DATABASE MANAGEMENT SYSTEM

Data:

- Data is any collection of raw facts. It can be anything from numbers and letters to images, videos, and audio recordings. Data is not very useful on its own, but when it is organized and analyzed, it can be used to gain valuable insights.

DataBase:

- A database is an organized collection of data.
- It is typically stored electronically in a computer system and can be accessed and managed by a database management system (DBMS).
- Databases are used to store a wide variety of data, including customer information, product information, financial data, and website content.

Data Types:

- INT: An integer value, such as a person's ID number.
- VARCHAR(255): A variable-length string value with a maximum length of 255 characters, such as a person's name.
- INT: An integer value, such as a person's age.
- DECIMAL(10,2): A decimal value with 10 digits to the left of the decimal point and 2 digits to the right, such as a person's salary.
- DATE: A date value, such as a person's hire date.
- BOOLEAN: A Boolean value, either TRUE or FALSE, such as whether a person is still active.
- BLOB: A binary data type, such as a person's photo.

Create Table:

```
CREATE TABLE Persons (

   PersonID int,

   firstName varchar(255),

   LastName varchar(255),

   Address varchar(255),

   City varchar(255)

);
```

Check:

```
SELECT * FROM Persons;
```

Insert Command:

INSERT INTO Persons VALUES (1,’Sai’,’Kishore’,’particular’,’Chennai’);

Data Definition Language:

DDL stands for Data Definition Language. DDL commands are used to create, modify, and delete database objects, such as tables, views, indexes, and procedures.

- CREATE TABLE: Creates a new table.

Ex: CREATE TABLE TABLENAME

- ALTER TABLE: Modifies an existing table.

Ex: ALTER TABLE TABLENAME ADD COLUMNNAME COLUMNTYPE

- DROP TABLE: Deletes a table.

Ex: ALTER TABLE TABLENAME DROP COLUMN COLUMNNAME

- RENAME: To Change Table Name and Column Name

Ex: SP_RENAME ‘Old TableName’,’New TableName’;

Ex:SP_RENAME ‘TableName.OldColumnName’,’NewColumnName’,’COLUMN’

- CREATE VIEW: Creates a new view.
- DROP VIEW: Deletes a view.
- CREATE INDEX: Creates a new index on a table.
- DROP INDEX: Deletes an index from a table.
- CREATE PROCEDURE: Creates a new stored procedure.
- DROP PROCEDURE: Deletes a stored procedure.

DML Commands

DML stands for Data Manipulation Language. DML commands are used to insert, update, and delete data in database tables.

Here are some examples of DML commands:

- INSERT INTO: Inserts new data into a table.

Ex: INSERT INTO TABLENAME VALUES (1,’Value’,’Value’);

- UPDATE: Updates existing data in a table.

Ex: UPDATE TABLENAME SET Name='Sai' WHERE ID=2;

- DELETE FROM: Deletes data from a table.

Ex: DELETE FROM customers WHERE customer_id = 1;

Retrieve Data:

SELECT * FROM TABLENAME WHERE Id=1;

Logical Operators:

- AND: The AND operator returns TRUE only if both of the conditions are TRUE.

Ex: SELECT * FROM customers WHERE first_name = 'John' AND last_name = 'Doe';

- OR: The OR operator returns TRUE if either of the conditions is TRUE.

Ex: SELECT * FROM customers WHERE first_name = 'John' OR last_name = 'Doe';

- NOT: The NOT operator reverses the truth value of a condition.

Ex: SELECT * FROM customers WHERE NOT first_name = 'John';

- BETWEEN: The BETWEEN operator returns TRUE if a value is within a specified range.

Ex: SELECT * FROM customers WHERE age BETWEEN 18 AND 30;

Like Operators:

The LIKE operator in SQL is used to match strings against a specified pattern. The pattern can include wildcards, which are special characters that represent one or more characters.

The two most common wildcards used with the LIKE operator are:

- % - Represents zero, one, or multiple characters.
- _ - Represents a single character.

-- Select all customers whose first name starts with the letter 'J'.

- SELECT * FROM customers WHERE first_name LIKE 'J%';

-- Select all customers whose last name ends with the letter 'E'.

- SELECT * FROM customers WHERE last_name LIKE '%E';

-- Select all customers whose email address contains the string '@example.com'.

- SELECT * FROM customers WHERE email LIKE '%@example.com%';-- Select all customers whose phone number contains the sequence of digits '123'
- SELECT * FROM customers WHERE phone LIKE '%123%';

The HAVING clause in SQL

The HAVING clause in SQL is used to filter groups of rows based on aggregate functions like SUM(), AVG(), COUNT(), MIN(), and MAX(). Unlike the WHERE clause, which filters individual rows based on their values, the HAVING clause filters groups based on the aggregated values of those groups.

Purpose:

- Filter groups of rows based on aggregated values.
- Used after the GROUP BY clause and before the ORDER BY clause.

