **Definition**: A composite key is a primary key made up of two or more columns. It is used when no single column can uniquely identify a row.
* **Characteristics**:
  + The combination of values in the specified columns must be unique.
* **Example**:

CREATE TABLE enrollment (

student\_id INT,

course\_id INT,

PRIMARY KEY (student\_id, course\_id)

);

**5. Candidate Key**

* **Definition**: A candidate key is a column or a set of columns that can uniquely identify a row in a table. Each table can have multiple candidate keys, but only one can be selected as the primary key.
* **Characteristics**:
  + Must contain unique values.
  + Cannot contain NULL values.
* **Example**: In a table with columns like employee\_id, email, and phone, all three can be candidate keys if they are unique.

**6. Alternate Key**

* **Definition**: An alternate key is a candidate key that is not selected as the primary key. It can still uniquely identify rows in a table.
* **Example**: Continuing from the previous example, if employee\_id is the primary key, then email and phone can be considered alternate keys.

**7. Surrogate Key**

* **Definition**: A surrogate key is an artificial key (usually a number) that is created to uniquely identify a row in a table. It has no business meaning.
* **Characteristics**:
  + Typically auto-incremented.
  + Often used as a primary key when natural keys are not suitable.
* **Example**:

CREATE TABLE employees (

employee\_id SERIAL PRIMARY KEY,

employee\_name VARCHAR(100)

);

**8. Natural Key**

* **Definition**: A natural key is a key that is formed of attributes that already exist in the real world and are used to uniquely identify an entity.
* **Example**: A social security number (SSN) can serve as a natural key for identifying individuals.

**Summary of Key Differences**

| **Key Type** | **Uniqueness** | **NULLs Allowed** | **Description** |
| --- | --- | --- | --- |
| Primary Key | Unique | No | Uniquely identifies each row in a table. |
| Foreign Key | Not Unique | Yes | References a primary key in another table. |
| Unique Key | Unique | Yes | Ensures all values in a column are unique. |
| Composite Key | Unique | No (in Primary Key) | Combination of two or more columns that is unique. |
| Candidate Key | Unique | No | Can uniquely identify a row; may become a primary key. |
| Alternate Key | Unique | No | A candidate key not selected as the primary key. |
| Surrogate Key | Unique | No | Artificial key, typically auto-generated. |
| Natural Key | Unique | No | Key based on real-world attributes. |

### What is an Index?

An **index** is a database object that improves the speed of data retrieval operations on a database table at the cost of additional storage space. It acts like a reference point, allowing the database engine to find rows quickly, similar to an index in a book that helps locate information faster.

### Benefits of Using Indexes

1. **Faster Query Performance**: Indexes speed up data retrieval operations, making searches and queries more efficient.
2. **Reduced Disk I/O**: With indexes, the database can locate rows without scanning the entire table, reducing the number of disk reads.
3. **Improved Sorting**: Indexes can also help with sorting data, making ORDER BY queries faster.
4. **Unique Constraints**: Indexes can enforce uniqueness on column values, preventing duplicate entries.

### Types of Indexes

1. **Single Column Index**: An index created on a single column of a table.
2. **Composite Index**: An index created on multiple columns of a table. It is useful for queries that filter on multiple columns.
3. **Unique Index**: Ensures that all values in the indexed column(s) are unique.
4. **Clustered Index**: Determines the physical order of data in the table. Each table can have only one clustered index.
5. **Non-Clustered Index**: A separate structure from the data table that allows for quick lookups. A table can have multiple non-clustered indexes.

### How Indexes Work

When a query is executed, the database engine can use an index to quickly locate the requested rows instead of scanning the entire table. It maintains a pointer to the actual rows in the table, which allows for faster access.

### Creating an Index

The basic syntax for creating an index in SQL is:

CREATE INDEX index\_name ON table\_name (column1, column2, ...);

#### Example: Creating an Index

Assume we have a table named employees:

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(100),

department\_id INT,

salary DECIMAL(10, 2)

);

You can create an index on the employee\_name column to speed up searches:

CREATE INDEX idx\_employee\_name ON employees (employee\_name);

### Using Composite Index

If you frequently query by both department\_id and salary, you can create a composite index:

CREATE INDEX idx\_dept\_salary ON employees (department\_id, salary);

### Dropping an Index

To remove an index, you can use the DROP INDEX statement:

DROP INDEX idx\_employee\_name;

### Querying Data with Indexes

#### Example: Query without Index

Without an index, a query may scan the entire table:

SELECT \* FROM employees WHERE employee\_name = 'John Doe';

#### Example: Query with Index

With an index on employee\_name, the database can use the index to locate the row(s) much faster.

### Monitoring Index Usage

Most DBMSs provide tools to monitor index usage. For example, you can check how many times an index is used for query execution and whether it is beneficial.

### Performance Considerations

While indexes improve read performance, they can impact write performance because:

* Every time a record is inserted, updated, or deleted, the associated indexes must also be updated.
* Too many indexes can lead to performance degradation.

### Best Practices for Indexing

1. **Index Selectively**: Create indexes only on columns that are frequently used in WHERE clauses, JOINs, or ORDER BY statements.
2. **Avoid Redundant Indexes**: Ensure that indexes do not duplicate the functionality of others.
3. **Monitor Performance**: Regularly review the performance of queries and the effectiveness of indexes.
4. **Consider Column Order in Composite Indexes**: The order of columns in a composite index matters. Place the most selective columns first.

### Practical Example

1. **Create the employees table**:

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(100),

department\_id INT,

salary DECIMAL(10, 2)

);

1. **Insert sample data**:

INSERT INTO employees (employee\_id, employee\_name, department\_id, salary)

VALUES

(1, 'John Doe', 1, 60000),

(2, 'Jane Smith', 2, 70000),

(3, 'Alice Johnson', 1, 80000),

(4, 'Bob Brown', 3, 55000);

1. **Create an index on the employee\_name column**:

CREATE INDEX idx\_employee\_name ON employees (employee\_name);

1. **Execute a query that benefits from the index**:

SELECT \* FROM employees WHERE employee\_name = 'Alice Johnson';

1. **Check execution plan** (in databases like SQL Server, Oracle, etc.) to see if the index is being used.
2. **Drop the index when no longer needed**:

DROP INDEX idx\_employee\_name;

### Summary

Indexes are essential for optimizing the performance of data retrieval in databases. By understanding how to create and manage indexes, you can significantly improve the efficiency of your SQL queries. Following best practices and monitoring index usage will help maintain a well-optimized database system.

