PRASAD V POTLURI SIDDHARTHA INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
UNIT-3

Subject Name: **Database Management Systems**
Subject Code: **20CS3402**

Syllabus:

Relational Model: Introduction to relational model, concepts of domain, attribute, tuple, relation, importance of null values, constraints (Domain, Key constraints, integrity constraints) and their importance, Relational Algebra **Basic SQL:** Simple Database schema, data types, table definitions (create, alter), different DML operations (insert, delete, update). SQL querying using where clause, arithmetic & logical operations, SQL functions (Date and Time, Numeric, String). Creating tables with relationship, implementation of key and integrity constraints, nested queries, sub queries, grouping, aggregation, ordering, implementation of different types of joins, views, relational set operations.

Introduction to relational model:

The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd 1970), and it attracted immediate attention due to its simplicity and mathematical foundation.

The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values or, to some extent, a flat file of records. It is called a flat file because each record has a simple linear or flat structure.

When a relation is thought of as a table of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row.

Example: In STUDENT relation because each row represents facts about a particular student entity. The column names Name, Student_ number, Class, and Major specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a tuple, a column header is called an attribute, and the table is called a relation. The data type describing the types of values that can appear in each column is represented by a domain of possible values.

concepts of domain, attribute, tuple, relation:

Domain:

A **domain** D is a set of atomic values. By **atomic** we mean that each value in the domain is invisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the

domain are drawn. It is also useful to specify the name for the domain, to help in interpreting its values.

Some examples of domains follow:

- Usa_phone_numbers: The set of ten-difgit phone numbers valid in United States.
- Social_security_numbers: The set of valid nine-digit social security numbers.
- Names: The set of character strings that represents the names of persons.
- Employee_ages: Possible ages of employees in a company; each must be an integer value between 15 and 80.

Attribute:

An **attribute** Ai is the name of a role played by some domain D in the relation schema R. D is called the domain of Ai and is denoted by dom(Ai).

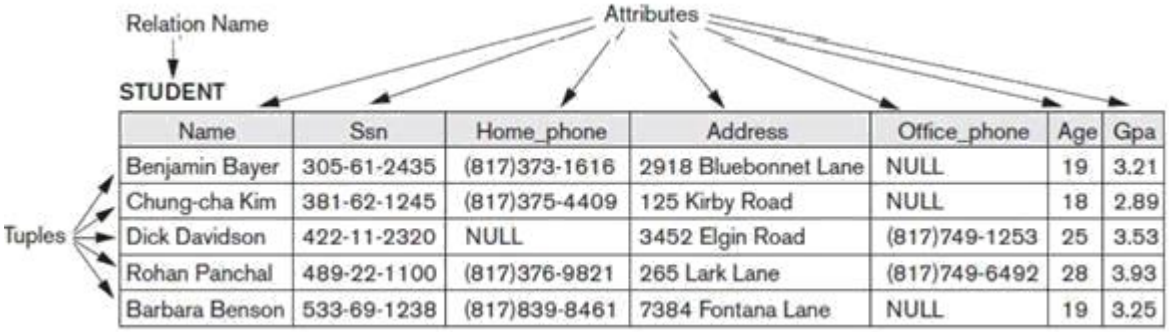
Tuple:

Mapping from attributes to values drawn from the respective domains of those attributes. Tuples are intended to describe some entity (or relationship between entities) in the miniworld **Example:** a tuple for a PERSON entity might be

{ Name → ”smith”, Gender→Male, Age →25 }

Relation:

A named set of tuples all of the same form i.e., having the same set of attributes.



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25

Importance of null values:

In a Database Management System (DBMS), a null value represents missing, unknown, or inapplicable data. Unlike zero or an empty string, a null value explicitly indicates that no value is assigned to a particular field.

constraints (Domain, Key constraints, integrity constraints) and their importance:

Domain Constraints

Definition:

Domain constraints define the **permissible values** for a column (attribute) in a table. Each column has a data type that restricts the values it can store.

Example:

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT CHECK (age > 18 AND age < 65),
    salary DECIMAL(10,2)
);
```

Here, the age column must be **greater than 18 and less than 65**, ensuring that only valid employee ages are stored.

Importance of Domain Constraints:

- Prevents **invalid data** entry (e.g., entering text in a numeric field).
- Ensures **data accuracy** by restricting values to a valid range.
- Reduces **data inconsistencies** and errors.

Key Constraints

Definition:

Key constraints ensure **uniqueness** and **uniquely identify** records in a table. The main types of key constraints are:

a) Primary Key (PK)

- Ensures **uniqueness** and **non-null values** in a column.
- A table can have **only one primary key**.
- Example:

```
CREATE TABLE Students (student_id INT PRIMARY KEY,name VARCHAR(50),email VARCHAR(100));
```

- Each student_id must be unique and cannot be null.

b) Unique Key

- Ensures column values are **unique** but allows nulls.
- Example:

CREATE TABLE Users (user_id INT PRIMARY KEY, username VARCHAR(50) UNIQUE);

- No two users can have the same username.

c) Foreign Key (FK)

- Establishes a **relationship between tables** by enforcing **referential integrity**.
- Example:

```
CREATE TABLE Orders (
   order_id INT PRIMARY KEY,
   customer_id INT,
   FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

- Prevents deleting a customer_id in the **Customers** table if it exists in **Orders**.

