UNIT – III

Structured Query Language: Basic structure of SQL queries, Examples of basic SQL queries, set operations, Aggregate functions, Logical connectivity's(AND, OR, NOT), comparison operators, comparison using NULL values, Integrity constraints over relations, enforcing integrity constraints, disallowing null values, introduction to nested queries, outer joins, creating, altering and destroying views, triggers.

Relational Database Design: Features of good relational design, functional dependencies, inference rules, anomalies, Normalization, 1NF to 5NF, domain key normal form, denormalization.

COURSE OBJECTIVES:

- To get familiar with fundamental concepts of database management such as database design, database languages, and database-system implementation

COURSE OUTCOMES:

- Develop the knowledge of fundamental concepts of database management systems.

STRUCTURED QUERY LANGUAGE

BASIC STRUCTURE OF SQL QUERIES

SQL is a database computer language designed for managing data in relational database management systems (RDBMS), and originally based upon Relational Algebra. Its scope includes data query and update, schema creation and modification, and data access control. SQL was one of the first languages for Edgar F. Codd's relational model in his influential 1970 paper, "A Relational Model of Data for Large Shared Data Banks" and became the most widely used language for relational databases.

- IBM developed SQL in mid of 1970's.
- Oracle incorporated in the year 1979.
- SQL used by IBM/DB2 and DS Database Systems.
- SQL adopted as standard language for RDBS by ASNI in 1989.

The basic use of SQL for data professionals and SQL users is to insert, update, and

delete the data from the relational database.

- SQL allows the data professionals and users to retrieve the data from the relational database management systems.
- It also helps them to describe the structured data.

- It allows SQL users to create, drop, and manipulate the database and its tables.
- It also helps in creating the view, stored procedure, and functions in the relational database.
- It allows you to define the data and modify that stored data in the relational database.
- It also allows SQL users to set the permissions or constraints on table columns, views, and stored procedures.

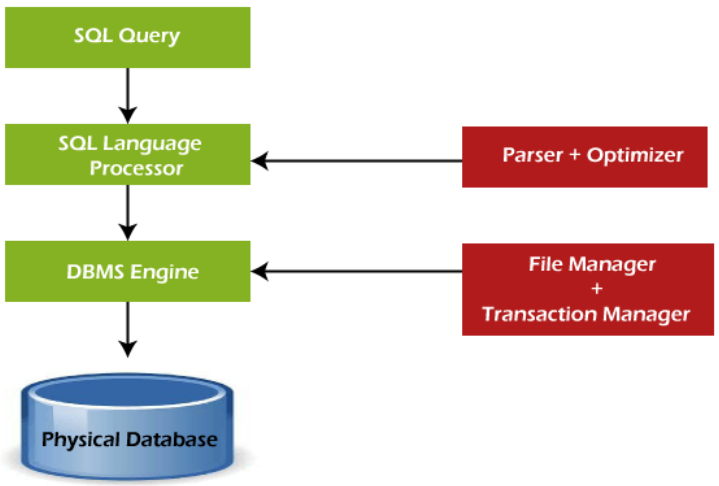


Figure: A classic query engine

EXAMPLES OF BASIC SQL QUERIES

CREATE TABLE
CREATE TABLE creates a new table inside a database. The terms int and varchar(255) in this example specify the datatypes of the columns we're creating.

```
CREATE TABLE customers (
    customer_id int,
    name varchar(255),
    age int
);
```

CREATE INDEX
CREATE INDEX generates an index for a table. Indexes are used to retrieve data from a database faster.

```
CREATE INDEX idx_name
ON customers (name);
```

CREATE VIEW
CREATE VIEW creates a virtual table based on the result set of an SQL statement. A view is like\ a regular table (and can be queried like one), but it is not saved as a permanent table in the database.

```
CREATE VIEW [Bob Customers] AS
SELECT name, age
FROM customers
WHERE name = 'Bob';
```

DROP
DROP statements can be used to delete entire databases, tables or indexes.

It goes without saying that the DROP command should only be used where absolutely necessary.

DROP TABLE
DROP TABLE deletes a table as well as the data within it.

```
DROP TABLE customers;
```

DROP INDEX
DROP INDEX deletes an index within a database.

```
DROP INDEX idx_name;
```

UPDATE
The UPDATE statement is used to update data in a table. For example, the code below would update the age of any customer named Bob in the customers table to 56.

```
UPDATE customers
SET age = 56
WHERE name = 'Bob';
```

DELETE
DELETE can remove all rows from a table (using), or can be used as part of a WHERE clause to delete rows that meet a specific condition.

```
DELETE FROM customers
WHERE name = 'Bob';
```

ALTER TABLE
ALTER TABLE allows you to add or remove columns from a table. In the code snippets below, we'll add and then remove a column for surname. The text varchar(255) specifies the datatype of the column.

```
ALTER TABLE customers
ADD surname varchar(255);
ALTER TABLE customers
DROP COLUMN surname;
```

INSERT INTO Syntax
It is possible to write the INSERT INTO statement in two ways:

1. Specify both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the INSERT INTO syntax would be as follows:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

SELECT
SELECT is probably the most commonly-used SQL statement. You'll use it pretty much every time you query data with SQL. It allows you to define what data you want your query to return.

For example, in the code below, we're selecting a column called name from a table called customers.

```
SELECT name FROM customers;
```

SELECT *
SELECT used with an asterisk (*) will return all of the columns in the table we're querying.

```
SELECT * FROM customers;
```

SELECT DISTINCT
SELECT DISTINCT only returns data that is distinct — in other words, if there are duplicate records, it will return only one copy of each.

The code below would return only rows with a unique name from the customers table.

```
SELECT DISTINCT name FROM customers;
```

SELECT INTO
SELECT INTO copies the specified data from one table into another.

```
SELECT * INTO customers FROM customers_backup;
```

SELECT TOP
SELECT TOP only returns the top x number or percent from a table.