**SQL Views:**

**Definition**: A view in SQL is a virtual table that is based on the result of a SELECT query. It contains rows and columns, just like a real table, but it does not store the data itself. Instead, it presents the data stored in one or more tables through a defined query.

### Key Characteristics of Views

1. **Virtual Table**: A view does not store data physically; it pulls data from the underlying tables each time it is queried.
2. **Simplified Data Access**: Views can simplify complex queries by encapsulating them, allowing users to access the data easily.
3. **Security**: Views can restrict access to specific columns or rows in a table, enhancing security by exposing only the necessary data.
4. **Updatable Views**: In some cases, views can be updated if they meet certain criteria (e.g., if they are based on a single table without any aggregation).
5. **Reusable**: Views can be reused in multiple queries, promoting code reusability and maintainability.

### Creating a View

To create a view, you use the CREATE VIEW statement followed by the view name and the SELECT query that defines the view.

#### Syntax:

CREATE VIEW view\_name AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

### Practical Examples

#### 1. Creating a Simple View

Suppose you have an employees table and you want to create a view that shows only the names and salaries of the employees.

**Table Structure**:

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(100),

department\_id INT,

salary DECIMAL(10, 2)

);

**Insert Sample Data**:

INSERT INTO employees (employee\_id, employee\_name, department\_id, salary) VALUES

(1, 'John Doe', 1, 60000.00),

(2, 'Jane Smith', 2, 70000.00),

(3, 'Alice Johnson', 1, 80000.00),

(4, 'Bob Brown', 3, 55000.00);

**Create the View**:

CREATE VIEW employee\_salaries AS

SELECT employee\_name, salary

FROM employees;

**Query the View**:

SELECT \* FROM employee\_salaries;

**Output**:

| **employee\_name** | **salary** |
| --- | --- |
| John Doe | 60000.00 |
| Jane Smith | 70000.00 |
| Alice Johnson | 80000.00 |
| Bob Brown | 55000.00 |

#### 2. Creating a View with a Filter

You can also create views that filter data. For example, you may want to create a view that shows only employees with a salary greater than 60000.

**Create the View with Filter**:

CREATE VIEW high\_salary\_employees AS

SELECT employee\_name, salary

FROM employees

WHERE salary > 60000;

**Query the View**:

SELECT \* FROM high\_salary\_employees;

**Output**:

| **employee\_name** | **salary** |
| --- | --- |
| Jane Smith | 70000.00 |
| Alice Johnson | 80000.00 |

#### 3. Updating Data Through a View

In some cases, you can update data through a view. Consider a view that allows updates.

**Create an Updatable View**:

CREATE VIEW employee\_update\_view AS

SELECT employee\_id, employee\_name, salary

FROM employees;

**Update a Record Through the View**:

UPDATE employee\_update\_view

SET salary = 75000

WHERE employee\_name = 'Jane Smith';

**Verify the Update**:

SELECT \* FROM employees WHERE employee\_name = 'Jane Smith';

**Output**:

| **employee\_id** | **employee\_name** | **department\_id** | **salary** |
| --- | --- | --- | --- |
| 2 | Jane Smith | 2 | 75000.00 |

#### 4. Dropping a View

If you no longer need a view, you can drop it using the DROP VIEW statement.

**Drop the View**:

DROP VIEW high\_salary\_employees;

### Limitations of Views

1. **Performance**: Views can impact performance since they are re-evaluated each time they are accessed, especially if they involve complex queries.
2. **Non-Updatable Views**: Not all views are updatable. Views that contain aggregate functions, GROUP BY, or JOIN operations on multiple tables are typically not updatable.
3. **Dependency**: If the underlying table structure changes (e.g., columns are renamed or dropped), the view may become invalid, leading to errors when accessed.

### Conclusion

Views are powerful tools in SQL that provide a way to simplify complex queries, enhance security, and facilitate easier access to data. By using views, you can create virtual representations of your data, ensuring that your database interactions remain efficient and secure. Proper use of views can significantly improve data management and access in your applications.

### Subqueries in SQL

#### Definition

A **subquery** is a SQL query nested inside another query (the outer query). It is executed first, and its results are used by the outer query. Subqueries can be used in various clauses, including SELECT, WHERE, FROM, and HAVING.

### Types of Subqueries

1. **Single-row Subquery**
   * Returns a single value (one row, one column).
   * Commonly used with comparison operators (=, <, >, etc.).

**Example:**

SELECT name

FROM employees

WHERE salary = (SELECT MAX(salary) FROM employees);

1. **Multiple-row Subquery**
   * Returns multiple rows (multiple values).
   * Typically used with operators like IN, ANY, or ALL.

**Example:**

SELECT name

FROM employees

WHERE department\_id IN (SELECT department\_id FROM departments WHERE location = 'New York');

1. **Correlated Subquery**
   * References columns from the outer query.
   * Executed once for each row processed by the outer query.

**Example:**

SELECT name, salary

FROM employees e

WHERE salary > (SELECT AVG(salary) FROM employees WHERE department\_id = e.department\_id);

1. **Scalar Subquery**
   * Returns a single value (one row and one column).
   * Can be used in SELECT statements, WHERE clauses, or expressions.

**Example:**

SELECT name,

(SELECT COUNT(\*) FROM orders WHERE customer\_id = c.customer\_id) AS order\_count

FROM customers c;

1. **Nested Subquery**
   * A subquery that is nested inside another subquery.
   * Useful for performing operations that require multiple levels of filtering or aggregation.

**Example:**

SELECT name

FROM employees

WHERE salary > (

SELECT AVG(salary)

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM departments

WHERE location = 'New York'

)

);

### Syntax of Subqueries

The general syntax for a subquery can be outlined as follows:

SELECT column1, column2, ...