Importance of Key Constraints:

- Prevents **duplicate records**.
- Ensures **data relationships** remain valid.
- Supports **data consistency** across tables.

3. Integrity Constraints

Definition:

Integrity constraints ensure that **data is accurate and consistent** throughout the database.

Types of Integrity Constraints:

a) Entity Integrity

- Ensures that **every table has a unique identifier (Primary Key)** and that it cannot be NULL.
- Example:

CREATE TABLE Employees (emp_id INT PRIMARY KEY, emp_name VARCHAR(50));

- Every employee must have a unique emp_id.

b) Referential Integrity

- Ensures that **foreign keys** reference valid primary keys in another table.
- Prevents **orphan records** (records without valid references).
- Example:

```
CREATE TABLE Orders (
```

order_id INT PRIMARY KEY,

customer_id INT,

FOREIGN KEY (customer_id) REFERENCES Customers(customer_id) ON DELETE CASCADE);

- If a **customer is deleted**, all their orders are also deleted.

c) Check Constraint

- Enforces a condition on a column.
- Example:

CREATE TABLE Products (product_id INT PRIMARY KEY, price DECIMAL(10,2) CHECK (price > 0));

- Ensures that the price is always **greater than zero**.

d) Not Null Constraint

- Ensures that a column **cannot contain NULL values**.
- Example:

CREATE TABLE Employees (emp_id INT PRIMARY KEY, emp_name VARCHAR(50) NOT NULL);

- Every employee must have a name.

Importance of Integrity Constraints:

- Prevents **orphan records** and maintains valid relationships.
- Ensures **data accuracy** and prevents invalid entries.
- Maintains **data reliability and consistency** across tables.

Relational Algebra :

Relational Algebra is a procedural query language used in DBMS to retrieve data from relational databases. It consists of a set of operations that take one or more relations (tables) as input and produce a new relation as output.

Types of Relational Algebra Operations

Relational Algebra operations are categorized into two types:

A. Basic (Fundamental) Operations

1. **Selection (σ) – Filters Rows**

- Used to retrieve specific **rows** that satisfy a given condition.
- **Symbol**: σ (sigma)
- **Syntax**:

σ condition(Relation)

Example:

SELECT * FROM Employees WHERE department = 'CSE';

Relational Algebra:

σ department=′CSE′(Employees)

2. **Projection (π) – Filters Columns**

- Retrieves specific **columns** from a table.
- **Symbol**: π (pi)
- **Syntax**:

Π column1 , column2,...(Relation)

Example:

SELECT name, salary FROM Employees;

Relational Algebra:

Π name, salary(Employees)

3. **Cartesian Product (×) → Combining Tables Without Condition**

Relational Algebra Notation:

R×S

Example:
Get all possible combinations of customers and orders tables.

SELECT * FROM customers, orders;

4. **Natural Join**

Relational Algebra Notation:

R⋈S

SQL Equivalent:

SELECT * FROM R NATURAL JOIN S;

5. **Union (∪) → Combining Results**

Relational Algebra Notation:

R∪S

SQL Equivalent:

SELECT column1, column2 FROM R

UNION

SELECT column1, column2 FROM S;

Basic SQL:

Simple Database schema:

A database schema defines the structure of a database, including tables, columns, data types, and relationships. Below is a basic schema for an Employee Management System using SQL.

Creating a Simple Database Schema

Tables in the Schema

1. **Employee** – Stores employee details.
2. **Department** – Stores department details.
3. **Project** – Stores project details.
4. **Works_On** – Tracks which employees work on which projects.

SQL Schema Definition

Employee Table

CREATE TABLE Employee (Emp_ID INT PRIMARY KEY, Name VARCHAR(50) NOT NULL, Age INT, Salary DECIMAL(10,2), Dept_ID INT,FOREIGN KEY (Dept_ID) REFERENCES Department(Dept_ID));

Department Table

CREATE TABLE Department (Dept_ID INT PRIMARY KEY, Dept_Name VARCHAR(50) UNIQUE NOT NULL);

Project Table

CREATE TABLE Project (Proj_ID INT PRIMARY KEY, Proj_Name VARCHAR(100) NOT NULL, Budget DECIMAL(12,2));

Works_On Table (Many-to-Many Relationship)

CREATE TABLE Works_On (Emp_ID INT, Proj_ID INT, Hours_Worked INT, PRIMARY KEY (Emp_ID, Proj_ID),

FOREIGN KEY (Emp_ID) REFERENCES Employee(Emp_ID),

FOREIGN KEY (Proj_ID) REFERENCES Project(Proj_ID));

Data types:

Data types are used to represent the nature of the data that can be stored in the database table. Data types mainly classified into three categories for every database.