The code below would return the top 50 results from the customers table:

```
SELECT TOP 50 * FROM customers;
```

The code below would return the top 50 percent of the customers table:

```
SELECT TOP 50 PERCENT * FROM customers;
```

AS
AS renames a column or table with an alias that we can choose. For example, in the code below, we're renaming the name column as first_name:

```
SELECT name AS first_name ROM customers;
```

FROM
FROM specifies the table we're pulling our data from:

```
SELECT name FROM customers;
```

WHERE
WHERE filters your query to only return results that match a set condition. We can use this together with conditional operators like =, >, <, >=, <=, etc.

```
SELECT name FROM customers WHERE name = 'Bob';
```

AND
AND combines two or more conditions in a single query. All of the conditions must be met for the result to be returned.

SELECT name FROM customers
WHERE name = 'Bob' AND age = 55;

OR
OR combines two or more conditions in a single query. Only one of the conditions must be met for a result to be returned.

SELECT name FROM customers
WHERE name = 'Bob' OR age = 55;

BETWEEN
BETWEEN filters your query to return only results that fit a specified range.

SELECT name FROM customers WHERE age BETWEEN 45 AND 55;

LIKE
LIKE searches for a specified pattern in a column. In the example code below, any row with a name that included the characters Bob would be returned.

SELECT name FROM customers WHERE name LIKE '%Bob%';

Other operators for LIKE:
%x — will select all values that begin with x
%x% — will select all values that include x
x% — will select all values that end with x
x%y — will select all values that begin with x and end with y
_x% — will select all values have x as the second character
x_% — will select all values that begin with x and are at least two characters long. You can add additional _ characters to extend the length requirement, i.e. x___%

IN
IN allows us to specify multiple values we want to select for when using the WHERE command.

SELECT nameFROM customers WHERE name IN ('Bob', 'Fred', 'Harry');

IS NULL
IS NULL will return only rows with a NULL value.
SELECT name FROM customers WHERE name IS NULL;

IS NOT NULL
IS NOT NULL does the opposite — it will return only rows without a NULL value.

SELECT name FROM customers WHERE name IS NOT NULL;

SET OPERATIONS
SET operators are special type of operators which are used to combine the result of two queries. Operators covered under SET operators are:

- UNION
- UNION ALL
- INTERSECT
- MINUS

There are certain rules which must be followed to perform operations using SET operators in SQL. Rules are as follows:

- The number and order of columns must be the same.
- Data types must be compatible.

UNION
Union is used to combine the results of two or more SELECT statements. However it will eliminate duplicate rows from its resultset. In case of union, number of columns and datatype must be same in both the tables, on which UNION operation is being applied.

Example of UNION
The First table,

ID	Name
1	abhi
2	adam

The Second table,

ID	Name
2	adam
3	Chester

Union SQL query will be,

```
SELECT * FROM First
UNION
SELECT * FROM Second;
```

The resultset table will look like,

ID	NAME
1	abhi
2	adam
3	Chester

UNIONALL
This operation is similar to Union. But it also shows the duplicate rows.

Example of Union All
The First table,

ID	NAME
1	abhi
2	adam

The Second table,

ID	NAME
2	adam
3	Chester

Union All query will be like,

```
SELECT * FROM First
UNION ALL
SELECT * FROM Second;
```

The resultset table will look like,

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

INTERSECT

Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of Intersect the number of columns and datatype must be same.

NOTE: MySQL does not support INTERSECT operator.
intersect set operatoin in sql

Example of Intersect
The First table,

ID	NAME
1	abhi
2	adam

The Second table,

ID	NAME
2	adam
3	Chester

Intersect query will be,

```
SELECT * FROM First
INTERSECT
SELECT * FROM Second;
```

The resultset table will look like

ID	NAME
2	adam

MINUS

Minus operation combines results of two SELECT statements and return only those in the final result, which belongs to the first set of the result.
Example of Minus
The First table,

ID	NAME
1	abhi
2	adam

The Second table,

ID	NAME
2	adam
3	Chester

Minus query will be,

```
SELECT * FROM First
MINUS
SELECT * FROM Second;
```

The result set table will look like,

ID	NAME
1	abhi

AGGREGATE FUNCTIONS

SQL Aggregate functions are functions where the values of multiple rows are grouped as input on certain criteria to form a single value result of more significant meaning.

It is used to summarize data, by combining multiple values to form a single result.

SQL Aggregate functions are mostly used with the GROUP BY clause of the SELECT statement.

Various Aggregate Functions

1. Count()
2. Sum()
3. Avg()
4. Min()
5. Max()

Example:

```
--Count the number of employees
SELECT COUNT(*) AS TotalEmployees FROM Employee;

-- Calculate the total salary
SELECT SUM(Salary) AS TotalSalary FROM Employee;

-- Find the average salary
SELECT AVG(Salary) AS AverageSalary FROM Employee;

-- Get the highest salary
SELECT MAX(Salary) AS HighestSalary FROM Employee;

-- Determine the lowest salary
SELECT MIN(Salary) AS LowestSalary FROM Employee;
```

LOGICAL CONNECTIVITY'S(AND, OR, NOT)

SQL logical operators are used to test for the truth of the condition. A logical operator like the Comparison operator returns a boolean value of TRUE, FALSE, or UNKNOWN. In this article, we will discuss different types of Logical Operators.