FROM table\_name

WHERE column\_name operator (SELECT column\_name FROM table\_name WHERE condition);

You can also use subqueries in the FROM clause:

SELECT \*

FROM (SELECT column1, column2 FROM table\_name WHERE condition) AS alias\_name;

### Practical Examples

#### 1. Basic Subquery in a WHERE Clause

**Objective:** Find employees whose salary is greater than the average salary.

SELECT name, salary

FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees);

#### 2. Subquery in the FROM Clause

**Objective:** Calculate the total salary for employees in each department.

SELECT d.department\_name, dept\_totals.total\_salaries

FROM (

SELECT department\_id, SUM(salary) AS total\_salaries

FROM employees

GROUP BY department\_id

) AS dept\_totals

JOIN departments d ON dept\_totals.department\_id = d.department\_id;

#### 3. Correlated Subquery

**Objective:** Find employees whose salary is above the average salary of their department.

SELECT e.name, e.salary

FROM employees e

WHERE e.salary > (

SELECT AVG(e2.salary)

FROM employees e2

WHERE e2.department\_id = e.department\_id

);

#### 4. Nested Subquery

**Objective:** Retrieve names of employees earning more than the average salary of employees in departments located in 'New York'.

SELECT name

FROM employees

WHERE salary > (

SELECT AVG(salary)

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM departments

WHERE location = 'New York'

)

);

### Differences Between SQL Queries and Subqueries

| **Aspect** | **SQL Query** | **Subquery** |
| --- | --- | --- |
| **Definition** | A SQL query is a standalone command that retrieves or manipulates data. | A subquery is a query nested within another SQL query. |
| **Execution** | Executed independently; returns data directly. | Executed first; its results are used by the outer query. |
| **Use Case** | Used for straightforward data retrieval and manipulation. | Used to perform operations based on the results of another query. |
| **Complexity** | Generally simpler and easier to understand. | Can be complex, especially with nested or correlated subqueries. |
| **Performance** | Can be optimized easily. | May lead to performance issues, especially if correlated. |
| **Placement** | Can stand alone in SELECT, UPDATE, DELETE, etc. | Typically used in SELECT, WHERE, FROM, and HAVING clauses. |
| **Return Type** | Returns a result set directly. | Can return a single value, multiple values, or a result set. |

### Use Cases for Subqueries

* **Filtering Data:** Use subqueries to filter records based on results from another query.
* **Aggregated Data Calculations:** When you need to calculate aggregate values that inform the outer query.
* **Hierarchical Relationships:** Retrieve hierarchical data, such as finding employees under specific managers or departments.
* **Complex Conditions:** Use subqueries to manage complex filtering conditions, especially when they involve multiple steps.

### Advantages of Subqueries

* **Modularity:** Subqueries simplify complex SQL queries into more manageable parts, improving readability.
* **Encapsulation:** Encapsulate logic in subqueries that can be reused in different parts of the outer query.
* **Powerful Filtering:** They provide powerful filtering capabilities, allowing queries to be dynamic based on other query results.

### Disadvantages of Subqueries

* **Performance Issues:** Subqueries, especially correlated subqueries, can lead to performance degradation as they may execute multiple times for each row processed by the outer query.
* **Complexity:** Overusing subqueries can make SQL statements harder to read and understand, especially with multiple nested levels.

### Best Practices for Using Subqueries

* **Consider Using Joins:** If the same results can be achieved with a JOIN, it’s often more efficient and easier to read.
* **Limit Nesting:** Avoid multiple levels of nesting when possible, as it can complicate understanding and debugging the query.
* **Indexing:** Ensure that columns used in subqueries are indexed to improve performance.
* **Test Performance:** Always test the performance of queries that use subqueries compared to alternative methods, such as joins or common table expressions (CTEs).

### Conclusion

Subqueries are a powerful feature in SQL, allowing for complex data retrieval and manipulation. Understanding their structure, use cases, advantages, and potential pitfalls can greatly enhance your ability to write efficient and effective SQL queries. Additionally, recognizing the differences between standard SQL queries and subqueries helps in choosing the right approach for data operations.

### SQL Clauses

SQL clauses are the building blocks of SQL statements. They specify actions to be performed on the database and filter or organize the data retrieved or manipulated. Each clause has its unique purpose and functionality.

### Setting Up a Sample Database

We'll create a database called CompanyDB and a table called Employees to demonstrate various SQL clauses.

#### Step 1: Create Database

CREATE DATABASE CompanyDB;

#### Step 2: Use the Database

USE CompanyDB;

#### Step 3: Create the Employees Table

Here's the SQL statement to create the Employees table:

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

DepartmentID INT,

Salary DECIMAL(10, 2),

HireDate DATE

);

#### Step 4: Insert Sample Data

Let's insert some sample data into the Employees table.

INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID, Salary, HireDate) VALUES

(1, 'John', 'Doe', 10, 50000.00, '2023-01-15'),

(2, 'Jane', 'Smith', 20, 60000.00, '2022-05-20'),

(3, 'Sam', 'Brown', 10, 55000.00, '2021-03-22'),

(4, 'Emily', 'Davis', 30, 45000.00, '2020-11-10'),

(5, 'Michael', 'Johnson', 20, 70000.00, '2019-09-12'),

(6, 'Chris', 'Lee', 30, 48000.00, '2022-02-18'),

(7, 'Sarah', 'Wilson', 10, 51000.00, '2021-12-05');

**1. SELECT Clause**

**Definition:**  
The SELECT clause is used to specify the columns you want to retrieve from a table.

**Working:**  
When you use the SELECT clause, you define which columns to display in the result set. You can retrieve all columns using \* or specify particular columns.

**Example:**

SELECT FirstName, LastName FROM Employees;

* This retrieves the FirstName and LastName of all employees from the Employees table.

**2. WHERE Clause**

**Definition:**  
The WHERE clause is used to filter records based on specific conditions.

**Working:**  
When you apply the WHERE clause, SQL evaluates each row in the table against the condition. Only rows that meet the condition will be included in the result set.

**Example:**

SELECT \* FROM Employees WHERE DepartmentID = 10;

* This retrieves all columns for employees who work in department 10.

**3. ORDER BY Clause**

**Definition:**  
The ORDER BY clause is used to sort the result set by one or more columns.

**Working:**  
You specify the column(s) by which to sort the data, along with the order (ascending or descending). If no order is specified, the default is ascending.

**Example:**

SELECT \* FROM Employees ORDER BY LastName ASC;

* This retrieves all employees sorted alphabetically by their last names in ascending order.

**4. GROUP BY Clause**

**Definition:**  
The GROUP BY clause is used to group rows that have the same values in specified columns into summary rows.

**Working:**  
When you use GROUP BY, SQL consolidates rows that share the same value in the specified column(s). It is commonly used with aggregate functions like COUNT(), SUM(), etc.

**Example:**

SELECT DepartmentID, COUNT(\*) AS EmployeeCount FROM Employees GROUP BY DepartmentID;

* This counts the number of employees in each department.