- String Data types
- Numeric Data types

- Date and time Data types

Data Types in MySQL:

1. Numeric Types:
    - INT (Integer): Used for storing whole numbers.
    - DECIMAL or NUMERIC: Used for storing fixed-point numbers.
    - FLOAT and DOUBLE: Used for storing floating-point numbers.
2. String Types:
    - CHAR and VARCHAR: Used for storing character strings. CHAR has a fixed length, while VARCHAR has a variable length.
    - TEXT: Used for large text data.
3. Date and Time Types:
    - DATE: Used for storing dates in the format 'YYYY-MM-DD'.
    - TIME: Used for storing times in the format 'HH:MM:SS'.
    - DATETIME: Used for storing both date and time in the format 'YYYY-MM-DD HH:MM:SS'.
    - TIMESTAMP: Similar to DATETIME, but often used to store the current timestamp.

Table definitions (create, alter):

Data Definition Language (DDL)

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

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

a. CREATE It is used to create a new table in the database.
Syntax:

CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);

Example:

CREATE TABLE EMPLOYEE(Name VARCHAR(20), Email VARCHAR (100), DOB DATE);

b. DROP: It is used to delete both the structure and record stored in the table.
Syntax

DROP TABLE table_name;

Example

DROP TABLE EMPLOYEE;

c. ALTER: It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.
Syntax:
To add a new column in the table

ALTER TABLE table_name ADD column_name COLUMN-definition;

To modify existing column in the table:

ALTER TABLE table_name RENAME (column_definitions....);

To rename existing column name in the table:

ALTER TABLE table_name RENAME old_column_name to new-column_name;

EXAMPLE

ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR(20));

ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR(20));

d. TRUNCATE: It is used to delete all the rows from the table and free the space containing the table.
Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE TABLE EMPLOYEE;

Different DML operations (insert, delete, update):

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

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

a. INSERT: The INSERT statement is a SQL query. It is used to insert data into the row of a table.
Syntax:

INSERT INTO TABLE_NAME

(col1, col2, col3,.... col N)

VALUES (value1, value2, value3, ... valueN);

Or

INSERT INTO TABLE_NAME

VALUES (value1, value2, value3, ... valueN);

For example:

INSERT INTO students (Name, Subject) VALUES ("Sonoo", "DBMS");

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

Syntax:

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

HERE CONDITION]

For example:

UPDATE students

SET User_Name = 'Sonoo'

WHERE Student_Id = '3'

c. DELETE: It is used to remove one or more rows from a table.

Syntax:

DELETE FROM table_name [WHERE condition];

For example:

DELETE FROM EMPLOYEE

WHERE Name="Sonoo";

SQL querying using where clause:

i. SELECT statement with where clause

SELECT statement with where clause syntax

SELECT column1, column2, ...FROM table_name WHERE condition;

ii. SELECT statement with WHERE clause using Comparison Operators.

In MySQL, and many other SQL databases, you can use various comparison operators in the **WHERE** clause to filter rows based on specific conditions. Here are some common comparison operators:

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

- >=: Greater than or equal to

Equal to (=):

Used to check if two expressions are equal.

SELECT * FROM table_name WHERE column_name = value;

Not equal to (!= or <>):

Used to check if two expressions are not equal.

SELECT * FROM table_name WHERE column_name != value;

or

SELECT * FROM table_name WHERE column_name <> value;

Greater than (>):

Used to check if one expression is greater than another.

SELECT * FROM table_name WHERE column_name > value;

Less than (<):

Used to check if one expression is less than another.

SELECT * FROM table_name WHERE column_name < value;

Greater than or equal to (>=):

Used to check if one expression is greater than or equal to another.

SELECT * FROM table_name WHERE column_name >= value;

Less than or equal to (<=):

Used to check if one expression is less than or equal to another.

SELECT * FROM table_name WHERE column_name <= value;

IS NULL:

Used to check if a column contains a NULL value.

SELECT * FROM table_name WHERE column_name IS NULL;

IS NOT NULL:

Used to check if a column does not contain a NULL value.

SELECT * FROM table_name WHERE column_name IS NOT NULL;

iii. SELECT statement with WHERE clause using AND , OR ,NOT.

a. WHERE clause with AND.

SELECT column-names

FROM table-name

WHERE condition1 AND condition2

b. WHERE clause with OR

SELECT column1, column2, ...

FROM table_name

WHERE condition1 OR condition2;

c. WHERE clause with NOT

SELECT column1, column2, ...

FROM table_name

WHERE NOT condition;

d. SELECT statement with WHERE Clause Using IN operator

The **IN** operator in a **Where** clause is used to specify a range for multiple values in a column. It allows you to match a value against a set of values. Here's the syntax for a **SELECT** statement with the **IN** operator:

SELECT column1, column2, ...FROM table_name WHERE column_name IN (value1, value2, ...);

Example:

SELECT employee_id, first_name, last_name, department

FROM employees

WHERE department IN ('Sales', 'Marketing');

e. SELECT statement with WHERE clause using BETWEEN Operator

The **BETWEEN** operator in a **WHERE** clause is used to filter rows based on a range of values. It's commonly used with numeric, date, or timestamp columns. Here's the syntax for a **SELECT** statement with the **BETWEEN** operator:

Syntax:

SELECT column1, column2, ...

FROM table_name

WHERE column_name BETWEEN value1 AND value2;

Example:

SELECT product_id, product_name, price

FROM products

WHERE price BETWEEN 50 AND 100;

f. SELECT statement with WHERE Clause using LIKE operator.

