**Simple syntax**

1. Create database db\_name;
2. Create database if not exists db\_name;
3. Drop database db\_name;
4. Drop database if exists db\_name;
5. Show databases;
6. Show tables;
7. Show tables form db\_name;
8. Use db\_name;
9. Drop table table\_name
10. Alter table table\_name add column\_name datatype constrains;

Select

1. Select \* from table\_name;
2. Select \* from db\_name.table\_name;
3. Select col1, col2 from table\_name;
4. Select distinct column from table\_name;

**Creating Table**

Create table table\_name(

Column\_name datatype constrains,

Column\_name datatype constrains,

…………………………………………………….,

…………………………………………………….

);

Constrains : constrains is just the rules of columns, for example : not null, primary key, unique, foreign key, check, default, auto-increment, etc.

Insert into db\_name.table\_name

Values

(value1, value2, …………….),

(value1, value2, …………….),

(.……………………………………);

**Datatype**

Data types define the data format stored in a database and determine how that data can be used and processed. What Are the Different Types of Data? Different data types have distinct properties with particular usages and possibilities of manipulation. For example, text and numeric have very different properties and require other methods to process and analyze. The SQL data types vary slightly depending on the database management system (DBMS). However, some of the typical data types in SQL include:

* + Numeric: INT, BIGINT, DECIMAL, NUMERIC, FLOAT, REAL, SMALLINT, TINYINT
  + Character: CHAR, VARCHAR, TEXT
  + Date/time: DATE, TIME, DATETIME, TIMESTAMP
* Binary: BINARY, VARBINARY, BLOB
* Other data types: BOOLEAN, ARRAY, JSON, XML

Let’s take a closer look at some of the most common data types you’re likely to encounter as a data analyst. Integer Types Integers store whole numbers that have no fractional component.

* TINYINT: stores values between -128 and 127 (or 0 and 255 if unsigned)
* SMALLINT: stores values between -32,768 and 32,767 (or 0 and 65,535 if unsigned)
* INT: stores values between -2,147,483,648 and 2,147,483,647 (or 0 and 4,294,967,295 if unsigned)
* BIGINT: stores values between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 (or 0 and 18,446,744,073,709,551,615 if unsigned).

Floating-Point Types The floating-point data type store numeric values with a decimal component. It is helpful for scientific and mathematical values that require high precision and a wide range of possible values.

* FLOAT: stores single-precision floating-point values with a precision of 7 digits
* DOUBLE: stores double-precision floating-point values with a precision of 15 digits.

Character Types Characters store strings of text or character data.

* CHAR: stores fixed-length character strings of a specified length
* VARCHAR: stores variable-length character strings of up to a specified length.

Date and Time Types SQL supports several variations of date and time data types, including DATE, TIME, DATETIME, and TIMESTAMP, each with different characteristics and capabilities.

* DATE: YYYY-MM-DD
* TIME: HH:MM:SS
* DATETIME: YYYY-MM-DD HH:MM:SS
* TIMESTAMP: YYYY-MM-DD HH:MM:SS[.fraction] – fraction represents the fractional component which allows for nanosecond-level precision.

Boolean Types Boolean is true/false values for logical comparisons and conditions. Most programming languages and database systems have built-in boolean types, typically represented as TRUE and FALSE.

Signed and Unsigned also available in DBMS

**Keys**

1. Primary key (must have to be unique and not null at the same time)
2. Foreign key (multiple possible, can have duplicate and null values)

Primary Key : It is a column (or set of columns) in a table that uniquely identifies each row. There is only one primary key in a table and it should be not null

Foreign Key : Is a column or set of column in a table that refers to the primary key in another table.

create table table\_name(

column datatype contrains,

primary key (col1, col2),

foreign key (col3, col4) reference table\_name (column\_name, column\_name)

);

|  |  |  |  |
| --- | --- | --- | --- |
| **Id (Primary key)** | **name** | **City\_id (Foreign key)** | **City\_name** |
| 101 | Saquif | 1 | Rangpur |
| 102 | A | 1 | Rangpur |
| 103 | B | 3 | Rajshahi |
| 104 | C | 2 | Dhaka |

|  |  |
| --- | --- |
| **Id (primary key)** | **City\_name** |
| 1 | Rangpur |
| 2 | Dhaka |
| 3 | Rajshahi |
| 4 | Tetulia |

City\_id in first table is foreign key (primary key of second table).