**5. HAVING Clause**

**Definition:**  
The HAVING clause is used to filter groups created by the GROUP BY clause based on a specified condition.

**Working:**  
HAVING is similar to WHERE, but it operates on groups of rows rather than individual rows. It allows you to apply conditions to aggregate functions.

**Example:**

SELECT DepartmentID, COUNT(\*) AS EmployeeCount

FROM Employees

GROUP BY DepartmentID

HAVING COUNT(\*) > 1;

* This retrieves departments that have more than one employee.

**6. JOIN Clause**

**Definition:**  
The JOIN clause is used to combine rows from two or more tables based on a related column.

**Working:**  
JOIN combines rows from both tables where the specified condition is met. Different types of joins (INNER, LEFT, RIGHT, FULL) determine how unmatched rows are handled.

**Example:**

SELECT e.FirstName, e.LastName, d.DepartmentName

FROM Employees e

JOIN Departments d ON e.DepartmentID = d.DepartmentID;

* This retrieves employee names along with their corresponding department names by matching DepartmentID.

**7. UNION Clause**

**Definition:**  
The UNION clause is used to combine the results of two or more SELECT statements.

**Working:**  
UNION merges the result sets, removing duplicate records. The number and order of columns in each SELECT statement must be the same.

**Example:**

SELECT FirstName, LastName FROM Employees

UNION

SELECT FirstName, LastName FROM Contractors;

* This retrieves a combined list of unique first and last names from both employees and contractors.

**8. INTERSECT Clause**

**Definition:**  
The INTERSECT clause returns only the rows that are common between two SELECT statements.

**Working:**  
INTERSECT identifies matching records in both result sets, displaying only those that appear in both.

**Example:**

SELECT FirstName FROM Employees

INTERSECT

SELECT FirstName FROM Contractors;

* This retrieves the first names that are found in both the employees and contractors tables.

**9. EXCEPT Clause**

**Definition:**  
The EXCEPT clause returns the rows from the first SELECT statement that are not present in the second SELECT statement.

**Working:**  
EXCEPT filters out records found in the second result set from the first, showing only unique records from the first set.

**Example:**

SELECT FirstName, LastName FROM Employees

EXCEPT

SELECT FirstName, LastName FROM Contractors;

* This retrieves employees who are not contractors.

**10. LIMIT Clause**

**Definition:**  
The LIMIT clause is used to specify the maximum number of records to return from a query.

**Working:**  
LIMIT restricts the result set to the specified number of rows, which is useful for pagination.

**Example:**

SELECT \* FROM Employees LIMIT 3;

* This retrieves only the first three employees from the Employees table.

**11. OFFSET Clause**

**Definition:**  
The OFFSET clause is used to skip a specified number of rows before starting to return records.

**Working:**  
When combined with LIMIT, OFFSET allows for more granular control over which records are retrieved, facilitating pagination.

**Example:**

SELECT \* FROM Employees LIMIT 2 OFFSET 2;

* This skips the first two employees and retrieves the next two.

**12. DISTINCT Clause**

**Definition:**  
The DISTINCT clause is used to remove duplicate rows from the result set.

**Working:**  
DISTINCT ensures that each row in the result is unique, filtering out any duplicates from the selected columns.

**Example:**

SELECT DISTINCT DepartmentID FROM Employees;

* This retrieves unique department IDs from the Employees table, excluding duplicates.

### Note:

This detailed guide covers the creation of a sample SQL database, table setup, and practical demonstrations of various SQL clauses. These examples illustrate how to retrieve and manipulate data effectively using SQL.

### SQL Transactions

### ****Definition of a Transaction****

A **transaction** is a sequence of operations that perform as a single logical unit of work. In this case, the transaction will involve transferring money from Amarjeet's account to Hira's account. Transactions ensure that all operations within them are executed successfully; otherwise, they should leave the database unchanged.

### ****ACID Properties of Transactions****

Transactions have four essential properties known as **ACID**:

**1. Atomicity**

**Definition:**  
Atomicity ensures that all operations within a transaction are treated as a single unit. This means that either all operations are completed successfully, or none of them are applied at all. If any part of the transaction fails, the database will roll back to its previous state, leaving it unchanged.

**Real-Time Example:**  
Imagine a banking system where you want to transfer $500 from **Amarjeet**'s account to **Hira**'s account. This transfer involves two operations:

* Deducting $500 from Amarjeet's account.
* Adding $500 to Hira's account.

If the deduction from Amarjeet's account is successful, but the addition to Hira's account fails (e.g., due to a database error), then atomicity ensures that the entire transaction is rolled back. Amarjeet's account remains unchanged, and Hira does not receive any funds.

**SQL Example:**

BEGIN;

UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1; -- Amarjeet

UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 2; -- Hira

COMMIT; -- If both updates succeed

ROLLBACK; -- If any update fails

**2. Consistency**

**Definition:**  
Consistency ensures that a transaction takes the database from one valid state to another, adhering to all defined rules and constraints. This includes maintaining integrity constraints (e.g., foreign keys, unique constraints).

**Real-Time Example:**  
Continuing with the banking example, suppose there is a rule that an account balance cannot go below zero. If Amarjeet tries to transfer $600 while only having $500 in his account, consistency ensures that this transaction will not be allowed, as it violates the business rule.

**SQL Example:**

BEGIN;

UPDATE Accounts SET Balance = Balance - 600 WHERE AccountID = 1; -- Amarjeet

IF (SELECT Balance FROM Accounts WHERE AccountID = 1) < 0 THEN

ROLLBACK; -- Ensures the account does not go negative

ELSE

COMMIT; -- If the account remains valid

END IF;

**3. Isolation**

**Definition:**  
Isolation ensures that transactions do not interfere with each other. The intermediate state of a transaction should not be visible to other transactions until it is committed. This prevents issues like dirty reads (reading uncommitted data) and ensures that each transaction operates as if it is the only transaction running.

**Real-Time Example:**  
Imagine that both Amarjeet and Hira initiate transactions simultaneously to transfer funds. If Amarjeet is transferring $500 and Hira is transferring $300 at the same time, isolation ensures that the operations do not see each other’s intermediate states.

* If Amarjeet's transaction is in progress but not yet committed, Hira's transaction should not see Amarjeet's updated balance until the transaction is complete.