Logical operators are used to combine or manipulate the conditions given in a query to retrieve or manipulate data. There are some logical operators in SQL like OR, AND, NOT.
AND operator
When multiple conditions are combined using the AND operator, all rows which meet all of the given conditions will be returned.
Example: SELECT * FROM members WHERE Age < 50 AND Location = 'Los Angeles';

OR operator
When multiple conditions are combined using the OR operator, all rows which meet any of the given conditions will be returned
Example: SELECT * FROM members WHERE Location = 'Los Angeles' OR LastName = 'Hanks'

NOT operator
When multiple conditions are combined using the NOT operator, all rows which do not meet the given conditions will be returned.
Example:SELECT * FROM members WHERE NOT Location = 'Los Angeles'

COMPARISON OPERATORS

The comparison operators in SQL are categorized into the following six operators category:

1.SQL Equal Operator (=)
2.SQL Not Equal Operator (!=)
3.SQL Greater Than Equals to Operator (>=)
4.SQL Less Than Operator (<)
5.SQL Greater Than Operator (>)
6.SQL Less Than Equals to Operator (<=)

Examples:

```
SELECT * FROM Employee WHERE Emp_Salary = 35000;
SELECT * FROM Cars WHERE Car_Price != 900000;
SELECT * FROM Cars_Details WHERE Car_Number >= 6000;
SELECT * FROM Cars_Details WHERE Car_Number > 6000;
SELECT * FROM Cars_Details WHERE Car_Number <= 6000;
SELECT * FROM Cars_Details WHERE Car_Number < 6000;
```

COMPARISON WITH NULL VALUES

NULL values in a database represent the 'unknown values' or 'missing values'. These NULL values do not represent the empty string or zero value. These NULL values can be assigned to any data type in SQL.

The NULLs in SQL play an essential role in defining the 'unknown values' from the database. In SQL, almost every database contains such types of values to represent the empty spaces. The comparison operators (=, <>, <, >) are widely used in SQL queries to solve complex models.

These comparison operators cannot be used directly with the NULL values as it will result in an unknown value. There are special operators to handle the NULL values with comparison operators that are 'IS NULL' and 'IS NOT NULL' operators.

IS NULL: The IS NULL operator is used to check if the values are NULL.
IS NOT NULL: The IS NOT NULL operator is used to check if the values are not NULL.

Example:

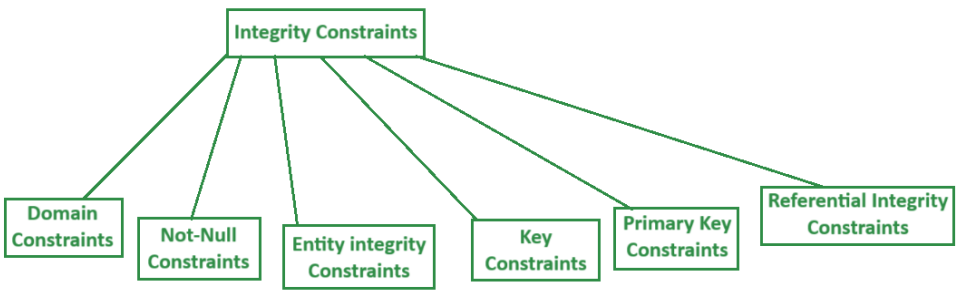
```
CREATE TABLE Employees (
   EmployeeID INT,
   Name VARCHAR(50),
   Department VARCHAR(50),
   Salary INT
);

INSERT INTO Employees (EmployeeID, Name, Department, Salary)
VALUES
   (1, 'A', 'IT', 5000),
   (2, 'B', NULL, 6000),
   (3, 'C', 'HR', NULL),
   (4, 'D', 'Marketing', NULL),
   (5, 'E', NULL, NULL);

SELECT * FROM Employees WHERE Department IS NULL;
```

INTEGRITY CONTRAINTS OVER RELATIONS

Integrity constraints are the set of predefined rules that are used to maintain the quality of information. Integrity constraints ensure that the data insertion, data updating, data deleting and other processes have to be performed in such a way that the data integrity is not affected. They act as guidelines ensuring that data in the database remain accurate and consistent. So, integrity constraints are used to protect databases.

Types of Integrity Constraints:

- Domain Constraints
- Not-Null Constraints
- Entity integrity Constraints
- Key Constraints
- Primary Key Constrains
- Referential integrity constraints



Domain Constraints
These are defined as the definition of valid set of values for an attribute. The data type of domain include string, char, time, integer, date, currency etc. The value of the attribute must be available in comparable domains.

Not-Null Constraints
It specifies that within a tuple, attributes overs which not-null constraint is specified must not contain any null value.

Entity Integrity Constraints
Entity integrity constraints state that primary key can never contain null value because primary key is used to determine individual rows in a relation uniquely, if primary key contains null value then we cannot identify those rows. A table can contain null value in it except primary key field.

Key Constraints
Keys are the entity set that are used to identify an entity within its entity set uniquely. An entity set can contain multiple keys, bit out of them one key will be primary key. A primary key is always unique, it does not contain any null value in table.
Primary Key Constraints
It states that the primary key attributes are required to be unique and not null. That is, primary key attributes of a relation must not have null values and primary key attributes of two tuples must never be same. This constraint is specified on database schema to the primary key attributes to ensure that no two tuples are same.

Referential integrity constraints
It can be specified between two tables. In case of referential integrity constraints, if a Foreign key in Table 1 refers to Primary key of Table 2 then every value of the Foreign key in Table 1 must be null or available in Table 2.