Functionality:

- Similar to the WHERE clause, but operates on aggregate values instead of individual rows.
- Provides an additional layer of filtering based on the output of aggregate functions.

```
SELECT customer_id, SUM(order_amount) AS total_amount

FROM orders

GROUP BY customer_id

HAVING total_amount > 1000;
```

Primary Key:

- Primary keys can be simple or composite. A simple primary key is a single column that uniquely identifies each row in the table. A composite primary key is a set of two or more columns that uniquely identify each row in the table.

```
CREATE TABLE Persons (

    ID int PRIMARY KEY,

    LastName varchar(255) NOT NULL,

    FirstName varchar(255),

    Age int);
```

Foreign Keys in SQL:

A foreign key is a column or set of columns in a relational database table that references the primary key of another table. It establishes a relationship between the two tables, ensuring data consistency and integrity.

Purpose:

- Link related data across multiple tables.
- Enforce data integrity by preventing invalid references.
- Improve data organization and simplify retrieval.

Structure:

- Declared within a table definition.
- References a primary key in another table.
- Can be defined on one or more columns.

Benefits:

- Data consistency: Ensures related data exists across tables.
- Referential integrity: Prevents orphaned records.
- Data organization: Makes tables easier to understand and manage.
- Simplified retrieval: Allows efficient querying of related data.

```
CREATE TABLE Books (
    book_id INT PRIMARY KEY,

    title VARCHAR(255) NOT NULL,

    author_id INT NOT NULL,

    FOREIGN KEY (author_id) REFERENCES Authors(author_id)

);



CREATE TABLE Authors (

    author_id INT PRIMARY KEY,

    name VARCHAR(255) NOT NULL

);



CREATE TABLE BorrowedBooks (

    user_id INT NOT NULL,

    book_id INT NOT NULL,

    PRIMARY KEY (user_id, book_id),

    FOREIGN KEY (user_id) REFERENCES Users(user_id),

    FOREIGN KEY (book_id) REFERENCES Books(book_id)

);
```

Column Aliasing:

- In SQL, column aliasing is a technique used to give a column in a table a temporary name for the duration of a query.

Ex: SELECT first_name AS "name" FROM Customers;

Ex:SELECT ID AS "Roll No, Name AS "Student Name", score as "Total marks", score/10 as "CGPA" FROM Student_marks;

Retrive Specific Row :

- SELECT * FROM MOCK_DATA WHERE ID=500;

Logical Operators:

- The Logical Operators are Classified into Three Types

1) AND

- SELECT * FROM MOCK_DATA WHERE ID>500 AND Gender = 'Male';

2) OR

- SELECT * FROM MOCK_DATA WHERE ID>2000 or ID<700;

3) NOT

- SELECT * FROM MOCK_DATA WHERE NOT NAME='SAI';

4) IN

- SELECT * FROM MOCK_DATA WHERE ID IN (101,202,307,999);
- SELECT * FROM MOCK_DATA WHERE ID NOT IN (1,3,5,7);

5) BETWEEN

- SELECT * FROM MOCK_DATA WHERE ID BETWEEN 500 AND 700;
- SELECT * FROM MOCK_DATA WHERE ID NOT BETWEEN 10 AND 20;

Like Operator :

- The **Like** operator is a powerfool tool for pattern matching in SQL. It allows you to search for strings that match a specific pattern, including partial matches and wild cards. The operator is commonly used in the **WHERE** clause to filter data based on specific patterns.

- The LIKE operator supports two wildcards:
  1) %: Represents zero,one,or multiple characters.
  2) _ :Represents a single character.

Syntax:

- SELECT * FROM customers WHERE city LIKE '%London%';
- SELECT * FROM customers WHERE first_name LIKE 'A%';

CASE Statement:

- The CASE statement in SQL is a control flow tool that allows you to add if-else logic to a query.
- It is a conditional statement that helps us to make decisions based on a set of conditions.
- The CASE expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement).
- So once a condition is true, it will stop reading and return the result.
- If no conditions are true, it returns the value in the ELSE clause. If there no ELSE part no conditions are true, it returns NULL

Syntax:

```
select *,
CASE
when score>60 then 'passed'
when score>70 then 'super'
when score >80 then 'keka'
end as "Grade"
from student_marks
order by score DESC;
```

Example-2:

```
select *,
CASE
when score>=90 then 'A'
when score>=80 then 'B'
when score>=70 then 'C'
when score>=60 then 'D'
when score>=50 then 'C'
else 'Take another course in LevelUp'
end as 'Grade'
from student_marks order by score DESC;
```

Example-3:

```
ALTER TABLE student_marks ADD results VARCHAR(20);
UPDATE student_marks  --Update Command
SET results = CASE
   WHEN score > 60 THEN 'Passed'
   WHEN score = 60 THEN 'Just passed'
   ELSE 'Failed'
END;
DELETE FROM student_marks WHERE id=1; --Delete Command
```