**SQL Example:**

-- Amarjeet's Transaction

BEGIN;

UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1; -- Amarjeet

-- Hira's Transaction

BEGIN;

UPDATE Accounts SET Balance = Balance + 300 WHERE AccountID = 2; -- Hira

COMMIT; -- Hira's transaction can only see Amarjeet's balance after Amarjeet commits

ROLLBACK; -- If needed

**4. Durability**

**Definition:**  
Durability ensures that once a transaction has been committed, its effects are permanent, even in the event of a system failure, such as a crash or power outage. The data must be stored in such a way that it can be recovered.

**Real-Time Example:**  
If Amarjeet successfully transfers $500 to Hira and the system crashes immediately afterward, durability guarantees that Hira's account will reflect the updated balance ($500 more) when the system is restored. The transaction's effects will not be lost.

**SQL Example:**

BEGIN;

UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1; -- Amarjeet

UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 2; -- Hira

COMMIT; -- At this point, even if the system crashes, the changes are durable

**Summary**

* **Atomicity** ensures complete success or failure of transactions.
* **Consistency** guarantees that transactions always transition the database to a valid state.
* **Isolation** protects transactions from interference.
* **Durability** makes certain that committed transactions remain in effect despite failures.

### ****Step 1: Create the Accounts Table****

We start by creating a table called Accounts to hold the account details for Amarjeet and Hira.

CREATE TABLE Accounts (

AccountID INT PRIMARY KEY,

AccountHolder VARCHAR(100),

Balance DECIMAL(10, 2)

);

### ****Step 2: Insert Initial Data****

Next, we insert the initial account balances for Amarjeet and Hira.

INSERT INTO Accounts (AccountID, AccountHolder, Balance) VALUES

(1, 'Amarjeet', 1000.00),

(2, 'Hira', 500.00);

At this point:

* Amarjeet's balance: ₹1000.00
* Hira's balance: ₹500.00

### ****Step 3: Begin the Transaction****

Now we want to transfer ₹200 from Amarjeet’s account to Hira’s account. We will use a transaction to ensure that this process is executed correctly.

#### **SQL Commands for the Transaction**

1. **Start the transaction**.
2. **Subtract ₹200 from Amarjeet's account**.
3. **Add ₹200 to Hira's account**.
4. **Commit the transaction** if both operations succeed, or roll back if any operation fails.

#### **Transaction Implementation**

### **DELIMITER //**

### **CREATE PROCEDURE UpdateAccountBalance()**

### **BEGIN**

### **DECLARE account\_balance DECIMAL(10, 2);**

### **-- Step 1: Get the balance for AccountID 1**

### **SELECT Balance INTO account\_balance FROM Accounts WHERE AccountID = 1;**

### **-- Step 2: Check if the balance is less than 0**

### **IF account\_balance < 0 THEN**

### **-- If not sufficient, rollback**

### **ROLLBACK;**

### **ELSE**

### **-- Step 3: Add ₹200 to Hira's account (AccountID 2)**

### **UPDATE Accounts SET Balance = Balance + 200 WHERE AccountID = 2;**

### **-- Step 4: Commit the transaction**

### **COMMIT;**

### **END IF;**

### **END //**

### **DELIMITER ;**

### **Call this stored procedure**

### **START TRANSACTION;**

### **CALL UpdateAccountBalance();**

### ****Step 4: Explanation of Each SQL Command****

1. **BEGIN**: This command starts a new transaction. All subsequent operations will be part of this transaction until it is either committed or rolled back.
2. **UPDATE Accounts SET Balance = Balance - 200 WHERE AccountID = 1**: This command deducts ₹200 from Amarjeet's account. If Amarjeet’s balance becomes negative, it indicates insufficient funds.
3. **IF (SELECT Balance FROM Accounts WHERE AccountID = 1) < 0 THEN ROLLBACK**: This conditional checks Amarjeet's balance after the deduction. If the balance is less than zero, the transaction is rolled back to avoid overdraft.
4. **UPDATE Accounts SET Balance = Balance + 200 WHERE AccountID = 2**: This command adds ₹200 to Hira's account. This operation only occurs if Amarjeet has sufficient funds.
5. **COMMIT**: If all operations succeed, the changes made during the transaction are saved permanently in the database.

### ****Step 5: Checking Final Balances****

After executing the transaction, you can check the final balances to verify that the money transfer was successful.

SELECT \* FROM Accounts;

**Expected Result:**

* Amarjeet's balance: ₹800.00 (after deduction)
* Hira's balance: ₹700.00 (after addition)

### ****Conclusion****

Using transactions in this manner ensures the integrity and reliability of operations involving multiple steps, like transferring money. The ACID properties help to maintain data consistency, allowing for safe and effective database management.

### ****Error Handling Example****

Suppose we try to transfer ₹1200 from Amarjeet to Hira instead. The transaction would fail:

DELIMITER //

CREATE PROCEDURE TransferFunds()

BEGIN

DECLARE balance DECIMAL(10, 2);

START TRANSACTION;

-- Attempt to subtract ₹1200 from Amarjeet's account

UPDATE Accounts SET Balance = Balance - 1200 WHERE AccountID = 1;

-- Get the current balance

SELECT Balance INTO balance FROM Accounts WHERE AccountID = 1;

-- Check if the balance is sufficient

IF balance < 0 THEN

ROLLBACK; -- If not sufficient, rollback

ELSE

-- Add ₹1200 to Hira's account

UPDATE Accounts SET Balance = Balance + 1200 WHERE AccountID = 2;

COMMIT; -- Commit the transaction

END IF;

END //

DELIMITER ;

All this stored procedure

CALL TransferFunds();

In this case, since Amarjeet's balance would become negative, the ROLLBACK command would be executed, and no changes would occur in the database, keeping the data consistent and accurate.

This example highlights the importance of transactions in ensuring safe and reliable operations in database systems. If you have any more questions or need further clarification, feel free to ask!

## **SQL Injection (SQLI)**

### ****Definition****

**SQL Injection** is a code injection technique that exploits vulnerabilities in an application’s software by injecting malicious SQL statements into an entry field for execution (e.g., to dump the database contents to the attacker). SQL injection allows attackers to bypass application security measures, gain unauthorized access to the database, and manipulate data.

### ****How SQL Injection Works****

1. **User Input Manipulation**: Attackers provide specially crafted input that modifies the intended SQL query. This manipulation can be done through input fields, URL parameters, cookies, etc.
2. **Query Execution**: When the application processes this input without proper validation or sanitization, the injected SQL code is executed by the database, potentially revealing sensitive information or allowing unauthorized operations.
3. **Unauthorized Access**: Attackers can gain access to sensitive data, modify or delete records, execute administrative operations on the database, or even compromise the entire system.