**Operators (not completed reading)**

In Database Management Systems (DBMS), operators are used to perform various operations on data. These operators can be classified into several types depending on the operations they perform. Here's an overview of the most common types of operators used in DBMS:

**1. Relational Operators**

These operators are used to operate on relations (tables) and derive new relations. They form the backbone of relational algebra.

* **Selection (σ)**: Used to select rows that satisfy a given condition.
  + Example: σ (age > 30) (Employee) selects employees whose age is greater than 30.
* **Projection (π)**: Used to select specific columns from a table.
  + Example: π (name, age) (Employee) returns only the name and age columns.
* **Union (∪)**: Combines tuples from two relations and removes duplicates.
  + Example: Employee ∪ Manager combines employees and managers.
* **Set Difference (-)**: Returns tuples from one relation that are not in another relation.
  + Example: Employee - Manager returns employees who are not managers.
* **Cartesian Product (×)**: Combines every tuple of one relation with every tuple of another relation.
  + Example: Employee × Department combines each employee with each department.
* **Intersection (∩)**: Returns common tuples between two relations.
  + Example: Employee ∩ Manager returns employees who are also managers.
* **Join (⨝)**: Combines related tuples from two relations based on a common attribute.
  + **Inner Join**: Combines rows that have matching values in both tables.
  + **Outer Join**: Includes rows that do not have a match in one or both tables (Left, Right, Full Outer Joins).
* **Division (÷)**: Used when querying for "for all" type queries.
  + Example: Find employees who work on all projects.

**2. Comparison Operators**

Used in queries to compare values. These are similar to comparison operators in programming languages.

* = : Equal to
* != : Not equal to
* < : Less than
* > : Greater than
* <= : Less than or equal to
* >= : Greater than or equal to

**3. Logical Operators**

Used to combine multiple conditions in SQL queries.

* **AND**: True if all conditions are true.
* **OR**: True if any one of the conditions is true.
* **NOT**: Negates the result of the condition.

**4. Arithmetic Operators**

These operators are used to perform arithmetic operations on numeric data.

* + : Addition
* - : Subtraction
* \* : Multiplication
* / : Division
* % : Modulus (returns the remainder of a division)

**5. Aggregate Operators (Functions)**

Aggregate functions perform a calculation on a set of values and return a single value.

* **SUM()**: Adds up all values in a column.
* **COUNT()**: Returns the number of rows.
* **AVG()**: Returns the average of a numeric column.
* **MAX()**: Returns the largest value in a column.
* **MIN()**: Returns the smallest value in a column.

**6. String Operators**

Operators used to perform operations on strings in queries.

* **LIKE**: Used for pattern matching in SQL queries.
  + Example: SELECT \* FROM Employee WHERE name LIKE 'J%' returns names that start with "J".
* **CONCAT()**: Concatenates two or more strings.

**7. Assignment Operator**

* \*\*=`: Assigns values to a variable or a column. Used in SQL to set values.
  + Example: SET salary = 5000 WHERE id = 1;

Clauses in DBMS

# 1. SELECT Clause

The SELECT clause is used to specify the columns to be retrieved from the database. It is one of the most important clauses as it determines what data will be displayed after the query is executed.

Syntax: SELECT column1, column2 FROM table\_name;

# 2. FROM Clause

The FROM clause is used to specify the table from which the data will be retrieved. It is a required clause in every SELECT statement.

Syntax: SELECT column1, column2 FROM table\_name;

# 3. WHERE Clause

The WHERE clause is used to filter records. It specifies a condition that must be satisfied for a record to be included in the results.

Syntax: SELECT column1, column2 FROM table\_name WHERE condition;

# 4. GROUP BY Clause

The GROUP BY clause groups rows that have the same values in specified columns into summary rows, often used with aggregate functions like COUNT, MAX, MIN, SUM, and AVG.

Syntax: SELECT column1, aggregate\_function(column2) FROM table\_name GROUP BY column1;

# 5. HAVING Clause

The HAVING clause is used to filter groups based on a condition, often used after the GROUP BY clause. It is similar to the WHERE clause but operates on aggregated data.

