**SQL Commands :**

SQL (Structured Query Language) commands are categorized into different types based on their functionality. Here are the main types:

### ****1. Data Definition Language (DDL)****

DDL commands are used to define or modify the structure of a database (e.g., tables, schemas).

|  |  |
| --- | --- |
| **Command** | **Description** |
| CREATE | Creates a new database, table, or object. |
| ALTER | Modifies an existing database object (e.g., table, column). |
| DROP | Deletes a database or table permanently. |
| TRUNCATE | Removes all data from a table without deleting its structure. |
| RENAME | Changes the name of an existing database object. |

### ****2. Data Manipulation Language (DML)****

DML commands handle the manipulation of data within tables.

|  |  |
| --- | --- |
| **Command** | **Description** |
| INSERT | Adds new records (rows) into a table. |
| UPDATE | Modifies existing records in a table. |
| DELETE | Removes specific records from a table. |

### ****3. Data Query Language (DQL/DRL)****

DQL is primarily used to query the database for retrieving specific data.

|  |  |
| --- | --- |
| **Command** | **Description** |
| SELECT | Fetches data from a database. |
| WHERE | Adds conditions to filter records in a query. |

### ****4. Data Control Language (DCL)****

DCL commands manage permissions and access to the database.

|  |  |
| --- | --- |
| **Command** | **Description** |
| GRANT | Grants specific permissions to users. |
| REVOKE | Removes permissions from users. |

### ****5. Transaction Control Language (TCL)****

TCL commands manage transactions in the database.

|  |  |
| --- | --- |
| **Command** | **Description** |
| COMMIT | Saves all changes made in the current transaction. |
| ROLLBACK | Undoes changes made in the current transaction. |
| SAVEPOINT | Sets a point in a transaction to which you can roll back. |
| SET TRANSACTION | Sets properties for a transaction, such as isolation level. |

### ****6. Additional Common Clauses****

These are used along with SQL commands for advanced data handling.

|  |  |
| --- | --- |
| **Command** | **Description** |
| ORDER BY | Sorts the result set in ascending or descending order. |
| GROUP BY | Groups records that have the same values. |
| HAVING | Filters grouped data based on conditions. |
| JOIN | Combines rows from two or more tables based on a related column. |
| LIMIT / TOP | Restricts the number of rows returned in a query result. |

**Examples of SQL Commands:**

1. **CREATE TABLE** (DDL):

|  |
| --- |
| **CREATE TABLE** Employees (  EmployeeID INT PRIMARY KEY,  **Name** VARCHAR(50),  **Department** VARCHAR(50),  **Salary** DECIMAL(10, 2)  ); |

1. **INSERT INTO** (DML):

|  |
| --- |
| **INSERT INTO** Employees (EmployeeID, Name, Department, Salary)  **VALUES** (1, 'John Doe', 'HR', 50000); |

1. **SELECT** (DQL):

|  |
| --- |
| **SELECT** Name, Salary  FROM Employees  **WHERE** Department = 'HR'; |

1. **UPDATE** (DML):

|  |
| --- |
| **UPDATE** Employees  SET Salary = 55000  **WHERE** EmployeeID = 1; |

1. **DELETE** (DML):

|  |
| --- |
| **DELETE** FROM Employees  **WHERE** EmployeeID = 1; |

1. **GRANT** (DCL):

|  |
| --- |
| **GRANT** SELECT, INSERT ON Employees TO user\_name; |

1. **COMMIT** (TCL):

|  |
| --- |
| **COMMIT;** |

In MySQL, constraints are rules applied to table columns to enforce data integrity and ensure valid data entry. When creating tables, you can use the following types of constraints:

**1. NOT NULL**

* **Purpose**: Ensures that a column cannot store NULL values.
* **Use Case**: Use it for mandatory fields such as IDs, names, or any data that must always be provided.
* **Details**:
  + Without NOT NULL, a column can store NULL values, indicating missing or unknown data.
  + This constraint enforces that every row must have a valid value in the column.
* **Example**:

|  |
| --- |
| **CREATE** TABLE users (  id INT **NOT NULL**,  name VARCHAR(50) **NOT NULL**,  email VARCHAR(100)  ); |

In this example:

* id and name cannot be NULL.
* email can store NULL values if no email is provided.