The **like** operator in a **where** clause is used to search for a specified pattern in a column. It is often used with wildcard characters such as **%** (percent) and **_** (underscore). Here's the syntax for a **Select** statement with a **where** clause using the **like** operator:

SELECT column1, column2, ...

FROM table_name

WHERE column_name LIKE pattern;

Example:

SELECT product_id, product_name

FROM products

WHERE product_name LIKE '_Laptop_';

Arithmetic & logical operations:

Arithmetic Operations:

Arithmetic operations allow mathematical computations in SQL queries.

Operators & Usage:

Operator	Description	Example
+	Addition	SELECT 10 + 5; → 15
-	Subtraction	SELECT 10 - 5; → 5
*	Multiplication	SELECT 10 * 5; → 50

Operator	Description	Example
/	Division	SELECT 10 / 5; → 2
%	Modulus (Remainder)	SELECT 10 % 3; → 1

Example in a Query:

SELECT product_name, price, price * 1.1 AS increased_price

FROM products;

2. Logical Operations in MySQL

Logical operators help in filtering and combining conditions in SQL queries.

Operators & Usage:

Operator	Description	Example
AND	Returns TRUE if both conditions are true	SELECT * FROM users WHERE age > 18 AND city = 'New York';
OR	Returns TRUE if at least one condition is true	SELECT * FROM users WHERE age > 18 OR city = 'New York';
NOT	Negates a condition	SELECT * FROM users WHERE NOT city = 'New York';
XOR	Returns TRUE if only one condition is true (but not both)	SELECT * FROM users WHERE age > 18 XOR city = 'New York';

Example Query:

SELECT * FROM employees

WHERE salary > 50000 AND department = 'IT';

SQL functions (Date and Time, Numeric, String):

1. String Functions:

a**. concat()**: Concatenates two or more strings.

select concat (‘string1’,’srting2’);

b. **lpad():** Pad a string on the left side.

select lpad (string, length, pad_string);

c. **rpad():** Pad a string on the right side.

select rpad (string, length, pad_string);

d**. ltrim():** Removes left side spaces from a string.

select ltrim(‘ string’);

e. **rtrim():** Removes right side spaces from a string.

select rtrim(‘string ’);

f. **lower():**Converts a string to lowercase.

select lower(‘string’);

g. **upper():** Converts a string to uppercase.

select upper(‘string’);

h**. initcap():** Capitalizes the first letter of each word in a string.

select initcap(‘string’);

i.**length():**Returns the number of characters in a string.

select length(‘string’);

j.**substr():** Extracts a substring from a string.

select substr(string, start_position, length)

k.**instr():**Returns the position of the first occurrence of a substring in a string.

select instr(string, substring)

2. **Numeric Functions**:

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

SELECT ABS (-10) AS absolute_value;

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

SELECT ROUND (3.14159, 2) AS rounded_number;

c.**greatest():** Returns the greatest value from the list of values.

Select greatest(3,1,12,5,-4,7);

d.**least():** Returns the least value from the list of values.

Select least(3,1,12,5,-4,7);

e. **TRUNC():**Truncates a number to a specified decimal precision.

SELECT TRUNCATE(123.4567, 2) AS truncated_number;

3. **Date Functions**:

a**. CURDATE ()**: Returns the current date.

SELECT CURDATE ();

b. **DATEDIFF ()**: Calculates the difference between two dates.

SELECT DATEDIFF ('2024-02-08', '2024-01-01') AS days_difference;

c. **DATE_FORMAT ()**: Formats a date as specified.

SELECT DATE_FORMAT ('2025-01-27', '%W, %M , %Y') AS formatted_date;
d.**SYSDATE():**Returns the current system date and time.

SELECT SYSDATE();

e.**NEXT_DAY**():Returns the date of the next occurrence of a specified day of the week. Select NEXT_DAY(date, 'day');

f. **TIMESTAMPDIFF ()**: Calculates the difference between two timestamps.

SELECT TIMESTAMPDIFF (MINUTE, '2024-02-08 12:00:00', '2024-02-08 12:30:00') AS minutes_difference;

g**.TIME_FORMAT ()**: Formats a time as specified.

SELECT TIME_FORMAT ('12:30:00', '%h:%i :%p') AS formatted_time;

h**. add_months():**Adds a specific number of months to a date.

SELECT DATE_ADD('2025-01-25', INTERVAL 3 MONTH) AS add_months;

i**.LAST_DAY():**Returns the last day of the month for a given date.

SELECT LAST_DAY('2025-01-25') AS last_day;

j**. MONTHS_BETWEEN():**Calculates the number of months between two dates.

k. **TO_CHAR():** Converts a value to a string in a specified format.

SELECT DATE_FORMAT(NOW(), '%Y-%m-%d %H:%i:%s') AS formatted_date;

l. **TO_DATE():**Converts a string to a date in a specified format.

SELECT STR_TO_DATE('25-01-2025', '%d-%m-%Y') AS to_date;

Creating tables with relationship:

relationships between tables are defined using Primary Keys (PKs) and Foreign Keys (FKs). These relationships help maintain data integrity.