### ****Types of SQL Injection Attacks****

1. **In-band SQL Injection**: The attacker uses the same channel to both launch the attack and gather results. This type can be further categorized into:
   * **Error-based SQL Injection**: Uses error messages thrown by the database to gain insight into the database structure.
   * **Union-based SQL Injection**: Utilizes the UNION SQL operator to combine the results of the original query with results from other queries.
2. **Blind SQL Injection**: The attacker cannot see the results of the query directly but can infer information based on the application’s behavior. It can be:
   * **Boolean-based**: The attacker asks a question that results in a true or false response, allowing them to infer information about the database.
   * **Time-based**: The attacker causes the database to wait (using the SLEEP function) to see how long it takes for a response, which can provide information about the database structure.
3. **Out-of-band SQL Injection**: This technique is used when the attacker cannot use the same channel to launch the attack and gather results. It typically relies on the database's ability to make DNS or HTTP requests to send data to an external server controlled by the attacker.

### ****Real-World Example****

Consider a login form that takes a username and password. If the application uses the following SQL query to validate users:

SELECT \* FROM users WHERE username = '$username' AND password = '$password';

An attacker could input the following as the username:

' OR '1'='1

And leave the password blank:

SELECT \* FROM users WHERE username = '' OR '1'='1' AND password = '';

This results in a query that always returns true, allowing the attacker to bypass authentication and gain access.

### ****Consequences of SQL Injection****

1. **Data Breach**: Unauthorized access to sensitive information, including user data, financial records, or proprietary information.
2. **Data Loss**: Data can be deleted or modified by attackers, leading to corruption or loss of critical information.
3. **Reputation Damage**: Companies suffer significant reputational damage after a data breach, leading to loss of customer trust.
4. **Legal and Regulatory Consequences**: Organizations may face lawsuits or penalties for failing to protect sensitive data.

### ****Prevention Techniques****

1. **Parameterized Queries**: Use prepared statements or parameterized queries, which ensure that user input is treated as data, not executable code.

**Example in Java (using JDBC)**:

String sql = "SELECT \* FROM users WHERE username = ? AND password = ?";

PreparedStatement statement = connection.prepareStatement(sql);

statement.setString(1, username);

statement.setString(2, password);

ResultSet resultSet = statement.executeQuery();

1. **Stored Procedures**: Use stored procedures instead of dynamic SQL queries to separate data from code.
2. **Input Validation**: Validate and sanitize user inputs to ensure they conform to expected formats (e.g., numeric inputs, string lengths).
3. **Least Privilege Principle**: Limit database permissions for the application user to the minimum necessary for the application’s functionality.
4. **Web Application Firewalls (WAF)**: Deploy WAFs to monitor and filter incoming traffic to detect and block SQL injection attempts.
5. **Regular Security Audits**: Perform security assessments and penetration testing to identify and remediate vulnerabilities in applications.
6. **Error Handling**: Implement proper error handling to avoid exposing sensitive information in error messages.

### ****Conclusion****

SQL injection is a serious threat that can compromise the integrity, confidentiality, and availability of data. Understanding how SQL injection works and implementing robust security measures are crucial for protecting applications and databases from such attacks. By following best practices and maintaining security awareness, organizations can significantly reduce the risk of SQL injection vulnerabilities.

**Joins in SQL**

Let’s start by creating the **Students** and **Courses** tables, inserting the provided records, and then performing various types of joins as requested. I’ll explain each step in detail.

### Step 1: Create Tables

#### 1. Create the Students Table

CREATE TABLE Students (

student\_id INT,

student\_name VARCHAR(50),

PRIMARY KEY (student\_id)

);

* **student\_id**: Integer, acts as the primary key (unique identifier).
* **student\_name**: String (up to 50 characters), stores the name of the student.

#### 2. Create the Courses Table

CREATE TABLE Courses (

course\_id INT,

course\_name VARCHAR(50),

student\_id INT,

FOREIGN KEY (student\_id) REFERENCES Students(student\_id)

);

* **course\_id**: Integer, serves as a unique identifier for each course.
* **course\_name**: String (up to 50 characters), stores the name of the course.
* **student\_id**: Integer, acts as a foreign key referencing the student\_id in the Students table.

### Step 2: Insert Data into Tables

#### 1. Insert Data into Students Table

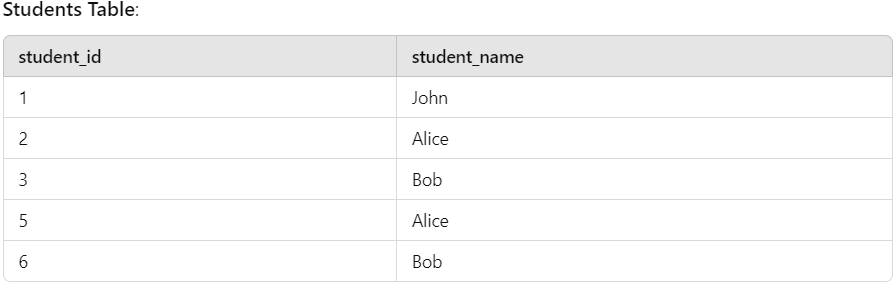
INSERT INTO Students (student\_id, student\_name) VALUES (1, 'John');

INSERT INTO Students (student\_id, student\_name) VALUES (2, 'Alice');

INSERT INTO Students (student\_id, student\_name) VALUES (3, 'Bob');

INSERT INTO Students (student\_id, student\_name) VALUES (5, 'Alice');

INSERT INTO Students (student\_id, student\_name) VALUES (6, 'Bob');



#### 2. Insert Data into Courses Table

INSERT INTO Courses (course\_id, course\_name, student\_id) VALUES (1, 'Mathematics', 1);

INSERT INTO Courses (course\_id, course\_name, student\_id) VALUES (2, 'Science', 2);