### 2. UNIQUE

* **Purpose**: Ensures that all values in a column or combination of columns are unique across the table.
* **Use Case**: Ideal for fields like email addresses, usernames, or any column requiring unique identification.
* **Details**:
  + A table can have multiple UNIQUE constraints.
  + Duplicate values are not allowed, but NULL values are permitted (only one NULL per column).
* **Example**:

|  |
| --- |
| **CREATE TABLE** employees (  id INT **NOT NULL**,  email VARCHAR(100) **UNIQUE**  ); |

Here, no two rows can have the same email.

**Composite Unique Constraint**:

|  |
| --- |
| **CREATE TABLE** orders (  order\_id INT **NOT NULL**,  product\_id INT **NOT NULL**,  **UNIQUE** (order\_id, product\_id)  ); |

The combination of order\_id and product\_id must be unique.

### 3. PRIMARY KEY

* **Purpose**: Identifies each row in a table uniquely.
* **Use Case**: Suitable for the main identifier of a record, like an id column.
* **Details**:
  + Combines the properties of NOT NULL and UNIQUE.
  + A table can have only one PRIMARY KEY, which can consist of a single column or multiple columns (composite key).
  + Typically used with AUTO\_INCREMENT.
* **Example**:

|  |
| --- |
| **CREATE TABLE** products (  product\_id INT **PRIMARY KEY**,  name VARCHAR(100)  ); |

**Composite Primary Key**:

|  |
| --- |
| **CREATE TABLE** course\_enrollment (  student\_id INT,  course\_id INT,  **PRIMARY KEY** (student\_id, course\_id)  ); |

### 4. FOREIGN KEY

* **Purpose**: Enforces referential integrity by linking a column in one table to the primary key in another table.
* **Use Case**: Useful for relationships between tables, such as orders referencing customers.
* **Details**:
  + Ensures that the value in the foreign key column matches a value in the referenced table or is NULL.
  + Supports cascading actions (ON DELETE CASCADE, ON UPDATE CASCADE).
* **Example**:

|  |
| --- |
| **CREATE TABLE** departments (  id INT **PRIMARY KEY**,  name VARCHAR(50)  );  **CREATE TABLE** employees (  id INT **PRIMARY KEY**,  department\_id INT,  **FOREIGN KEY** (department\_id) **REFERENCES** departments(id)  ); |

department\_id in employees must match an existing id in departments.

**Cascade Actions**:

|  |
| --- |
| **FOREIGN KEY** (department\_id) **REFERENCES** departments(id)  **ON DELETE** CASCADE  **ON UPDATE** CASCADE; |

### 5. CHECK (Introduced in MySQL 8.0.16)

* **Purpose**: Restricts the values that can be stored in a column based on a condition.
* **Use Case**: Validate data at the database level, such as age limits or value ranges.
* **Details**:
  + If a value does not meet the condition, the database rejects the operation.
  + Not supported in earlier MySQL versions (prior to 8.0.16).
* **Example**:

|  |
| --- |
| **CREATE TABLE** employees (  id INT **PRIMARY KEY**,  age INT **CHECK** (age >= 18)  ); |

Ensures age is at least 18.

**Complex Check**:

|  |
| --- |
| **CREATE TABLE** orders (  order\_id INT **PRIMARY KEY**,  quantity INT **CHECK** (quantity > 0 **AND** quantity <= 100)  ); |

### 6. DEFAULT

* **Purpose**: Assigns a default value to a column if no value is provided during insertion.
* **Use Case**: Useful for columns with standard or predictable default values, such as status flags or timestamps.
* **Details**:
  + Saves effort by automatically filling in a value for missing data.
* **Example**:

|  |
| --- |
| **CREATE TABLE** tasks (  id INT **PRIMARY KEY**,  status VARCHAR(20) **DEFAULT** 'pending'  ); |

If no value for status is provided, it defaults to 'pending'.