Types of Table Relationships

One-to-One (1:1)

- Each row in Table A maps to only one row in Table B.
- Example: Person ↔ Passport

Explanation: A person can have only one passport.

```
CREATE TABLE Person (
   person_id INT PRIMARY KEY,
   person_name VARCHAR(100) NOT NULL );
```

```
CREATE TABLE Passport (
    passport_id INT PRIMARY KEY,
    person_id INT UNIQUE,
    issue_date DATE,
    FOREIGN KEY (person_id) REFERENCES Person(person_id) ON DELETE CASCADE
);
```

One-to-Many (1:M)

- One row in Table A maps to multiple rows in Table B.
- Example: Department ↔ Employees

  Explanation :A department has many employees.

```
CREATE TABLE Departments (
  dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100) NOT NULL
);
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id) ON DELETE SET NULL
);
```

Many-to-Many (M:N)

- Many rows in Table A map to many rows in Table B.
- Requires a junction table.
- Example: Students ↔ Courses (via Enrollment table)

Explanation: A student can enroll in multiple courses, and a course can have multiple students.This requires an Enrollment table.

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL
);
```

```
CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100) NOT NULL
);

CREATE TABLE Enrollment (
    student_id INT,
    course_id INT,
    enrollment_date DATE,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES Students(student_id) ON DELETE CASCADE,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id) ON DELETE CASCADE
);
```

Implementation of key and integrity constraints:

Constraints in SQL

Constraints in SQL means we are applying certain conditions or restrictions on the database. This further means that before inserting data into the database, we are checking for some conditions. If the condition we have applied to the database holds true for the data which is to be inserted, then only the data will be inserted into the database tables.

Constraints in SQL can be categorized into two types:

1. **ColumnLevelConstraint:**
   Column-level constraint is used to apply a constraint on a single column.
2. **TableLevelConstraint:**
   Table Level Constraint is used to apply a constraint on multiple columns.

Constraints available in SQL are:

1. NOT NULL
2. UNIQUE
3. PRIMARY KEY
4. FOREIGN KEY
5. CHECK

1. NOT NULL

- NULL means empty, i.e., the value is not available.

- Whenever a table's column is declared as NOT NULL, then the value for that column cannot be empty for any of the table's records.
- There must exist a value in the column to which the NOT NULL constraint is applied.

NOTE: NULL does not mean zero. NULL means empty column, not even zero.

NOT NULL:
Syntax to apply the NOT NULL constraint during table creation:
Query 1:

```
CREATE TABLE student
(StudentID INT NOT NULL,
Student_FirstName VARCHAR (20),
Student_LastName VARCHAR (20),
Student_PhoneNumber VARCHAR (20),
Student_Email_ID VARCHAR (40)
);
```

Output:
Table Created

Query 2:
To verify that the not null constraint is applied to the table's column and the student table is created successfully, we will execute the following query:

DESC student;
Output:

Field	Type	Null	Key	Default	Extra
StudentID	int	NO		NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	

Query 3:
Syntax to apply the NOT NULL constraint on an existing table's column:
Syntax:
ALTER TABLE TableName CHANGE Old_ColumnName New_ColumnName Datatype NOT NULL;

Example:
Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply a NOT NULL constraint to one of the table's column. Then we will execute the following query:

ALTER TABLE student CHANGE StudentID StudentID INT NOT NULL;

2. UNIQUE

- Duplicate values are not allowed in the columns to which the UNIQUE constraint is applied.
- The column with the unique constraint will always contain a unique value.
- This constraint can be applied to one or more than one column of a table, which means more than one unique constraint can exist on a single table.
- Using the UNIQUE constraint, you can also modify the already created tables.

Syntax to apply the UNIQUE constraint on a single column:

```
CREATE TABLE TableName
(ColumnName1 datatype UNIQUE,
ColumnName2 datatype….,
ColumnNameN datatype
);
```

Example:

Create a student table and apply a UNIQUE constraint on one of the table's column while creating a table.

Case 1:

```
DROP TABLE STUDENT;

SHOW TABLES;

CREATE TABLE student

(StudentID INT UNIQUE,

Student_FirstName VARCHAR(20),

Student_LastName VARCHAR (20),

 Student_PhoneNumber VARCHAR (20),

Student_Email_ID VARCHAR (40)

);
```

Case 2:

ALTER TABLE student CHANGE StudentID StudentID INT UNIQUE;

DESC student;

Field	Type	Null	Key	Default	Extra
StudentID	int	YES	UNI	NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	

Syntax to apply the UNIQUE constraint on more than one column:

CREATE TABLE TableName

(ColumnName1 datatype,

ColumnName2 datatype,….,

ColumnNameN datatype,

UNIQUE (ColumnName1, ColumnName 2, ColumnNameN));

Example:

Create a student table and apply a UNIQUE constraint on more than one table's column while creating a table.