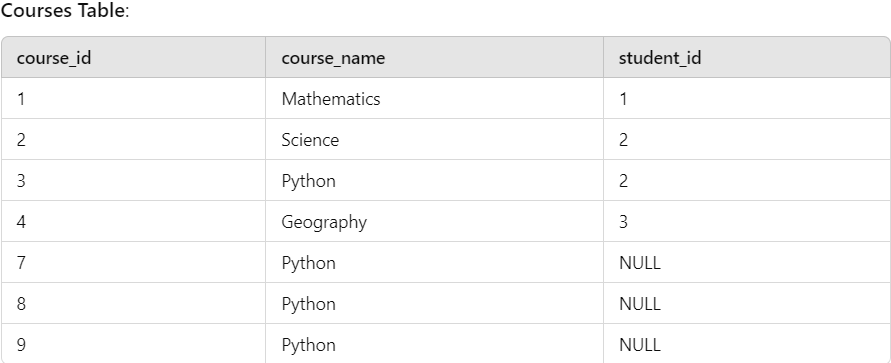
INSERT INTO Courses (course\_id, course\_name, student\_id) VALUES (3, 'Python', 2);

INSERT INTO Courses (course\_id, course\_name, student\_id) VALUES (4, 'Geography', 3);

INSERT INTO Courses (course\_id, course\_name, student\_id) VALUES (7, 'Python', NULL);

INSERT INTO Courses (course\_id, course\_name, student\_id) VALUES (8, 'Python', NULL);

INSERT INTO Courses (course\_id, course\_name, student\_id) VALUES (9, 'Python', NULL);



### 1. ****INNER JOIN****

**Definition**:

* An INNER JOIN returns rows that have matching values in both tables. If there is no match, those rows are excluded from the result.

**Query**:

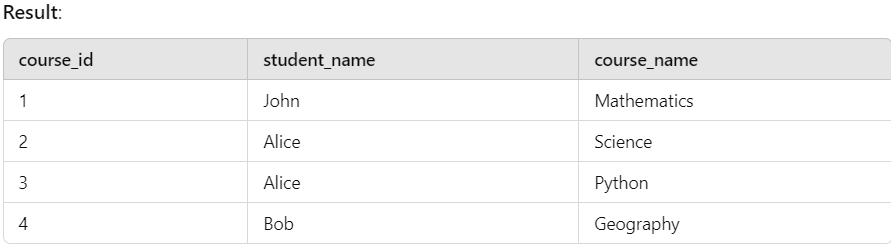
SELECT course\_id, student\_name, course\_name

FROM Students

INNER JOIN Courses ON Students.student\_id = Courses.student\_id;

**Explanation**:

* **SELECT course\_id, student\_name, course\_name**: Specifies the columns to retrieve.
* **FROM Students**: This indicates that the Students table is the primary source of data.
* **INNER JOIN Courses**: This means we're joining the Courses table to the Students table.
* **ON Students.student\_id = Courses.student\_id**: The join condition specifies that the student\_id in both tables must match.



**Practical Breakdown**:

* **Matching Rows**: The only students returned are those who are enrolled in courses (John, Alice, and Bob).
* **Exclusions**: Students with no courses (like student\_id 5 and 6) are excluded.

### 2. ****LEFT JOIN**** (or ****LEFT OUTER JOIN****)

**Definition**:

* A LEFT JOIN returns all rows from the left table (in this case, the students table), along with the matching rows from the right table (courses). If there is no match in the right table, it returns NULL for columns from the right table.

**Query**:

SELECT Courses.course\_id, Students.student\_name, Courses.course\_name

FROM Students

LEFT JOIN Courses ON Students.student\_id = Courses.student\_id;

**Explanation**:

* **LEFT JOIN Courses**: This means we want all records from the Students table, even if there’s no match in Courses.
* The **ON** clause remains the same.

### 

### 3. ****RIGHT JOIN**** (or ****RIGHT OUTER JOIN****)

**Definition**:

* A RIGHT JOIN returns all rows from the right table (in this case, the courses table), along with the matching rows from the left table (students). If there is no match in the left table, it returns NULL for columns from the left table.

**Query**:

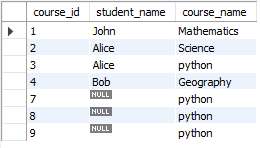
SELECT course\_id, student\_name, course\_name

FROM Students

RIGHT JOIN Courses ON Students.student\_id = Courses.student\_id;

**Explanation**:

* **RIGHT JOIN Courses**: This specifies that all records from the Courses table will be included in the result.
* The **ON** clause remains the same.



**Practical Breakdown**:

* **All Courses**: Every course from the Courses table is returned.
* **NULL Student Names**: Courses with course\_id 7, 8, and 9 do not have corresponding students (as the student\_id is NULL), resulting in NULL values for student\_name.
* **Matching Rows**: Courses that have students enrolled (John, Alice, and Bob) display their respective names.

### 4. ****FULL JOIN**** (or ****FULL OUTER JOIN**** using UNION)

**Definition**:

* A FULL JOIN returns all rows where there is a match in either the left or right table. This means it includes all students and all courses, even if there is no match between them.
* MySQL doesn’t support FULL JOIN directly, but we can simulate it using UNION of the LEFT JOIN and RIGHT JOIN results.

**Query**:

SELECT Students.student\_id, student\_name, course\_name, Courses.course\_id

FROM Students

LEFT JOIN Courses ON Students.student\_id = Courses.student\_id

UNION

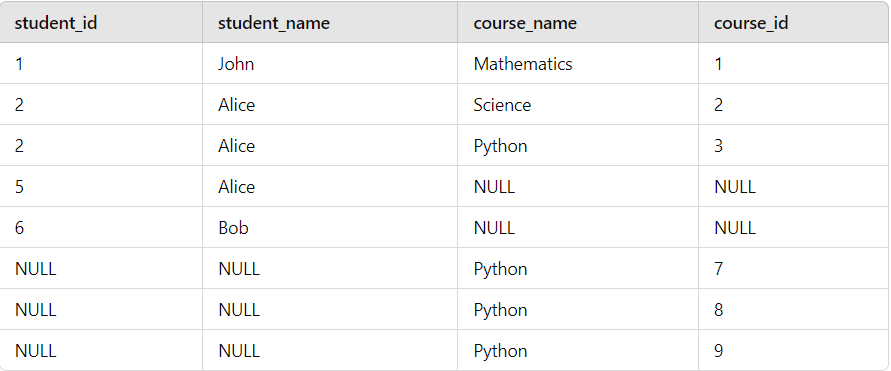
SELECT Students.student\_id, student\_name, course\_name, Courses.course\_id

FROM Students

RIGHT JOIN Courses ON Students.student\_id = Courses.student\_id;

**Explanation**:

* The first part of the query retrieves all students and their courses using a LEFT JOIN.
* The second part retrieves all courses and their corresponding students using a RIGHT JOIN.
* **UNION** combines both results and eliminates duplicates.