Syntax: SELECT column1, aggregate\_function(column2) FROM table\_name GROUP BY column1 HAVING condition;

# 6. ORDER BY Clause

The ORDER BY clause is used to sort the result set in either ascending or descending order based on one or more columns.

Syntax: SELECT column1, column2 FROM table\_name ORDER BY column1 [ASC|DESC];

# 7. JOIN Clause

The JOIN clause is used to combine rows from two or more tables, based on a related column between them. There are different types of joins including INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN.

Syntax: SELECT column1, column2 FROM table1 JOIN table2 ON table1.common\_column = table2.common\_column;

# 8. UNION Clause

The UNION clause is used to combine the result sets of two or more SELECT queries. It removes duplicate rows between the queries, while UNION ALL includes all duplicates.

Syntax: SELECT column1 FROM table1 UNION SELECT column1 FROM table2;

# 9. DISTINCT Clause

The DISTINCT clause is used to remove duplicate records from the result set, returning only unique values.

Syntax: SELECT DISTINCT column1 FROM table\_name;

# 10. LIMIT Clause

The LIMIT clause is used to restrict the number of rows returned in a query result. It is helpful in cases where you only need to display a certain number of rows.

Syntax: SELECT column1, column2 FROM table\_name LIMIT number\_of\_rows;

# 11. EXISTS Clause

The EXISTS clause is used to check whether the result of a subquery returns any rows. It returns TRUE if the subquery returns one or more records, otherwise it returns FALSE.

Syntax: SELECT column1 FROM table\_name WHERE EXISTS (subquery);

# 12. IN Clause

The IN clause is used to filter records that match any value in a given list. It is often used in combination with the WHERE clause.

Syntax: SELECT column1, column2 FROM table\_name WHERE column1 IN (value1, value2, ...);

# 13. BETWEEN Clause

The BETWEEN clause is used to filter records that fall within a specific range of values.

Syntax: SELECT column1, column2 FROM table\_name WHERE column1 BETWEEN value1 AND value2;

# 14. AS Clause

Renames a column or table with an alias.

SELECT column AS alias FROM table\_name;

Time : 01:42 date: 15/10/2024

**Practice Question :** Write the query to find avg marks in each city in ascending order.

**Solution :** select city, avg(marks) from student group by city order by city;

🡺 select city, count(rollno) from student group by city having max(marks) > 90;

🡺 select city from student where grade = "A" group by city;

🡺 select city from student where grade = "A" group by city having max(marks) > 90;

Table related Queries

**Update:** In MySQL, the UPDATE query is used to modify existing records in a table. You can update one or more fields in one or multiple rows. The basic syntax for an UPDATE query is:

UPDATE table\_name

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

WHERE condition;

Update table\_name set col1 = val1, col2 = val2 where condition;

Example : update student set grade = "A" where grade = "0";

update student set marks = 82 where rollno = 105;

update student set grade = "B" where marks between 80 and 89;

update student set marks = marks+1;

Time : 1:56

**Delete:** delete some existing rows.

Delete from table\_name where condition;

Revisiting Foreign Keys

Table Name : Department

|  |  |
| --- | --- |
| Id (primary key) | name |
| 101 | Science |
| 102 | English |
| 103 | Bangla |

Table Name : Teacher

|  |  |  |
| --- | --- | --- |
| Id (primary key) | name | dept\_id (Foreign Key :: Dept\_id) |
| 101 | Alice | 101 |
| 102 | Bob | 103 |
| 103 | Casey | 102 |
| 104 | Donald | 102 |

Time : 2:00

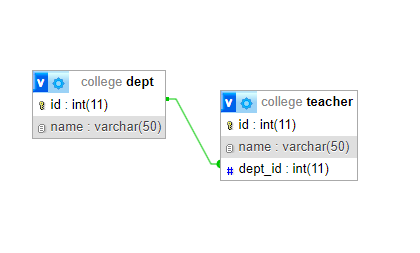


Figure : EER Diagram

Parent table : Department and child table : Teacher table

**Parent Table**

* The **parent table** is the table that contains the **primary key** that will be referenced by another table (the child table).
* It is considered the "source" of the relationship. Its data is referred to by other tables, but it does not depend on any other table for its existence.

**Child Table**

* The **child table** is the table that contains the **foreign key**. This foreign key references the primary key in the parent table.
* It is considered dependent on the parent table because it needs the foreign key to maintain the relationship. The child table relies on the parent table for referential integrity.