Example:

```
CREATE TABLE Persons (
    ID int NOT NULL PRIMARY KEY,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

ID	LastName	FirstName	Age
1	RAM	SHYAM	22
2	RADHE	SHYAM	23

ENFORCING INTEGRITY CONSTRAINTS

Constraints on a Single Relation are:
1. Not Null
2. Unique
3. Primary Key
4. Check (P), where P is a predicate
5. Default
6. Referential Integrity

1. Not Null Constraint:
Ex: Declare name and budget to be not null
Create table customer (id number (5), name varchar (20) not null, budget number (12, 2) not null);

2. Unique Constraint: unique (A1, A2, ..., Am)
The unique specification states that the attributes A1, A2, ... Am form a unique/candidate key.
Note: Unique keys are permitted to be null (in contrast to primary keys).

3. Primary key: Both unique and not null
Ex: dept_name vachar(20) primary key;
or
primary key (dept_name);
4. Check clause: The check clause permits domain values to be restricted with user specific constraints/conditions.

Syntax: check (P), where P is a predicate
Ex: ensure that semester is one of fall, winter, spring or summer:
create table section (course_id varchar (8), sec_id varchar (8), semester varchar (6),
year number (4), primary key (course_id, sec_id, semester, year),
check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));

5. Default: SQL allows a default value to be specified for an attribute as illustrated by the following
create table statement:

create table student (ID number (5), name varchar (20) not null, deptname varchar (20),
totcred number (3) default 0, primary key (ID));

The default value of the totcred attribute is declared to be 0. As a result, when a tuple is inserted into the
student relation, if no value is provided for the totcred attribute, its value is set to 0. The following insert
statement illustrates how an insertion can omit the value for the tot cred attribute.
insert into student(ID, name, deptname) values ('12789', 'Newman', 'Comp. Sci.');

6. Referential Integrity: Referential Integrity rule in DBMS is based on Primary and Foreign Key. The Rule defines that a foreign key have a matching primary key. Reference from a table

to another table should be valid. For example, the dept_name in the Course table has a matching valid dept_name in the Department table.
Ex: Create table Course (course_id varchar (8), course_name varchar (25), dept_name varchar (20),
credits number (3) check (credits > 0), primary key (course_id), foreign key (dept_name) references department)

DISALLOWING NULL VALUES

There may be an instance where you need to make a column non-nullable that already contains NULL values. If we try to make a column non-nullable while there are presently NULL values in said column, we'll get an error message

Before any changes are made to your table, it's important to briefly go over what data can (and cannot) be specified within an existing column that you wish to alter to NOT NULL, ensuring that no row is allowed to have a NULL value in that column.

Most critically, all existing NULL values within the column must be updated to a non-null value before the ALTER command can be successfully used and the column made NOT NULL. Any attempt to set the column to NOT NULL while actual NULL data remains in the column will result in an error and no change will occur.

Example:
UPDATE clients SET phone = '0-000-000-0000' WHERE phone IS NULL;

ALTER TABLE clients ALTER COLUMN phone VARCHAR(20) NOT NULL;

INTRODUCTION TO NESTED QUERIES

Nested queries, also known as subqueries, are a powerful feature in SQL that allows you to place one query inside another query. This can help you perform more complex data retrieval tasks that might be difficult or impossible with a single query.
Types of Nested Queries
1. Independent Nested Queries: These are executed from the innermost query to the outermost query. The inner query runs independently of the outer query, but its result is used by the outer query. For example:

```
SELECT S_NAME FROM STUDENT WHERE S_ID IN (
   SELECT S_ID FROM STUDENT_COURSE WHERE C_ID IN (
      SELECT C_ID FROM COURSE WHERE C_NAME = 'DSA' OR C_NAME = 'DBMS'
            )
         );
```

2. Correlated Nested Queries: These depend on the outer query for their values. The inner query is executed once for each row processed by the outer query. For example:

SELECT S_NAME FROM STUDENT S WHERE EXISTS (SELECT * FROM STUDENT_COURSE SC WHERE S.S_ID = SC.S_ID AND SC.C_ID = 'C1');

Benefits of Nested Queries

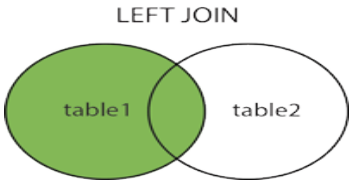
Simplify Complex Queries: Break down complex problems into simpler sub-tasks.
Enhanced Data Filtering: Perform operations that require multiple steps of filtering and aggregation.
Reusability: Use the results of one query as the input for another, making your SQL code more modular and easier to maintain.

OUTER JOINS

In a relational DBMS, we follow the principles of normalization that allows us to minimize the large tables into small tables. By using a select statement in Joins, we can retrieve the big table back. Outer joins are of following three types.

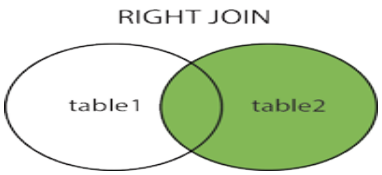
1.Left outer join
2.Right outer join
3.Full outer join

1.**Left Outer Join** : The left join operation returns all record from left table and matching records from the right table. On a matching element not found in right table, NULL is represented in that case.



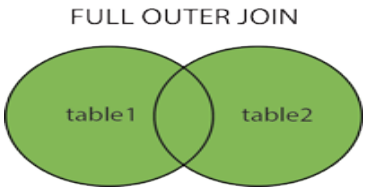
Syntax:
SELECT column_name(s) FROM table1 LEFT JOIN Table2 ON Table1.Column_Name=table2.column_name;

2. Right Outer Join : The right join operation returns all record from right table and matching records from the left table. On a matching element not found in left table, NULL is represented in that case.



Syntax:
SELECT column_name(s)FROM table1
RIGHT JOIN table2 ON table1.column_name = table2.column_name;

3. **Full Outer Join** : The full outer Join keyword returns all records when there is a match in left or right table records.



Syntax:

```
SELECT column_name FROM table1
FULL OUTER JOIN table2 ON table1.columnName = table2.columnName
WHERE condition;
```

CREATING, ALTERING AND DESTROYING VIEWS

Views in SQL are a kind of virtual table. A view also has rows and columns like tables, but a view doesn't store data on the disk like a table. View defines a customized query that retrieves data from one or more tables, and represents the data as if it was coming from a single source.

We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain conditions.