### 7. AUTO\_INCREMENT

* **Purpose**: Automatically generates a unique number for each new row in a column.
* **Use Case**: Commonly used for primary key columns.
* **Details**:
  + Starts from 1 by default and increments by 1 for each new row.
  + Can be customized using AUTO\_INCREMENT = N in the table definition.
* **Example**:

|  |
| --- |
| **CREATE TABLE** users (  id INT **AUTO\_INCREMENT** **PRIMARY KEY**,  name VARCHAR(50)  ); |

Each new row gets a unique id automatically.

### 8. Composite Keys and Constraints

* **Purpose**: Enforce rules across multiple columns.
* **Use Case**: Ensures data integrity for combined unique identification.
* **Details**:
  + Useful for tables with complex relationships or unique requirements spanning multiple columns.
* **Example**:

|  |
| --- |
| **CREATE TABLE** flight\_bookings (  booking\_id INT **NOT NULL**,  flight\_id INT **NOT NULL**,  passenger\_id INT **NOT NULL**,  **UNIQU**E (flight\_id, passenger\_id)  ); |

Ensures that a passenger cannot book the same flight multiple times.

**Summary Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Constraint** | **Ensures** | **Allow Multiple?** | **Nullable?** | **Example Use Case** |
| NOT NULL | No NULL values | Yes | No | Mandatory fields |
| UNIQUE | Unique values | Yes | Yes | Emails, usernames |
| PRIMARY KEY | Unique & Not Null | No | No | IDs |
| FOREIGN KEY | Referential integrity | Yes | Yes | Parent-child relationships |
| CHECK | Specific condition | Yes | Yes | Age limits, value ranges |
| DEFAULT | Default value for column | Yes | Yes | Status flags, timestamps |
| AUTO\_INCREMENT | Auto-generated unique ID | No | No | IDs |

Constraints ensure data consistency, help enforce business rules, and reduce potential errors in database operations.

In SQL, **COMMIT**, **ROLLBACK**, and **SAVEPOINT** are commands used to manage transactions. Transactions ensure that a group of operations on the database is treated as a single, atomic unit, providing consistency and integrity. Let’s explore each command in detail:

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

* **Purpose**: Saves all the changes made by the current transaction to the database permanently.
* **Use Case**: Use it when you are sure that all operations in a transaction have been executed successfully and should be saved.
* **Key Features**:
  + Once committed, changes cannot be undone.
  + All locks held during the transaction are released.
  + Marks the end of a transaction.
* **Example**:

|  |
| --- |
| **START TRANSACTION**;  **UPDATE** accounts **SET** balance = balance - 500 **WHERE** account\_id = 1;  **UPDATE** accounts **SET** balance = balance + 500 **WHERE** account\_id = 2;  **COMMIT;** |

This ensures that the fund transfer operation is completed and saved.

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

* **Purpose**: Reverts all changes made by the current transaction, restoring the database to its previous state.
* **Use Case**: Use it when an error occurs during a transaction, and you need to undo the partial or full changes.
* **Key Features**:
  + Undoes all changes made since the last COMMIT or START TRANSACTION.
  + Releases locks held during the transaction.
* **Example**:

|  |
| --- |
| **START TRANSACTION;**  **UPDATE** accounts **SET** balance = balance - 500 **WHERE** account\_id = 1;  -- An error occurs here, for example, insufficient balance  **ROLLBACK;** |

This cancels the withdrawal, and no changes are made to the database.

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

* **Purpose**: Creates a point within a transaction to which you can later roll back without undoing the entire transaction.
* **Use Case**: Use it for partial rollbacks within a transaction, especially in complex operations.
* **Key Features**:
  + Allows fine-grained control within a transaction.
  + You can define multiple savepoints and roll back to any of them.
  + Does not end the transaction.
* **Example**:

|  |
| --- |
| **START TRANSACTION;**  **UPDATE** accounts **SET** balance = balance - 500 **WHERE** account\_id = 1;  **SAVEPOINT** sp1; -- Create a savepoint  **UPDATE** accounts **SET** balance = balance + 500 **WHERE** account\_id = 2;  **SAVEPOINT** sp2; -- Create another savepoint  -- If something goes wrong, rollback to sp1  **ROLLBACK TO sp1**;  -- Proceed further if needed  **COMMIT;** |