BackUp Table:

```
CREATE TABLE BACKUP_TABLE (
ID INT PRIMARY KEY,
NAME VARCHAR(20),
SCORE INT,
RESULTS VARCHAR(20)
);
SELECT * FROM BACKUP_TABLE;     #check
INSERT INTO BACKUP_TABLE SELECT * FROM STUDENT_MARKS;  #insert command
```

Aggregations Functions:

```
CREATE TABLE PRODUCTQTY (
Region VARCHAR(100),
Product VARCHAR(50),
YEAR INT,
QUANTITY INT
);
SELECT * FROM PRODUCTQTY; #CHECK
INSERT INTO PRODUCTQTY VALUES (‘East’,’Computer’,2020,13000),
(‘South’,’Computer’,2020,45000),
(‘North’,’Computer’,2020,25000),
(‘East’,’HardDisk’,2020,1900),
(‘West’, ’Computer’,2021,25000),
(‘South’,’HardDisk’,2021,5600),
(‘West’,’HardDisk’,2021,6500),
(‘East’,’Pendrive’,2021,1200),
(‘North’,’Mouse’,2019,1600),
(‘South’,’Pendrive’,2019,2700),
(‘East’,’Mouse’,2019,2000),
(‘West’,’Pendrive’,2019,1900);
```

- SUM(): Returns the sum of all or distinct values in a set.
  - ➢ SELECT SUM(Quantity) AS “Total Sales” FROM PRODUCTQTY;
- MAX(): Returns the maximum value in a set.
  - ➢ SELECT MAX(Quantity) AS “Total Sales” FROM PRODUCTQTY;
- MIN(): Returns the minimum value in a set.
  - ➢ SELECT MIN(Quantity) AS “Total Sales” FROM PRODUCTQTY;
- COUNT(): Returns the number of items in a set.
  - ➢ SELECT COUNT(Quantity) AS “Total Sales” FROM PRODUCTQTY;-→Not in Rows.
- AVG(): Returns the average of a set.
  - ➢ SELECT AVG(QUANTITY) AS “Total Sales” FROM PRODUCTQTY;
- DISTINCT():Returns the Unique of a set.

- SELECT DISTINCT Region FROM PRODUCTQTY;

GROUP BY:

- The GROUP BY clause in SQL is used to arrange identical data into groups with the help of some functions.
- If a particular column has the same values in different rows, then it will arrange these rows in a group.

Syntax:

- SELECT Region, sum(Quantity) AS “Region Sales” FROM PRODUCTQTY group by Region order by “Region Sales” DESC;
- SELECT Region, sum(Quantity) AS “Region Sales” from PRODUCTQTY WHERE>30000 Group by Region order by “Region sales” DESC;
- SELECT Region, YEAR,SUM(Quantity) AS “Region sales” FROM PRODUCTQTY Group by Region, year order by “Region Sales” DESC;
- SELECT Region, YEAR, SUM(Quantity) AS “Region Sales” FROM PRODUCTQTY Group by Region, YEAR Order by YEAR DESC;

Procedures:

- ➔ A Stored procedure in SQL is a self-contained set of SQL statements that can be saved and reused. Stored procedures can be used to perform a variety of tasks, such as inserting, updating, and deleting data, as well as performing complex calculations.
- ➔ Stored procedures are compiled when they ae created, which means that they can be executed much faster than equivalent as-hoc SQL queries. They can also improve the performance of applications by reducing the number of round trips between the application and the database.
- ➔ Stored procedure are also useful for security, as they can be used to encapsulate sensitive data and logic. This makes it easier to control who has access to the data how it can be modified.

Create Procedure:

CREATE Procedure USP_MOCKDATA_Select1

AS

```
BEGIN

SELECT * FROM MOCK_DATA

END
```

Check:

- Exec USP_MOCKDATA_Select1;

Insert Procedure:

```
CREATE or Alter Procedure USP_MOCKDATA_INSERT

(@Id INT,

@FirstName VARCHAR(50),

@SurName VARCHAR(50),

@Email VARCHAR(50),

@Gender VARCHAR(50),

@PhoneNumber bigint)

AS

BEGIN

INSERT INTO MOCK_DATA (Id,first_name,last_name,email,gender,PhoneNumber) VALUES
(@Id, @FirstName,@SurName, @Email, @Gender,@PhoneNumber)

END

Exec USP_MOCKDATA_INSERT (9998,’Rashid’,’Khan’,’Rkhan@gmail.com’,’Male’,9908610847)
```

Check:

```
Exec USP_MOCKDATA_Select1 9998;
```

Update Command:

```
CREATE or ALTER Procedure USP_MOCKDATA_UPDATE

(@Id INT,

@firstName VARCHAR(50),

@SurName VARCHAR(50),
@Email VARCHAR(50),

@Gender VARCHAR(50),

@PhoneNumber bigint)

AS

BEGIN

UPDATE MOCK_DATA

SET

first_name=@firstName,last_name=@SurName,email=@Email,gender=@Gender,PhoneNumber
=@PhoneNumber WHERE Id=@id

END

Exec USP_MOCKDATA_UPDATE 9999,'Imran','Khan','Imran@gmail.com','male',9856324710
```