Cascading for Foreign Key in Databases

# 1. Introduction to Foreign Keys

In relational databases, a foreign key is a field or a group of fields in one table that uniquely identifies a row of another table. The table containing the foreign key is called the child table, while the table it refers to is called the parent table. A foreign key enforces referential integrity between two tables by ensuring that a record in the child table cannot exist unless it corresponds to an existing record in the parent table.

# 2. Cascading in Foreign Keys

Cascading is a feature available in foreign key constraints that helps maintain referential integrity when a record in the parent table is modified (updated or deleted). Depending on the type of cascading action defined, certain operations on the parent table can automatically trigger changes in the child table.

## 2.1 Cascade DELETE

When a foreign key is set with the ON DELETE CASCADE option, deleting a record in the parent table automatically deletes all related records in the child table. This ensures that orphaned records do not remain in the child table when the corresponding record in the parent table is deleted.  
For example:  
- If you delete a customer from the Customers table (parent), all of their associated orders in the Orders table (child) will be deleted.

## 2.2 Cascade UPDATE

When a foreign key is set with the ON UPDATE CASCADE option, updating a record in the parent table automatically updates the corresponding foreign key values in the child table. This ensures that the relationship remains intact even when the primary key value is modified in the parent table.  
For example:  
- If you change the primary key of a customer in the Customers table (parent), the corresponding foreign key values in the Orders table (child) will be updated automatically.

# 3. Practical Examples of Cascading Actions

Let’s consider a practical example with two tables: Customers (parent) and Orders (child).

CREATE TABLE Customers (  
 customer\_id INT PRIMARY KEY,  
 customer\_name VARCHAR(100)  
);  
  
CREATE TABLE Orders (  
 order\_id INT PRIMARY KEY,  
 order\_date DATE,  
 customer\_id INT,  
 FOREIGN KEY (customer\_id) REFERENCES Customers(customer\_id)

ON DELETE CASCADE

ON UPDATE CASCADE  
);

# 4. Benefits and Risks of Using Cascading

Cascading can be highly beneficial for maintaining database integrity, but it can also introduce certain risks if not used properly.

## 4.1 Benefits

- \*\*Automatic Maintenance\*\*: Cascading ensures that child records are automatically updated or deleted, reducing manual intervention.  
- \*\*Data Integrity\*\*: Helps maintain referential integrity and prevents orphaned records.

## 4.2 Risks

- \*\*Unintentional Data Loss\*\*: If cascading DELETE is used improperly, it could lead to unintentional deletion of a large number of related records.  
- \*\*Complexity\*\*: Cascading actions can make the database harder to understand, especially for new developers.

# 5. Best Practices for Using Cascading

- Always evaluate whether cascading actions are necessary for your specific use case.  
- Use cascading DELETE sparingly, especially in systems where preserving historical data is important.  
- Clearly document cascading actions to help future developers understand the implications.

Lecture on ALTER Statement in SQL

# 1. Introduction to the ALTER Statement

The SQL 'ALTER' statement is used to make changes to the structure of an existing database object. Most commonly, it is used to modify tables (ALTER TABLE) and databases (ALTER DATABASE). With 'ALTER', you can add, modify, rename, or drop columns in tables and rename tables themselves.

# 2. Using ALTER TABLE

The 'ALTER TABLE' statement is used to modify the structure of an existing table. This can include:

* - Adding new columns  
  - Modifying the datatype of existing columns  
  - Dropping (removing) columns  
  - Renaming columns  
  - Renaming tables

## Adding Columns

To add a new column to an existing table, you use the following syntax:  
```  
ALTER TABLE table\_name ADD column\_name datatype;  
```  
For example, to add an 'email' column of type VARCHAR(255) to a 'users' table, you would run:  
```  
ALTER TABLE users ADD email VARCHAR(255);  
```

## Modifying Columns

To change the data type of an existing column, you use the following syntax:  
```  
ALTER TABLE table\_name MODIFY column\_name datatype;  
```  
For example, to modify the 'phone' column in the 'users' table to hold VARCHAR(20):  
```  
ALTER TABLE users MODIFY phone VARCHAR(20);  
```