```
-- Create StudentDetails table
CREATE TABLE StudentDetails (
    S_ID INT PRIMARY KEY,
    NAME VARCHAR(255),
    ADDRESS VARCHAR(255)
);

INSERT INTO StudentDetails (S_ID, NAME, ADDRESS)
VALUES
    (1, 'Harsh', 'Kolkata'),
    (2, 'Ashish', 'Durgapur'),
    (3, 'Pratik', 'Delhi'),
    (4, 'Dhanraj', 'Bihar'),
    (5, 'Ram', 'Rajasthan');

-- Create StudentMarks table
CREATE TABLE StudentMarks (
    ID INT PRIMARY KEY,
    NAME VARCHAR(255),
    Marks INT,
    Age INT
);
INSERT INTO StudentMarks (ID, NAME, Marks, Age)
VALUES
    (1, 'Harsh', 90, 19),
    (2, 'Suresh', 50, 20),
    (3, 'Pratik', 80, 19),
    (4, 'Dhanraj', 95, 21),
    (5, 'Ram', 85, 18);
```

We can create a view using CREATE VIEW statement. A View can be created from a single table or multiple tables.

Example 1: Creating View from a single table

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;

SELECT * FROM DetailsView;
```

Example 2: Creating View from multiple tables

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;

SELECT * FROM MarksView;
```

ALTER VIEW Statement

The ALTER VIEW statement in MySQL is used to modify the definition of an existing view.
It allows us to change the query or structure of a view without dropping and recreating it.
Users must have the ALTER and CREATE VIEW privileges to use this statement.
When we use ALTER VIEW, the existing view is replaced entirely.
We cannot modify specific columns in a view. However, it cannot be used to change the view's name or its underlying table.

```
ALTER VIEW view_name column_list AS select_statement;

ALTER VIEW myView (id,first_name,city)
AS
SELECT id, upper(first_name), city  FROM employee;

//verify the view
SELECT * from myView;
```

Drop VIEW
We can also DROP the view (myView) when not in use anymore using drop command:

```
DROP view myView;
```

TRIGGERS

Trigger is a statement that a system executes automatically when there is any modification to the database. In a trigger, we first specify when the trigger is to be executed and then the action to be performed when the trigger executes. Triggers are used to specify certain integrity constraints and referential constraints that cannot be specified using the constraint mechanism of SQL.

Steps in creating Trigger with an example

1.create table employ (employee_id int, salary decimal(8,2), primary key (employee_id));

2.insert into employ values (&employee_id, &salary);

```
3.CREATE TABLE salary_changes1 (
    employee_id INT,
    old_salary DECIMAL(8 , 2 ),
    new_salary DECIMAL(8 , 2 ),
    PRIMARY KEY (employee_id)
);


4.CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON employ
FOR EACH ROW
DECLARE
   sal_diff number;
   osal  number;
   nsal  number;
    id   number;

BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   osal:= :OLD.salary;
   nsal:= :NEW.salary;
   id:= :NEW.employee_id;

 INSERT INTO salary_changes1  VALUES(id,osal,nsal);
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/