Check:

- Exec USP_MOCKDATA_Select 9999;

Retrieve Particular Data:

```
CREATE or Alter Procedure USP_MOCKDATA_Select1

(@Id INT)

AS

BEGIN

SELECT * FROM MOCK_DATA WHERE Id=@Id

END

Exec USP_MOCKDATA_Select 855;
```

ADD Two Numbers:

```
CREATE Procedure SUM

(

@X INT,

@Y INT,
@SUM INT Output )

AS

BEGIN

Set @SUM=@X+@Y

END

DECLARE @ADD INT

EXEC SUM 10, 20, @ADD OUTPUT

Print @ADD
```

Procedural Operations:

```
CREATE or ALTER Procedure USP_MOCKDATA_CRUD

(@Id INT,

@FirstName VARCHAR(20),

@LastName VARCHAR(20),

@Email VARCHAR(20),

@Gender VARCHAR(20),

@Operation VARCHAR(20)=' ')

AS

BEGIN

If @Operation ='INSERT'

BEGIN

INSERT INTO MOCK_DATA (ID,First_Name,last_Name,Email,gender) VALUES
(@id,@FirstName,@LastName,@Email,@Gender)

END;
```

Update Operation:

```
If @Operation = 'Update'

BEGIN
```

UPDATE MOCK_DATA SET first_name=@first_name, last_name=@Last_name,Email=@Email,gender=@Gender

WHERE id=@Id

END

Delete operation:

If @Operation='Delete'

BEGIN

DELETE FROM MOCK_DATA WHERE id=@id

END

SELECT OPERATION:

If @Operation = 'Select'

BEGIN

SELECT * FROM MOCK_DATA WHERE id=@Id

END;

SCALER FUNCTION:

- A scalar function in SQL is a function that takes one or more input parameters and returns a single scalar value. Scalar functions can be used to perform a variety of operations on data, such as arithmetic calculations, string manipulation, and date and time operations.
- Scalar functions can be either built-in functions or user-defined functions. Built-in scalar functions are provided by the database engine and can be used directly in SQL queries. User-defined scalar functions are created by the database user and can be used to encapsulate custom business logic.

- LEN(): Returns the length of a string.
    - SELECT LEN('Hello World');
    - SELECT LEN(ColumnName) FROM TableName;
- UPPER(): Converts a string to uppercase.

- SELECT UPPER('SQL Tutorial is FUN!');

- SELECT UPPER(ColumnName) AS UppercaseColumnName FROM TableName;

- LOWER(): Converts a string to lowercase.
  - SELECT LOWER('SQL Tutorial is FUN!');
  - SELECT LOWER(ColumnName) AS LowercaseColumnName FROM TableName;
- AVG(): Calculates the average value of a column.
  - SELECT AVG(Price) AS “average price” FROM Products;
- SUM(): Calculates the sum of the values in a column.
  - SELECT SUM(Price) AS [total price] FROM Products;
- COUNT(): Counts the number of rows in a table or the number of non-null values in a column.
  - SELECT COUNT(ProductID) FROM Products WHERE Price > 20;

<u>SUM for Scalar:</u>

```
CREATE or ALTER FUNCTION Addition (
@Num1 Bigint,
@Num2 Bigint
)
Returns Bigint
END
Print dbo.(20000,20000);
```

<u>AVG for Scalar:</u>

```
CREATE or ALTER FUNCTION UDF_AVG
(@N1 Dec(10,2),
@N2 Dec (10,2),
@N3 Dec(10,2)
@N4 Dec(10,3)
)
Returns Dec(10,2)
AS
BEGIN
```

```
DECLARE @SUM Bigint
DECLARE @AVG Dec(10,2)
SET @SUM =@N1+@N2+@N3+@N4
SET @AVG=@SUM/4
Return @AVG
END
Print.dbo.UDF_AVG(100,200,300,400);
Print dbo.UDF_AVG(10,20,30,40);
```

<u>A^2+B^2+2AB:</u>

```
SELECT  POWER (2,3) AS "POWER"
CREATE OR ALTER FUNCTION UDF_AB
(@A INT,
@B INT)
RETURNS bigint
AS
BEGIN
RETURN POWER(@A,2)+POWER(@B,2)+2*@a*@b
END
Print dbo.UDF_AB(2,3)
```

<u>DISCOUNT:</u>

```
CREATE OR ALTER FUNCTION UDF_Net_Sales
(
@quantity INT,
@Price DEC (10,2),
@discount DEC(3,2)
)
RETURNS DEC(20,2)
AS
BEGIN
RETURN @quantity*@price*(1-@discount)
```

```
END;
Print dbo.UDF_net_Sales;
```

Avg&Sum Function:

```
CREATE OR ALTER FUNCTION AvgandSum (
@Num1 bigint,
@Num2 bigint)
Returns DEC (12,2)
AS
BEGIN
Declare @Sum Dec (12,2)
Declare @Avg Dec (12,2)
SET @Sum=@Num1+@Num2
SET@Avg=@Sum/2
Return @Avg
End
Print dbo.AvgandSum(120,240);
```

Table Valued Function:

- In SQL, a table-valued function (TVF) is a user-defined function that returns a table. Unlike scalar functions that return a single value, a table-valued function can return a result set that resembles a table with multiple rows and columns.