## Dropping Columns

To drop a column from a table, use the following syntax:  
```  
ALTER TABLE table\_name DROP COLUMN column\_name;  
```  
For example, to remove the 'address' column from the 'users' table:  
```  
ALTER TABLE users DROP COLUMN address;  
```

## Renaming Columns

To rename a column, you use the following syntax:  
```  
ALTER TABLE table\_name RENAME COLUMN old\_column\_name TO new\_column\_name;  
```  
For example, to rename the 'username' column in the 'users' table to 'user\_name':  
```  
ALTER TABLE users RENAME COLUMN username TO user\_name;  
```

## Renaming Tables

To rename a table, you use the following syntax:  
```  
ALTER TABLE old\_table\_name RENAME TO new\_table\_name;  
```  
For example, to rename the 'users' table to 'app\_users':  
```  
ALTER TABLE users RENAME TO app\_users;  
```

# 3. Using ALTER DATABASE

The 'ALTER DATABASE' statement is used to modify a database's characteristics, such as changing its name or settings.

To change the name of a database, use:  
```  
ALTER DATABASE old\_db\_name MODIFY NAME = new\_db\_name;  
```  
For example:  
```  
ALTER DATABASE mydb MODIFY NAME = newdb;  
```

# 4. Other ALTER Commands

Apart from altering tables and databases, the 'ALTER' command can be used to change other database objects, such as views, indexes, and triggers, depending on the database management system (DBMS) in use.

# 5. Conclusion

The 'ALTER' statement is a powerful tool for database administration, allowing changes to be made without needing to drop and recreate objects. It is essential to use 'ALTER' carefully, especially when dropping columns or making changes to key parts of the database structure.

TRUNCATE in SQL

In SQL, the TRUNCATE statement is used to remove all rows from a table. It is an efficient way to delete all data in a table while maintaining the structure of the table (i.e., its columns, indexes, etc.). The key difference between TRUNCATE and other deletion methods (like DELETE) lies in how they operate and their impact on performance, logging, and transactional control.

# 1. Syntax

TRUNCATE TABLE table\_name;

# 2. Key Features of TRUNCATE

- Fast Operation: TRUNCATE is faster than the DELETE statement when removing all rows from a table.   
- Resets Identity Columns: TRUNCATE resets the counter for an IDENTITY column, unlike DELETE.  
- Non-transactional in some DBMS: In some databases, TRUNCATE cannot be rolled back.  
- Frees Space: TRUNCATE releases storage space occupied by data, depending on the DBMS.

# 3. Difference Between TRUNCATE and DELETE

|  |  |  |
| --- | --- | --- |
| Feature | TRUNCATE | DELETE |
| Command Type | DDL (Data Definition Language) | DML (Data Manipulation Language) |
| Removes all rows? | Yes | Yes (with DELETE FROM table\_name) |
| Can filter rows? | No | Yes (can use a WHERE clause) |
| Transaction control? | Non-transactional in some DBMS | Transactional (can be rolled back) |
| Frees storage? | Yes | No |
| Resets identity values? | Yes | No |
| Logging | Minimal logging (faster) | Fully logged (slower for large tables) |

# 4. Usage Scenarios

- When to Use TRUNCATE:  
 - If you need to remove all data from a table and reset any auto-increment values.  
 - When you want a fast operation to clear a table without triggering any DELETE triggers.  
 - When rolling back the transaction is not required.  
  
- When Not to Use TRUNCATE:  
 - If you need to selectively remove data.  
 - When DELETE triggers must be fired.  
 - When you need to perform this operation in a transactional way.

# 5. Examples

Suppose you have a table `students`:  
  
```sql  
SELECT \* FROM students;  
```  
Running the TRUNCATE statement:  
```sql  
TRUNCATE TABLE students;  
```  
After truncation, if you query the table again, you will get an empty result set.

# 6. Limitations of TRUNCATE

- Cannot Be Used on Tables Involved in Foreign Key Constraints.  
- Cannot Use on Views.  
- Triggers Are Not Fired.

# 7. TRUNCATE vs DROP

- TRUNCATE only removes the data in the table but keeps the structure.  
- DROP removes both the data and the table itself.

# 8. Performance Considerations

- Speed: TRUNCATE is generally faster than DELETE for large datasets.  
- Minimal Locking: TRUNCATE locks the table rather than each row, making it faster.