```
CREATE TABLE student
(StudentID INT,
Student_FirstName VARCHAR(20),
Student_LastName VARCHAR(20),
Student_PhoneNumber VARCHAR(20),
Student_Email_ID VARCHAR(40),
UNIQUE(StudentID, Student_PhoneNumber));
```

To verify that the unique constraint is applied to more than one table's column and the student table is created successfully, we will execute the following query:
DESC STUDENT;

Output:

Field	Type	Null	Key	Default	Extra
StudentID	int	YES	MUL	NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	

Syntax to apply the UNIQUE constraint on an existing table's column:

Syntax:

ALTER TABLE TableName ADD UNIQUE (ColumnName);

Example:

Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply a UNIQUE constraint to one of the table's column. Then we will execute the following query:

ALTER TABLE student ADD UNIQUE (StudentID);

DESC STUDENT;

Field	Type	Null	Key	Default	Extra
StudentID	int	YES	MUL	NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	

3. PRIMARY KEY

- PRIMARY KEY Constraint is a combination of NOT NULL and Unique constraints.
- NOT NULL constraint and a UNIQUE constraint together forms a PRIMARY constraint.
- The column to which we have applied the primary constraint will always contain a unique value and will not allow null values.

Syntax of primary key constraint during table creation:

CREATE TABLE TableName

(ColumnName1 datatype PRIMARY KEY,

ColumnName2 datatype,….,

ColumnNameN datatype);

Example:

Create a student table and apply the PRIMARY KEY constraint while creating a table.

CREATE TABLE student
(StudentID INT PRIMARY KEY,
Student_FirstName VARCHAR (20),

Student_LastName VARCHAR (20),
Student_PhoneNumber VARCHAR (20),
Student_Email_ID VARCHAR (40)
);

To verify that the primary key constraint is applied to the table's column and the student table is created successfully, we will execute the following query:

DESC STUDENT;

Field	Type	Null	Key	Default	Extra
StudentID	int	NO	PRI	NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	

Syntax to apply the primary key constraint on an existing table's column:

Syntax:

ALTER TABLE TableName **ADD PRIMARY KEY** (ColumnName);

Example:

Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply the PRIMARY KEY constraint to the table's column. Then we will execute the following query:

ALTER TABLE STUDENT DROP COLUMN STUDENTID;

ALTER TABLE STUDENT ADD COLUMN STUDENTID INT;

Field	Type	Null	Key	Default	Extra
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	
STUDENTID	int	YES		NULL	

ALTER TABLE student **ADD PRIMARY KEY** (StudentID);

To verify that the primary key constraint is applied to the student table's column, we will execute the following query:

DESC STUDENT;

Field	Type	Null	Key	Default	Extra
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	
STUDENTID	int	NO	PRI	NULL	

4. FOREIGN KEY

- A foreign key is used for referential integrity.
- When we have two tables, and one table takes reference from another table, i.e., the same column is present in both the tables and that column acts as a primary key in one table. That particular column will act as a foreign key in another table.

Syntax to apply a foreign key constraint during table creation:

CREATE TABLE tablename

(ColumnName1 Datatype(**SIZE**) **PRIMARY KEY**,

ColumnNameN Datatype(**SIZE**),

FOREIGN KEY(ColumnName) **REFERENCES** PARENT_TABLE_NAME(Primary_Key_ColumnName));

Example:

Create an employee table and apply the FOREIGN KEY constraint while creating a table.

To create a foreign key on any table, first, we need to create a primary key on a table.

Example:

Create an employee table and apply the FOREIGN KEY constraint while creating a table.

To create a foreign key on any table, first, we need to create a primary key on a table.

CREATE TABLE employee

(Emp_ID **INT** NOT NULL **PRIMARY KEY**,

Emp_Name **VARCHAR** (40),

Emp_Salary **VARCHAR** (40));

To verify that the primary key constraint is applied to the employee table's column, we will execute the following query:

DESC STUDENT;

Field	Type	Null	Key	Default	Extra
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	
STUDENTID	int	NO	PRI	NULL	

Now, we will write a query to apply a foreign key on the department table referring to the primary key of the employee table, i.e., Emp_ID.

CREATE TABLE department

(Dept_ID INT NOT NULL PRIMARY KEY,

Dept_Name VARCHAR (40),

Emp_ID INT NOT NULL,

CONSTRAINT emp_id_fk FOREIGN KEY(Emp_ID) REFERENCES employee (Emp_ID));

To verify that the foreign key constraint is applied to the department table's column, we will execute the following query:

DESC department;

Field	Type	Null	Key	Default	Extra
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	
STUDENTID	int	NO	PRI	NULL	

Syntax to apply the foreign key constraint with constraint name:

CREATE TABLE tablename

(ColumnName1 Datatype **PRIMARY KEY**,

ColumnNameN Datatype (**SIZE**),

CONSTRAINT ConstraintName **FOREIGN KEY**(ColumnName) **REFERENCES** PARE NT_TABLE_NAME(Primary_Key_ColumnName));

Example:

Create an employee table and apply the FOREIGN KEY constraint with a constraint name while creating a table.

To create a foreign key on any table, first, we need to create a primary key on a table.

NOTE:

DROP DEPARTMENT;

DROP EMPLOYEE;

CREATE TABLE employee (Emp_ID INT NOT NULL PRIMARY KEY, Emp_Name VARCHAR (40), Emp_Salary VARCHAR (40));

	Field	Type	Null	Key	Default	Extra
▶	Emp_ID	int	NO	PRI	NULL	
	Emp_Name	varchar(40)	YES		NULL	
	Emp_Salary	varchar(40)	YES		NULL	

Syntax to apply the foreign key constraint on an existing table's column:

ALTER TABLE Parent_TableName **ADD FOREIGN KEY** (ColumnName) **REFERENCES** Child_TableName (ColumnName));

Example:

Consider we have an existing table employee and department. Later, we decided to apply a FOREIGN KEY constraint to the department table's column. Then we will execute the following query:

ALTER TABLE department ADD FOREIGN KEY (Emp_ID) REFERENCES employee (Emp_ID);

	Field	Type	Null	Key	Default	Extra
▶	Emp_ID	int	NO	PRI	NULL	
	Emp_Name	varchar(40)	YES		NULL	
	Emp_Salary	varchar(40)	YES		NULL	

5. CHECK

- Whenever a check constraint is applied to the table's column, and the user wants to insert the value in it, then the value will first be checked for certain conditions before inserting the value into that column.
- **For example:** if we have an age column in a table, then the user will insert any value of his choice. The user will also enter even a negative value or any other invalid

value. But, if the user has applied check constraint on the age column with the condition age greater than 18. Then in such cases, even if a user tries to insert an invalid value such as zero or any other value less than 18, then the age column will not accept that value and will not allow the user to insert it due to the application of check constraint on the age column.

Syntax to apply check constraint on a single column:

CREATE TABLE TableName

(ColumnName1 datatype **CHECK** (ColumnName1 Condition),

ColumnName2 datatype,….,

ColumnNameN datatype)

);

Example:

Create a student table and apply CHECK constraint to check for the age less than or equal to 15 while creating a table.

DROP TABLE STUDENT;

CREATE TABLE student

(StudentID INT,

Student_FirstName VARCHAR(20),

Student_LastName VARCHAR(20),

Student_PhoneNumber VARCHAR(20),

Student_Email_ID VARCHAR(40),

Age INT CHECK(Age <= 15));

DESC STUDENT;

Output:

Field	Type	Null	Key	Default	Extra
StudentID	int	YES		NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	
Age	int	YES		NULL	

Syntax to apply check constraint on multiple columns:

CREATE TABLE TableName

(ColumnName1 datatype,

ColumnName2 datatype **CHECK** (ColumnName1 Condition AND ColumnName2 Condition),…., ColumnNameN datatype));

Example:

Create a student table and apply CHECK constraint to check for the age less than or equal to 15 and a percentage greater than 85 while creating a table.