There are Classified into two types:

Inline Table-Valued Functions (Inline TVF):

- This type of function is similar to a view in that it returns a result set defined by a SELECT statement. The function body is essentially a SELECT statement that specifies the columns and rows to be returned. An inline TVF is defined using the **RETURNS TABLE** clause.

Multi-Statement Table-Valued Functions (Multi-Statement TVF): This type of function uses a series of **INSERT** statements to build and populate a table variable, which is then returned as the result of the function. The function body contains explicit **INSERT** statements to populate the table variable.

```
CREATE TABLE STUDENT_FUNCTION (

RollNo INT PRIMARY KEY,

Name VARCHAR(50),

TotalMarks INT);

SELECT * FROM STUDENT_FUNCTION;


CREATE TABLE SUBJECTS (

RollNo INT FOREIGN KEY REFERENCES STUDENT_FUNCTION,

Maths INT,

Physics INT,

Chemistry INT);

SELECT * FROM SUBJECTS;

INSERT INTO STUDENT_FUNCTION VALUES(1,'Vishnu',1000);

INSERT INTO STUDENT_FUNCTION VALUES(2,'Revi',30);

INSERT INTO STUDENT_FUNCTION VALUES(3,'MamaKutty',40);

INSERT INTO STUDENT_FUNCTION VALUES(4,'Uthaman',60);

INSERT INTO STUDENT_FUNCTION VALUES(5,'Bhaskar',80);

INSERT INTO SUBJECTS VALUES (1,40,30,30);

INSERT INTO SUBJECTS VALUES (2,10,10,10);

INSERT INTO SUBJECTS VALUES (3,20,10,10);

INSERT INTO SUBJECTS VALUES (4,20,20,20);
INSERT INTO SUBJECTS VALUES (5,40,20,20);
```

Student Total Marks:

```
CREATE OR ALTER FUNCTION getStudentTotalMarks

(@Rno INT)

RETURNS INT

AS

BEGIN

RETURN (SELECT TotalMarks FROM Student_Function WHERE Rollno=@Rno)

END

PRINT DBO.GetStudentTotalMarks(1)

Select*from student_function;

Select * from subjects;


Select * from student_function;

Select * from subjects;

CREATE OR ALTER FUNCTION GetPhysicsMarks

(@name VARCHAR(50))

RETURNS INT

AS

BEGIN

Declare @Rollno INT

Select @Rollno = Rollno from student_function where name=@name

RETURN (SELECT Physics FROM Subjects WHERE Rollno=@Rollno)
```

```
END

Print dbo.GetPhysicsMarks('MamaKutty')
```

Get All Subjects Marks:

```
CREATE OR ALTER FUNCTION GetAll

(@RollNo INT)

RETURNS Table

AS

RETURN SELECT * FROM Subjects WHERE RollNo=@RollNo

SELECT * FROM GetAll(4)
```

GetAverage:

```
CREATE OR ALTER FUNCTION GetAverage

(@Rno int)

Returns int

AS

Begin

Declare @Avg Dec(12,2)

Select @Avg = (Maths+Physics+Chemistry)/3 from subjects where RollNo=@Rno

Return @Avg

End

Print dbo.GetAverage(4)
```

Get Report Card:

```
CREATE OR ALTER FUNCTION UDF_Get_ReportCard

(@Name varchar(50))

Returns @Marks Table

(RollNo1 INT,

Name1 Varchar(20),

TotalMarks1 INT,

Maths1 INT,

Physics1 INT,

Chemistry1 INT,

Average Dec(5,2)

)

AS

BEGIN

Insert into @Marks (Name1) values (@Name)

Declare @Rno int

Declare @Avg Dec(5,2)

Select @Rno=Rollno from student_function where Name=@Name

Select @Avg=TotalMarks/3 from student_Function where RollNo=@Rno

Update @Marks set RollNo1=@Rno,

Average=@Avg,

TotalMarks1=(Select TotalMarks from student_function where RollNO=@Rno),

Physics1=(select physics from subjects where RollNo=@Rno),
Maths1=(select Maths from subjects where RollNo=@Rno),

Chemistry1=(select Chemistry from Subjects where RollNo=@Rno)

Where Name=@Name

Return

End

Select * from UDF_Get_ReportCard('Revi')
```

Joins:

- In SQL, a JOIN clause is used to combine rows from two or more tables based on a related column between them. It is a fundamental concept in relational databases and is used to retrieve data from multiple tables that are related to each other.

These are Classified into so many types:

INNER JOIN (Default JOIN): Returns records that have matching values in both tables. This is the most common type of JOIN.