5.UPDATE employ SET     salary = salary * 1.05 WHERE    employee_id = 1
```

FEATURES OF GOOD RELATIONAL DESIGN

Database Design can be defined as a set of procedures or collection of tasks involving various steps taken to implement a database. Following are some critical points to keep in mind to achieve a good database design:

- Data consistency and integrity must be maintained.
- Low Redundancy
- Faster searching through indices
- Security measures should be taken by enforcing various integrity constraints.
- Data should be stored in fragmented bits of information in the most atomic format possible.

FUNCTIONAL DEPENDENCIES

A functional dependency occurs when one attribute uniquely determines another attribute within a relation. It is a constraint that describes how attributes in a table relate to each other. If attribute A functionally determines attribute B we write this as the A→B.

Functional dependencies are used to mathematically express relations among database entities and are very important to understanding advanced concepts in Relational Database Systems.

Example

roll no	name	dept_name	dept_building
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	jkl	ME	B2

From the above table we can conclude some valid functional dependencies:

roll_no → { name, dept_name, dept_building },→ Here, roll_no can determine values of fields name, detonate and dept_building, hence a valid Functional dependency

Here are some invalid functional dependencies:

name → dept_name Students with the same name can have different dept_name, hence this is not a valid functional dependency.

Armstrong's axioms/properties of functional dependencies:

1. Reflexivity: If Y is a subset of X, then X→Y holds by reflexivity rule
Example, {roll_no, name} → name is valid.

2. Augmentation: If X → Y is a valid dependency, then XZ → YZ is also valid by the augmentation rule.
Example, {roll_no, name} → dept_building is valid,
hence {roll_no, name, dept_name} → {dept_building, dept_name} is also valid.

3.**Transitivity**: If X → Y and Y → Z are both valid dependencies, then X→Z is also valid by the Transitivity rule.
Example, roll_no → dept_name & dept_name → dept_building, then roll_no → dept_building is also valid.

Types of Functional Dependencies in DBMS

a. Trivial functional dependency
b. Non-Trivial functional dependency
c. Multivalued functional dependency
d. Transitive functional dependency

1. In **Trivial Functional Dependency**, a dependent is always a subset of the determinant. i.e. If **X → Y** and **Y is the subset of X**, then it is called trivial functional dependency

Example:

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, **{roll_no, name}** → **name** is a trivial functional dependency, since the dependent **name** is a subset of determinant set **{roll_no, name}.** Similarly, **roll_no** → **roll_no** is also an example of trivial functional dependency.

2. Non-trivial Functional Dependency

In **Non-trivial functional dependency**, the dependent is strictly not a subset of the determinant. i.e. If **X → Y** and **Y is not a subset of X**, then it is called Non-trivial functional dependency.

Example:

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, **roll_no → name** is a non-trivial functional dependency, since the dependent **name** is **not a subset of** determinant **roll_no.** Similarly, **{roll_no, name} → age** is also a non-trivial functional dependency, since **age** is **not a subset of {roll_no, name}**

3. Multivalued Functional Dependency

In **Multivalued functional dependency**, entities of the dependent set are **not dependent on each other.** i.e. If **a → {b, c}** and there exists **no functional dependency** between **b and c**, then it is called a **multivalued functional dependency.**

For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18
45	abc	19

Here, **roll_no → {name, age}** is a multivalued functional dependency, since the dependents **name** & **age** are **not dependent** on each other(i.e. **name → age** or **age → name doesn't exist !**)

4. Transitive Functional Dependency

In transitive functional dependency, dependent is indirectly dependent on determinant. i.e. If **a → b** & **b → c**, then according to axiom of transitivity, **a → c**. This is a **transitive functional dependency.**

For example,

enrol_no	name	dept	building_no
42	abc	CO	4
43	pqr	EC	2
44	xyz	IT	1
45	abc	EC	2

Here, **enrol_no → dept** and **dept → building_no.** Hence, according to the axiom of transitivity, **enrol_no → building_no** is a valid functional dependency. This is an indirect functional dependency, hence called Transitive functional dependency.

5. Fully Functional Dependency

In full functional dependency an attribute or a set of attributes uniquely determines another attribute or set of attributes. If a relation R has attributes X, Y, Z with the dependencies X->Y

and X->Z which states that those dependencies are fully functional.

6. Partial Functional Dependency

In partial functional dependency a non key attribute depends on a part of the composite key, rather than the whole key. If a relation R has attributes X, Y, Z where X and Y are the composite key and Z is non key attribute. Then X->Z is a partial functional dependency in RBDMS.

Inference Rules

There are 6 inference rules, which are defined below:

- **Reflexive Rule:** According to this rule, if B is a subset of A then A logically determines B. Formally, **B ⊆ A** then **A → B**.
    - Example: Let us take an example of the Address (A) of a house, which contains so many parameters like House no, Street no, City etc. These all are the subsets of A. Thus, address (A) → House no. (B).
- **Augmentation Rule:** It is also known as Partial dependency. According to this rule, If A logically determines B, then adding any extra attribute doesn't change the basic functional dependency.
    - Example: **A → B**, then adding any extra attribute let say C will give **AC → BC** and doesn't make any change.
- **Transitive rule:** Transitive rule states that if A determines B and B determines C, then it can be said that A indirectly determines B.
    - Example: If **A → B** and **B → C** then **A → C**.
- **Union Rule:** Union rule states that If A determines B and C, then A determines BC.
    - Example: If **A → B** and **A → C** then **A → BC.**
- **Decomposition Rule:** It is perfectly reverse of the above Union rule. According to this rule, If A determined BC then it can be decomposed as A → B and A → C.
    - Example: If **A → BC** then **A → B** and **A → C.**
- **Pseudo Transitive Rule:** According to this rule, If A determined B and BC determines D then BC determines D.
    - Example: If **A → B** and **BC → D** then **AC → D**.

ANOMALIES

Anomaly means inconsistency in the pattern from the normal form. In Database Management System (DBMS), anomaly means the inconsistency occurred in the relational table during the operations performed on the relational table.

There can be various reasons for anomalies to occur in the database. For example, if there is a lot of redundant data present in our database then DBMS anomalies can occur. If a table is constructed in a very poor manner then there is a chance of database anomaly. Due to database anomalies, the integrity of the database suffers.

Example 1:

Worker_id	Worker_name	Worker_dept	Worker_address
65	Ramesh	ECT001	Jaipur
65	Ramesh	ECT002	Jaipur

73	Amit	ECT002	Delhi
76	Vikas	ECT501	Pune
76	Vikas	ECT502	Pune
79	Rajesh	ECT669	Mumbai

There can be three types of an anomaly in the database:

1.Updation / Update Anomaly

When we update some rows in the table, and if it leads to the inconsistency of the table then this anomaly occurs. This type of anomaly is known as an updation anomaly. In the above table, if we want to update the address of Ramesh then we will have to update all the rows where Ramesh is present. If during the update we miss any single row, then there will be two addresses of Ramesh, which will lead to inconsistent and wrong databases.

2.Insertion Anomaly

If there is a new row inserted in the table and it creates the inconsistency in the table then it is called the insertion anomaly. For example, if in the above table, we create a new row of a worker, and if it is not allocated to any department then we cannot insert it in the table so, it will create an insertion anomaly.

3.Deletion Anomaly

If we delete some rows from the table and if any other information or data which is required is also deleted from the database, this is called the deletion anomaly in the database. For example, in the above table, if we want to delete the department number ECT669 then the details of Rajesh will also be deleted since Rajesh's details are dependent on the row of ECT669. So, there will be deletion anomalies in the table.

NORMALIZATION

Database Normalization is any systematic process of organizing a database schema such that no data redundancy occurs and there is least or no anomaly while performing any update operation on data. In other words, it means dividing a large table into smaller pieces such that data redundancy should be eliminated. The normalizing procedure depends on the functional dependencies among the attributes inside a table and uses several normal forms to guide the design process.

First Normal Form(1NF)

If a relation contain composite or multi-valued attribute, it violates first normal form or a relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is **singled valued attribute**.

- **Example 1 –** Relation STUDENT in table 1 is not in 1NF because of multi-valued attribute STUD_PHONE. Its decomposition into 1NF has been shown in table 2.

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721, 9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 1

Conversion to first normal form

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721	HARYANA	INDIA
1	RAM	9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 2

Second Normal Form(2NF)

A relation is in 2NF if it is in 1NF and any non-prime attribute (attributes which are not part of any candidate key) is not partially dependent on any proper subset of any candidate key of the table. In other words, we can say that, every non-prime attribute must be fully dependent on each candidate key.
A functional dependency X->Y (where X and Y are set of attributes) is said to be in **partial dependency**, if Y can be determined by any proper subset of X.
However, in 2NF it is possible for a prime attribute to be partially dependent on any candidate key, but every non-prime attribute must be fully dependent(or not partially dependent) on each candidate key of the table.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.
To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30

47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Math
83	Computer

Third Normal Form (3NF)

A relation is in the third normal form, if there is no transitive dependency for non-prime attributes as well as it is in the second normal form. A relation is in 3NF if at least one of the following conditions holds in every non-trivial function dependency X –> Y.

- X is a super key.
- Y is a prime attribute (each element of Y is part of some candidate key).

STUD_NO	STUD_NAME	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	HARYANA	INDIA	20
2	RAM	PUNJAB	INDIA	19
3	SURESH	PUNJAB	INDIA	21

Table 4

Functional Dependency set: {STUD_NO -> STUD_NAME,
STUD_NO -> STUD_STATE,
STUD_STATE -> STUD_COUNTRY,
STUD_NO -> STUD_AGE}

Candidate Key: {STUD_NO}

For this relation in table 4, STUD_NO -> STUD_STATE and STUD_STATE -> STUD_COUNTRY are true.

So STUD_COUNTRY is transitively dependent on STUD_NO. It violates the third normal form. To convert it in third normal form, we will decompose the relation STUDENT (STUD_NO, STUD_NAME, STUD_STATE, STUD_COUNTRY_STUD_AGE) as:

STUDENT (STUD_NO, STUD_NAME, STUD_STATE, STUD_AGE)

STUD_NO	STUD_NAME	STUD_STATE	STUD_AGE
1	RAM	HARYANA	20
2	RAM	PUNJAB	19
3	SURESH	PUNJAB	21

STATE_COUNTRY (STATE, COUNTRY)

STUD STATE	STUD COUNTRY
HARYANA	INDIA
PUNJAB	INDIA

Boyce–Codd Normal Form (BCNF)

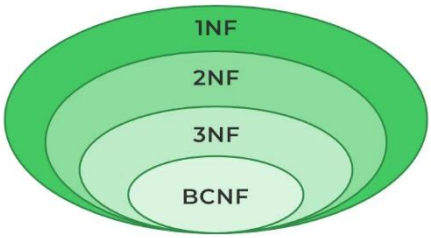
Boyce–Codd Normal Form (BCNF) is based on functional dependencies that take into account all candidate keys in a relation; however, BCNF also has additional constraints compared with the general definition of 3NF.

Rules for BCNF

Rule 1: The table should be in the 3rd Normal Form.

Rule 2: X should be a superkey for every functional dependency (FD) X−>Y in a given relation.

Note: To test whether a relation is in BCNF, we identify all the determinants and make sure that they are candidate keys.



To determine the highest normal form of a given relation R with functional dependencies, the first step is to check whether the BCNF condition holds. If R is found to be in BCNF, it can be safely deduced that the relation is also in 3NF, 2NF, and 1NF as the hierarchy shows. The 1NF has the least restrictive constraint – it only requires a relation R to have atomic values in each tuple. The 2NF has a slightly more restrictive constraint.

The 3NF has a more restrictive constraint than the first two normal forms but is less restrictive than the BCNF. In this manner, the restriction increases as we traverse down the hierarchy.

Example 1

Stu_ID	Stu_Branch	Stu_Course	Branch_Number	Stu_Course_No
101	Computer Science & Engineering	DBMS	B_001	201
101	Computer Science & Engineering	Computer Networks	B_001	202
102	Electronics & Communication Engineering	VLSI Technology	B_003	401
102	Electronics & Communication Engineering	Mobile Communication	B_003	402

Functional Dependency of the above is as mentioned:
Stu_ID −> Stu_Branch
Stu_Course −> {Branch_Number, Stu_Course_No}

Stu_Branch Table

Stu_ID	Stu_Branch
101	Computer Science & Engineering
102	Electronics & Communication Engineering

Candidate Key for this table: **Stu_ID**.

Stu_Course Table

Stu_Course	Branch_Number	Stu_Course_No
DBMS	B_001	201
Computer Networks	B_001	202
VLSI Technology	B_003	401
Mobile Communication	B_003	402

Candidate Key for this table: **Stu_Course**.

Stu_ID to Stu_Course_No Table

Stu_ID	Stu_Course_No
101	201
101	202
102	401
102	402

Candidate Key for this table: **{Stu_ID, Stu_Course_No}.**

Fourth Normal Form(4NF)

The Fourth Normal Form (4NF) is a level of database normalization where there are no non-trivial multivalued dependencies other than a candidate key. It builds on the first three normal forms (1NF, 2NF, and 3NF) and the Boyce-Codd Normal Form (BCNF). It states that, in addition to a database meeting the requirements of BCNF, it must not contain more than one multivalued dependency.