**Practical Breakdown**:

* **Combined Results**: This result set includes all unique pairs of students and courses, regardless of whether there is a match.
* **NULL Entries**: Students without courses and courses without students appear with NULL values in the respective columns.

**Q. If I will see only join is written to join two table then which join used how I will know?**

When you encounter the term **"JOIN"** in SQL without any qualifier (like LEFT, RIGHT, or INNER), it typically defaults to **INNER JOIN**. Here's how to identify it:

### Understanding the Default Join

1. **INNER JOIN**:
   * If only **"JOIN"** is specified, it means an **INNER JOIN** is being used.
   * An INNER JOIN returns only the rows that have matching values in both tables. If there is no match, those rows will not be included in the result set.

SELECT Students.student\_id, Students.student\_name, Courses.course\_name

FROM Students

JOIN Courses ON Students.student\_id = Courses.student\_id;

**Q. How I will know which one is right table and left table?**

### Understanding Left and Right Tables

1. **Left Table**:
   * The table that appears **before** the JOIN keyword (in a LEFT JOIN, it is the first table).
   * It is the main table from which you want to retrieve all records, regardless of whether there is a match in the other table.
2. **Right Table**:
   * The table that appears **after** the JOIN keyword (in a LEFT JOIN, it is the second table).
   * It is the table from which you want to retrieve matching records, but you will still get all records from the left table.

**Q. Can we join table without foreign key and with foreign key?**

### 1. ****Using Foreign Keys****

**What Are Foreign Keys?**

* Foreign keys establish a formal relationship between two tables, enforcing referential integrity. This means that a value in one table must match an existing value in another table.

**Benefits of Using Foreign Keys:**

* **Data Integrity**: Ensures that relationships between tables remain consistent. For example, if you have a Students table and a Courses table, a foreign key can ensure that every student\_id in the Courses table corresponds to an existing student\_id in the Students table.
* **Easier Maintenance**: Changes to the related table can be managed better with foreign key constraints, preventing orphaned records.

**Example**:

-- Assuming student\_id in Courses is a foreign key referencing Students

SELECT Students.student\_id, Students.student\_name, Courses.course\_name

FROM Students

JOIN Courses ON Students.student\_id = Courses.student\_id;

### 2. ****Without Using Foreign Keys****

**What If There Are No Foreign Keys?**

* You can still join tables based on common columns that may not be defined as foreign keys. This is often done in scenarios where relationships are logical but not enforced by database constraints.

**Benefits of Not Using Foreign Keys:**

* **Flexibility**: Allows for more complex queries where relationships may not be strictly defined.
* **Performance**: In some cases, avoiding foreign key constraints may lead to performance benefits during data manipulation, as there are no integrity checks enforced by the database.

**Example**:

-- Joining based on common columns, even without foreign keys

SELECT Students.student\_id, Students.student\_name, Courses.course\_name

FROM Students

JOIN Courses ON Students.student\_id = Courses.student\_id;

### Which One to Use?

**When to Use Foreign Keys:**

* Use foreign keys when you have a clear, defined relationship between two tables, and you want to enforce data integrity.
* Ideal for relational databases where maintaining consistent relationships is crucial (e.g., in transaction-heavy applications).

**When to Use Joins Without Foreign Keys:**

* Use joins without foreign keys when you need flexibility in querying data or when the relationships are more loosely defined.
* Suitable for scenarios where data integrity is managed at the application level rather than enforced by the database.

### Best Practices

1. **Define Foreign Keys When Appropriate**: If the relationship between two tables is essential and should be maintained consistently, define foreign keys.
2. **Use Joins Appropriately**: Depending on the type of analysis or data retrieval you need, choose the appropriate type of join (INNER, LEFT, RIGHT, FULL) regardless of foreign key definitions.
3. **Consider Performance**: In large databases, foreign key checks can slow down inserts and updates. If performance is critical, consider how and when to use foreign keys wisely.

### Conclusion

Ultimately, the choice between using joins with foreign keys or without them depends on your specific use case, requirements for data integrity, and the complexity of the relationships between your tables. If the relationships are vital for your application's integrity, prefer foreign keys. If you need flexibility and are managing relationships in your application logic, joining without foreign keys may be appropriate.

**Q. For doing full join how to do in mysql?**

MySQL does not have a built-in **FULL OUTER JOIN** operator. However, you can simulate a full outer join by combining a **LEFT JOIN** and a **RIGHT JOIN** using the **UNION** operator. This approach allows you to get all the records from both tables, including those that do not have matching rows in the other table.

SELECT Students.student\_id, student\_name, course\_name, Courses.course\_id

FROM Students

LEFT JOIN Courses ON Students.student\_id = Courses.student\_id

UNION

SELECT Students.student\_id, student\_name, course\_name, Courses.course\_id

FROM Students

RIGHT JOIN Courses ON Students.student\_id = Courses.student\_id;

**SQL vs. NoSQL**

SQL and NoSQL are two different types of database systems. While SQL databases are relational, NoSQL databases are designed for non-relational data models. Here’s a comparison:

| **Feature** | **SQL (Relational)** | **NoSQL (Non-Relational)** |
| --- | --- | --- |
| **Data Structure** | Uses tables (rows and columns). | Flexible data models: key-value pairs, documents, graphs, or wide-column stores. |
| **Schema** | Schema-based, with a fixed structure. | Schema-less, data can be unstructured or semi-structured. |
| **Query Language** | SQL (Structured Query Language). | Varies by system (e.g., MongoDB uses queries similar to JSON). |
| **Transactions** | ACID-compliant (ensures data integrity). | May not be fully ACID-compliant; instead focuses on eventual consistency. |
| **Scaling** | Vertical scaling (increasing power of the same server). | Horizontal scaling (adding more servers to distribute load). |
| **Use Cases** | Best for structured data and complex queries. | Ideal for unstructured data, high throughput, and real-time web apps. |

**SQL** is best suited for traditional applications like banking systems, HR software, and e-commerce sites where data consistency and relational operations are critical.

**NoSQL** is ideal for applications with large amounts of unstructured or semi-structured data, such as social media, real-time analytics, and IoT applications. Examples of NoSQL databases include MongoDB, Cassandra, and Redis.