Self Join:

- A self-join, also known as a reflexive join, is a type of SQL JOIN that combines a table to itself. It is used to compare rows within the same table based on specific conditions. Self-joins are particularly useful for identifying relationships between records within the same table.

Purpose of Self-Joins:

- Identify relationships within a table: Self-joins are used to identify relationships between records within the same table. This is useful for tasks such as finding employees with the same manager or identifying customers who have made multiple purchases.
- Hierarchical data representation: Self-joins can be used to represent hierarchical data structures within a single table. This can be useful for scenarios such as organizational charts or family trees.

- Recursive queries: Self-joins can be used to create recursive queries, which repeatedly join the same table to itself multiple times. This allows for traversing hierarchical data structures or performing iterative calculations.

CROSS JOIN:

- In SQL, a CROSS JOIN, also known as a Cartesian product join, is a type of JOIN that combines each row of one table with each row of another table. It generates a result set that contains all possible combinations of rows from the joined tables. This can result in a large number of rows, especially if the joined tables are large.

Purpose of CROSS JOINS:

- Generate all possible combinations: CROSS JOINs are primarily used to generate all possible combinations of rows from two or more tables. This can be useful for scenarios like creating a list of all possible product combinations or generating a full schedule of events.
- Enumerate possibilities: CROSS JOINs are used to enumerate all possible combinations of values, which can be useful for tasks like generating test cases or exploring different scenarios.
- Create pivot tables: CROSS JOINs can be used to create pivot tables by combining data from multiple tables and summarizing it based on specific criteria.

Left Outer Join:

- A LEFT OUTER JOIN is a type of SQL JOIN that combines rows from two tables based on a related column between them. It ensures that all rows from the left table are included in the result set, regardless of whether they have matching rows in the right table. If there is no match in the right table, the corresponding columns will be filled with NULL values.

Purpose of LEFT OUTER JOINs:

- Include all left table records: LEFT OUTER JOINs ensure that all records from the left table are included in the result set, even if they have no matching records in the right table. This is useful for tasks like retrieving information about all customers, even those who have not placed any orders.

- Enrich left table data: LEFT OUTER JOINs allow you to enrich data from the left table by adding additional information from the right table, even if there is no direct match. This can be useful for creating more comprehensive reports or analyses.
- Identifying unmatched records: LEFT OUTER JOINs can be used to identify records in the left table that do not have matching records in the right table. This can be useful for tasks like finding customers who have not made any recent purchases or identifying products that are not assigned to any category.

Right Outer Join

- A RIGHT OUTER JOIN is a type of SQL JOIN that combines rows from two tables based on a related column between them. It ensures that all rows from the right table are included in the result set, regardless of whether they have matching rows in the left table. If there is no match in the left table, the corresponding columns will be filled with NULL values.

Purpose of RIGHT OUTER JOINs:

- Include all right table records: RIGHT OUTER JOINs ensure that all records from the right table are included in the result set, even if they have no matching records in the left table. This is useful for tasks like retrieving information about all products, even those that have not been sold to any customers.
- Enrich right table data: RIGHT OUTER JOINs allow you to enrich data from the right table by adding additional information from the left table, even if there is no direct match. This can be useful for creating more comprehensive reports or analyses.
- Identifying unmatched records: RIGHT OUTER JOINs can be used to identify records in the right table that do not have matching records in the left table. This can be useful for tasks like finding products that are not listed in any category or identifying employees who have not been assigned to any projects.

FULL OUTER JOIN:

A FULL OUTER JOIN, also known as a FULL JOIN, is a type of SQL JOIN that combines rows from two tables based on a related column between them. It ensures that all rows from both the left and right tables are included in the result set, regardless of whether they have matching rows in the other table. If there is no match in the other table, the corresponding columns will be filled with NULL values.

Purpose of FULL OUTER JOINs:

- Include all rows from both tables: FULL OUTER JOINs ensure that all rows from both the left and right tables are included in the result set, regardless of whether they have matching records in the other table. This is useful for tasks like retrieving a complete list of customers and their corresponding orders, even if some customers have not placed any orders and some orders do not belong to any customer.
- Identifying unmatched records: FULL OUTER JOINs can be used to identify records in either table that do not have matching records in the other table. This can be useful for tasks like finding customers who have not made any purchases or identifying products that have not been sold.
- Comprehensive data retrieval: FULL OUTER JOINs provide a comprehensive view of data from both tables, including unmatched records, which can be useful for analyses and reporting.