```
DROP TABLE STUDENT;

CREATE TABLE student

(StudentID INT,

Student_FirstName VARCHAR(20),

Student_LastName VARCHAR(20),

Student_PhoneNumber VARCHAR(20),

Student_Email_ID VARCHAR(40),

Age INT, Percentage INT,

CHECK( Age <= 15 AND Percentage > 85)

);

DESC STUDENT;
```

Field	Type	Null	Key	Default	Extra
StudentID	int	YES		NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	
Age	int	YES		NULL	
Percentage	int	YES		NULL	

Syntax to apply check constraint on an existing table's column:

ALTER TABLE TableName **ADD CHECK** (ColumnName Condition);

Example:

Consider we have an existing table student. Later, we decided to apply the CHECK constraint on the student table's column. Then we will execute the following query:

ALTER TABLE student ADD CHECK (Age <=15);

To verify that the check constraint is applied to the student table's column, we will execute the following query:

DESC STUDENT;

Field	Type	Null	Key	Default	Extra
StudentID	int	YES		NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	
Age	int	YES		NULL	
Percentage	int	YES		NULL	

Nested queries and sub queries:

IN OPERATOR:

The IN operator is used to check if a value matches any value in a list or the result of a subquery.

Syntax:

SELECT column_name

FROM table_name

WHERE column_name IN (SELECT column_name FROM another_table WHERE condition);

ANY OPERATOR:

ANY operator is used to compare a value with a set of values returned by a subquery.

Syntax:

SELECT column_name

FROM table_name

WHERE column_name operator ANY (SELECT column_name FROM another_table WHERE condition);

ALL OPERATOR:

All operator is used to compare a value with all values returned by a subquery.

Syntax:

SELECT column_name

FROM table_name

WHERE column_name operator ALL (SELECT column_name FROM another_table WHERE condition);

EXISTS Operator :

The EXISTS operator tests for existence of rows in the results set of the subquery.

Syntax:

SELECT column_name

FROM table_name

WHERE EXISTS (SELECT * FROM another_table WHERE condition);

NOT EXISTS:

The NOT EXISTS operator is used to test for the absence of rows in a sub query result set.

Syntax:

SELECT column_name

FROM table_name

WHERE NOT EXISTS (SELECT * FROM another_table WHERE condition);

UNION

UNION combines the result sets of two or more queries and removes duplicates.

SELECT column_name FROM table_name WHERE condition;

Union

SELECT column_name FROM table_name WHERE condition;

INTERSECT

The **INTERSECT** operation in SQL returns only the **common records** between two queries.

SELECT column_name FROM table_name WHERE condition;

intersect

SELECT column_name FROM table_name WHERE condition

Grouping:

The GROUP BY Clause is used in SQL queries to organize data that have the same attribute values.

Syntax:

SELECT column_lists,
FROM table_name
GROUP BY column_lists;

HAVING Clause

HAVING clause in MySQL **used in conjunction with GROUP BY** clause enables us to specify conditions that filter which group results appear in the result. It returns only those values from the groups in the final result that fulfills certain conditions.

Syntax:

SELECT column_lists, aggregate_function (expression)
FROM table_name
GROUP BY column_lists
HAVING condition;

Aggregation:

Definition: Aggregate functions in SQL are unique functions that work on a group of rows in a table and produce a single value as a result.

1. Count()
2. Sum()

3. Avg()
4. Min()
5. Max()

1. COUNT():
This function returns the number of records(rows) in a table.
Syntax:

SELECT COUNT(column_name) **FROM** table_name;

SELECT COUNT(*) **FROM** Employee;

2. SUM():
This function returns the sum of all values of a column in a table.
Syntax:

SELECT SUM(column_name) **FROM** table_name;

3. AVG()
This function will return the average of all values present in a column**.**
Syntax:

SELECT AVG(column_name) **FROM** table_name;

4. MIN():
This function produces the lowest value in a column for a group of rows that satisfy a given criterion.
Syntax:

SELECT MIN(column_name) **FROM** table_name;

5. MAX()
The MAX function in SQL is used to return the highest value in a column for a group of rows that satisfy a given condition in a table.
Syntax:

SELECT MAX(column_name) **FROM** table_name;

Ordering: ORDER BY clause (sort by column name)

The order by clause in a select statement is used to sort the result set based on one or more columns. Here's the basic syntax:

SELECT column1, column2, ...

FROM table_name

ORDER BY column1 [ASC | DESC], column2 [ASC | DESC], ...;

Example:

SELECT first_name, last_name, salary

FROM employees

ORDER BY salary DESC;

Limit Clause:

The limit clause in a select statement is used to restrict the number of rows returned by a query. It is often used in combination with the order by clause to get a specific subset of rows based on a certain order. Here's the basic syntax:

SELECT column1, column2, ...

FROM table_name

ORDER BY column1 [ASC | DESC], column2 [ASC | DESC], ...

LIMIT number_of_rows;

Example:

SELECT first_name, last_name, salary

FROM employees

ORDER BY salary DESC

Implementation of different types of joins:

Definition: **Joins** used to retrieve data from multiple tables based on a related column.

Types of Join

1.Inner join

2. outer join (It can be categorized into three categories 1.left out join,2.right outer join,3. Full outer join)

INNER JOIN – Returns matching records from both tables.

Syntax:

select emp1.eid,emp1.name,dept1.dname

from emp1 inner join dept1 using(did);

LEFT JOIN (LEFT OUTER JOIN) – Returns all records from the left table and matching records from the right.

Syntax:

select emp1.eid,emp1.name,dept1.dname

from emp1 left outer join dept1 using(did);

RIGHT JOIN (RIGHT OUTER JOIN) – Returns all records from the right table and matching records from the left.

Syntax:

select emp1.eid,emp1.name,dept1.dname

from emp1 right outer join dept1 using(did);

FULL JOIN (FULL OUTER JOIN) – Not natively supported in MySQL, but can be simulated using UNION.

Syntax:

select emp1.name,dept1.dname

from emp1 full join dept1 on emp1.did=dept1.did;

CROSS JOIN – Returns the Cartesian product of two tables.

Syntax:

select emp1.eid,emp1.name,dept1.dname

from emp1 cross join dept1;

SELF JOIN – A table joins itself.

Syntax:

select emp1.eid,emp1.name,dept1.dname

from emp1 natural join dept1;

views:

views - creating and Updating views:

A **view** is a virtual table in MySQL that provides a way to represent the result of a query as if it were a table. Views do not store data themselves but retrieve data from the underlying tables.

Creating a View

The CREATE VIEW statement is used to create a view.

Syntax:

CREATE VIEW view_name AS

SELECT column1, column2, ...

FROM table_name

WHERE condition;

Updating a View:

If you need to modify a view, you can use **ALTER VIEW** or **CREATE OR REPLACE VIEW**.

Using ALTER VIEW:

ALTER VIEW HighSalaryEmployees AS

SELECT emp_id, emp_name, salary, dept_id
FROM Employees
WHERE salary > 50000;
Note: Adds the **dept_id** column to the view.

Using CREATE OR REPLACE VIEW

CREATE OR REPLACE VIEW HighSalaryEmployees AS
SELECT emp_id, emp_name, salary, dept_id
FROM Employees
WHERE salary > 60000;
Note:Updates the salary condition from **50,000** to **60,000**.

Dropping a View

The DROP VIEW statement is used to remove a view from the database.

Syntax:

DROP VIEW view_name;

Relational set operations:

UNION

INTERSECT

EXCEPT

UNION

- Combines results from two tables.
- Removes duplicates by default.
- Tables must have same number of columns and same data types.

Syntax:

SELECT column1, column2 FROM table1

UNION

SELECT column1, column2 FROM table2;

INTERSECT

- Returns **only common records** from both tables.

Syntax :

SELECT column1 FROM table1

WHERE column1 IN (SELECT column1 FROM table2);

EXCEPT

- Returns records from the **first table that do not exist** in the second table.

Syntax:

SELECT column1 FROM table1

WHERE column1 NOT IN (SELECT column1 FROM table2);