Properties

A relation R is in 4NF if and only if the following conditions are satisfied:

1. It should be in the Boyce-Codd Normal Form (BCNF).
2. The table should not have any Multi-valued Dependency.

A table with a multivalued dependency violates the normalization standard of the Fourth Normal Form (4NF) because it creates unnecessary redundancies and can contribute to inconsistent data. To bring this up to 4NF, it is necessary to break this information into two tables.

Example: Consider the database table of a class that has two relations R1 contains student ID(SID) and student name (SNAME) and R2 contains course id(CID) and course name (CNAME).

Table R1

SID	SNAME
S1	A
S2	B

Table R2

CID	CNAME
C1	C
C2	D

When their cross-product is done it resulted in multivalued dependencies.

Table R1 X R2

SID	SNAME	CID	CNAME
S1	A	C1	C
S1	A	C2	D
S2	B	C1	C
S2	B	C2	D

Multivalued dependencies (MVD) are:

SID->->CID; SID->->CNAME; SNAME->->CNAME

Fifth Normal Form(5NF)

A relation R is in Fifth Normal Form if and only if everyone joins dependency in R is implied by the candidate keys of R. A relation decomposed into two relations must have lossless join Property, which ensures that no spurious or extra tuples are generated when relations are reunited through a natural join.

Properties

A relation R is in 5NF if and only if it satisfies the following conditions:

1. R should be already in 4NF.
2. It cannot be further non loss decomposed (join dependency).

Example – Consider the above schema, with a case as "if a company makes a product and an agent is an agent for that company, then he always sells that product for the company". Under these circumstances, the ACP table is shown as:

Table ACP

Agent	Company	Product
A1	PQR	Nut
A1	PQR	Bolt
A1	XYZ	Nut
A1	XYZ	Bolt
A2	PQR	Nut

The relation ACP is again decomposed into 3 relations. Now, the natural Join of all three relations will be shown as:

Table R1

Agent	Company
A1	PQR
A1	XYZ
A2	PQR

Table R2

Agent	Product
A1	Nut

Agent	Product
A1	Bolt
A2	Nut

Table R3

Company	Product
PQR	Nut
PQR	Bolt
XYZ	Nut
XYZ	Bolt

The result of the Natural Join of R1 and R3 over 'Company' and then the Natural Join of R13 and R2 over 'Agent'and 'Product' will be **Table ACP**.

Hence, in this example, all the redundancies are eliminated, and the decomposition of ACP is a lossless join decomposition. Therefore, the relation is in 5NF as it does not violate the property of lossless join.

DOMAIN KEY NORMAL FORM

There is no Hard and fast rule to define normal form up to 5NF. Historically the process of normalization and the process of discovering undesirable dependencies were carried through 5NF, but it has been possible to define the stricter normal form that takes into account additional type of dependencies and constraints. The basic idea behind the *DKNF* is to specify the normal form that takes into account all the possible dependencies and constraints. In simple words, we can say that DKNF is a normal form used in database normalization which requires that the database contains no constraints other than domain constraints and key constraints. In other words, a relation schema is said to be in DKNF only if all the constraints and dependencies that should hold on the valid relation state can be enforced simply by enforcing the domain constraints and the key constraints on the relation. For a relation in DKNF, it becomes very straight forward to enforce all the database constraints by simply checking that each attribute value is a tuple is of the appropriate domain and that every key constraint is enforced. Reason to use DKNF are as follows:

1. To avoid general constraints in the database that are not clear key constraints.
2. Most database can easily test or check key constraints on attributes.

However, because of the difficulty of including complex constraints in a DKNF relation its practical utility is limited means that they are not in practical use, since it may be quite difficult to specify general integrity constraints.

Advantages of Domain Key Normal Form:

Improved Data Integrity: DK/NF ensures that all dependencies and constraints are preserved, resulting in improved data integrity.

Reduced Data Redundancy: DK/NF reduces data redundancy by breaking down a relation into smaller, more focused relations.

Improved Query Performance: By breaking down a relation into smaller, more focused relations, query performance can be improved.

Easier Maintenance and Updates: The smaller, more focused relations are easier to maintain and update than the original relation, making it easier to modify the database schema and update the data.

Better Flexibility: DK/NF can improve the flexibility of the database system by allowing for easier modification of the schema.

Disadvantages of Domain Key Normal Form:

Increased Complexity: Normalizing a relation to DK/NF can increase the complexity of the database system, making it harder to understand and manage.

Costly: Normalizing a relation to DK/NF can be costly, especially if the database is large and complex. This can require additional resources, such as hardware and personnel.

Reduced Performance: Although query performance can be improved in some cases, in others, normalization to DK/NF can result in reduced query performance due to the need for additional join operations.

Limited Scalability: Normalization to DK/NF may not scale well in larger databases, as the number of smaller, focused relations can become unwieldy.

DENORMALIZATION

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database. Note that *denormalization* does not mean 'reversing normalization' or 'not to normalize'. It is an optimization technique that is applied after normalization.

Basically, The process of taking a normalized schema and making it non-normalized is called denormalization, and designers use it to tune the performance of systems to support time-critical operations.

In a traditional normalized database, we store data in separate logical tables and attempt to minimize redundant data. We may strive to have only one copy of each piece of data in a database.

For example, in a normalized database, we might have a Courses table and a Teachers table. Each entry in Courses would store the teacherID for a Course but not the teacherName. When we need to retrieve a list of all Courses with the Teacher's name, we would do a join between these two tables.

In some ways, this is great; if a teacher changes his or her name, we only have to update the name in one place.

The drawback is that if tables are large, we may spend an unnecessarily long time doing joins on tables.

Denormalization, then, strikes a different compromise. Under denormalization, we decide that we're okay with some redundancy and some extra effort to update the database in order to get the efficiency advantages of fewer joins.

Pros of Denormalization:

1. Retrieving data is faster since we do fewer joins
2. Queries to retrieve can be simpler(and therefore less likely to have bugs), since we need to look at fewer tables.

Cons of Denormalization:

1. Updates and inserts are more expensive.
2. Denormalization can make *update* and *insert* code harder to write.
3. Data may be inconsistent.
4. Data redundancy necessitates more storage.

--- END OF UNIT-III ---