Code:

```
SELECT X.RollNo, X.marks,Y.RollNo,Y.subject1,Y.subject2,Y.subject3

FROM Student_functions X join subjects Y

ON X.RollNo=Y.RollNo;
```

-------------------------Non Ansi Join which uses Where Clause----------------------------

- Self Join
- Cross join

```
CREATE TABLE Students_joins (

Admission_No INT PRIMARY KEY IDENTITY(1000,1),

First_Name VARCHAR(45) NOT NULL,

Last_Name VARCHAR(45) NOT NULL,

Age INT,
CITY VARCHAR(25) NOT NULL);



CREATE TABLE FEE (

Admission_no VARCHAR(45) NOT NULL,

Course VARCHAR(45) NOT NULL,

Course VARCHAR(45) NOT NULL,

Amount_Paid INT );

INSERT INTO Students_Joins (first_Name,last_Name,age,city)

VALUES ('Lusia','Evans',13,'Texas'),

('paul','ward',15,'Alaska'),

('peter','Bennet',14,califonia'),

('carlos','peterson',17,'NewYork'),

('Rose','Huges',16,'Florida'),

('Marida','Simmons',15,'Arizona'),

('Antonio','Butter',14,'New York'),

('Diego','Cox',13,'Califonia');

SELECT * FROM Student_Joins;



INSERT INTO FEE(Admission_no,Course,amount_paid)

VALUES(1001,'Java',20000),

(1003,'Android',22000),

(1005,'Python',18000),

(1007,'SQL',15000),
(5112,'Machine Learning',3000);

SELECT * FROM Students_joins;

SELECT * FROM FEE;
```

Inner Join:

```
SELECT S.Admission_No,S.First_Name,S.last_Name,F.Course,F.Course,F.amount_Paid

FROM Students_joins S INNER JOIN FEE F

ON S.Admission_No=F.Admission_No;
```

Outer Join:

```
SELECT S.Admission_no,S.First_Name,S.last_Name,F.Course,F.amount_Paid

FROM Students_Joins OUTER JOIN FEE F

ON S.Admission_no=F.Admission_no--------------------With Alias
```

Left Outer Join

```
SELECT S.admission_no,S.first_name,S.last_name,F.Admission_no,F.Course,F.amount_paid

FROM Students_joins S LEFT OUTER JOIN FEE F

ON S.admission_no=F.admission_no;
```

Right Outer Join:

```
SELECT S.admission_no,S.first_name,S.last_name,F.Admission_no,F.Course,F.Amount_paid

FROM Students_joins Right OUTER JOIN FEE F

ON S.admission_no=F.admission_no;
```

Full Outer Join:

```
SELECT S.Admission_no,S.First_name,S.Last_name,F.Admission_no,F.Course,F.Amount_paid
```

```
FROM Students_Joins S FULL OUTER JOIN FEE F

ON S.admission_no=F.admission_no;
```

-----If you want to find people from the same city-----

```
SELECT S1.Admission_No,S2.First_Name,S1.City

FROM Students_Joins S1,Students_Joins S2

WHERE S1.City=S2.City and S1.Admission_No<>S2.Admission_No

ORDER BY CITY;
```

-----CROSS JOIN-----

```
SELECT S.Admission_no,  S.First_Name, S.Last_name, F.course, F.amount_paid

FROM Students_Joins S

CROSS JOIN FEE F;
```

Triggers:

- In SQL, a trigger is a stored procedure that is automatically executed in response to a specific event, such as an INSERT, UPDATE, or DELETE statement. Triggers are used to enforce business rules, maintain data integrity, and automate certain actions within a database.

Types of Triggers

- DML triggers: These triggers are fired in response to data manipulation language (DML) statements, such as INSERT, UPDATE, and DELETE.
- DDL triggers: These triggers are fired in response to data definition language (DDL) statements, such as CREATE, ALTER, and DROP.

Trigger Timing

- BEFORE INSERT: This trigger executes before a new row is inserted into a table.

- AFTER INSERT: This trigger executes after a new row is inserted into a table.
- BEFORE UPDATE: This trigger executes before an existing row is updated in a table.
- AFTER UPDATE: This trigger executes after an existing row is updated in a table.
- BEFORE DELETE: This trigger executes before an existing row is deleted from a table.
- AFTER DELETE: This trigger executes after an existing row is deleted from a table.

Benefits of Using Triggers

- Encapsulation: Triggers can encapsulate business logic in a single location, making it easier to maintain and reuse.
- Automation: Triggers can automate tasks that would otherwise have to be performed manually.
- Data integrity: Triggers can help to maintain data integrity by enforcing business rules and preventing invalid data from being entered into the database.

Disadvantages of Using Triggers

- Complexity: Triggers can be complex to write and maintain.
- Performance: Triggers can impact the performance of database operations.
- Debugging: Triggers can be difficult to debug.

```
SELECT * FROM EmployeesAudit;

SELECT * FROM Employees_Triggers;

TRUNCATE TABLE Employees;

TRUNCATE TABLE EmployeesAudit;



CREATE TABLE Employees_Triggers (
EmployeeID INTEGER PRIMARY KEY IDENTITY(1,1),

EmployeeName VARCHAR(50),

EmployeeAddress VARCHAR(50),

MonthSalary NUMERIC(10,2));

CREATE TABLE EmployeesAudit (

AuditID INTEGER IDENTITY(1,1),

EmployeeName VARCHAR(50),

EmployeeAddress VARCHAR(50),

MonthSalary NUMERIC(10,2),

ModifiedBy VARCHAR(12),

ModifiedDate DATETIME,

Operation VARCHAR(20));


--------FORM 1: DML TRIGGER FOR INSERT STATEMENT---------

INSERT INTO Employees_Triggers Values (‘Sachin’,’Mumbai’,1000000)

INSERT INTO Employees_Triggers values (‘Mahi’,’Ranchi’,10000000),

(‘kohli’,’Delhi’,1000000000)

(‘Ashwin’,’Chennai’,1000000000)


SELECT * FROM EmployeesAudit;
SELECT * FROM Employees_Triggers



CREATE OR ALTER Trigger TR_Employees_INSERT on dbo.Employees_Triggers

For Insert

As

Begin

Select * from Inserted

Select * from Deleted

If exits(select * from Inserted)

Begin

INSERT INTO dbo.EmployeesAudit
(EmployeeID,EmployeeName,EmployeeAddress,MonthSalary,ModifiedBy,ModifiedDate,Operation
)

SELECT
I.EmployeeID,I.EmployeeName,I.EmployeeAddress,I.MonthSalary,Host_Name(),GETDATE(),'IN
SERT'

FROM Inserted I

END

END



---2-----

SELECT * FROM Employees_Triggers
```

```
CREATE OR ALTER TRIGGER TR_Employees_UPDATE ON dbo.Employees_Triggers

FOR UPDATE

AS

BEGIN

SELECT * FROM INSERTED

SELECT * FROM DELETED

IF EXISTS (SELECT * FROM DELETED)

BEGIN

INSERT INTO dbo.EmployeesAudit
(EmployeeID,EmployeeName,EmployeeAddress,MonthSalary,ModifiedBy,ModifiedDate,Operation
)

SELECT
D.EmployeeID,D.EmployeeName,D.EmployeeAddress,D.MonthSalary,Host_Name(),GETDATE()
,'UPDATE',

FROM deleted D

END

END

UPDATE Employees_Triggers SET EmployeeName='Mahendra Singh Dhoni' where
EmployeeID=2

SELECT * FROM Employees_Triggers

SELECT * FROM employeesaudit

--------3--------
```

```
CREATE OR ALTER TRIGGER TR_Employees_DELETE ON dbo.Employees_triggers

FOR DELETE

AS

BEGIN

SELECT * FROM INSERTED

SELECT * FROM DELETED

IF EXISTS(SELECT * FROM DELETED)

BEGIN

INSERT INTO dbo.EmployeesAudit

(EmployeeID,EmployeeName,EmployeeAddress,MonthSalary,ModifiedBy,ModifiedDate,Operation
)

SELECT
D.EmployeeID,D.EmployeeName,D.EmployeeAddress,D.MonthSalary,HOST_NAME(),GETDATE
(),'DELETE'

FROM DELETED D

END

END

DELETE FROM Employees_Triggers WHERE EmployeeID=1


SELECT * FROM Employees_Triggers;

SELECT * FROM EmployeesAudit;
```

------------VIEWS-------------

VIEW TABLE:

- In SQL, there is no specific "view table" concept. However, there is a database object called a "view." A view is a virtual table that is based on the result of a SELECT query. It does not store the data itself but provides a way to represent the data stored in one or more underlying tables.

```
SELECT * FROM MOCK_DATA

CREATE VIEW VW_MaleCandidates AS

SELECT First_Name,Last_Name,Gender

FROM MOCK_DATA

WHERE Gender='Male'

Select * from VW_Ione

Select * from VW_MaleCandidates



Select * from Students_Joins

Select * from Fee



CREATE VIEW VW_EnrolledStudents AS

Select S.First_Name,S.Last_Name,F.Course

FROM Students_Joins S Full Oute Join Fee F

ON S.Admission_No=F.admission_no

Select * from VW_EnrolledStudents
```

```
Update VW_EnrolledStudents Set First_Name=’Nile’

DROP View VW_EnrolledStudents
```

------Union Operator--------

-----In SQL Server,the UNION Operator is used to combine the result-set of two or more SELECT Statements.

```
SELECT Name from Student_Functions where RollNo<5

UNION

SELECT First_Name FROM MOCK_DATA WHERE ID<10

ORDER BY Name;
```

Full Name:

```
Select Concat(‘Deepak’,’ ‘,Chaha’)as FastBowler

SELECT * FROM MOCK_DATA
```

---5---

- SELECT ID,Concat(First_Name,’ ‘,Last_Name) as FullName,Email,Gender FROM MOCK_DATA ;
- SELECT Concat (First_Name,’ ‘,Last_Name)AS “FullName” FROM MOCK_DATA;

--------------------Cast and Convert----------------------

The CAST function will return a value in which data type we want to convert.

- SELECT CAST(10.95 AS INT) AS Result;
- SELECT CAST(10.95 AS DEC(3,0)) AS Result;

- SELECT CAST('2021-04-26' AS DATETIME) AS Result;
- SELECT GETDATE() AS Todaysdate;
- SELECT CAST(GETDATE() AS VARCHAR) AS Result;