### Syntax Change Between My Sql And Sql Server:

|  |  |
| --- | --- |
| MYSQL | SQL SERVER |
| Developed by Oracle. It is open source. It is freely available | Developed by Microsoft. It is not free. |
| Uses IFNULL() to return a value if the expression is NULL | Uses COALESCE() for the same purpose |
| Uses DATE\_FORMAT() to format dates  Ex. SELECT DATE\_FORMAT(date\_column, '%Y-%m-%d') FROM table\_name; | Uses FORMAT() or CONVERT() for date formatting  Ex. SELECT FORMAT(date\_column, 'yyyy-MM-dd') FROM table\_name; |
| Uses the CONCAT() function for concatenation | Uses the + operator for concatenation. |
| Uses the LIMIT clause to limit the number of rows returned | Uses the TOP clause |
| Uses IFNULL() to return a specified value if the expression is NULL | Uses ISNULL() for the same purpose |
| Uses SUBSTRING() and LENGTH() | Uses SUBSTRING() and LEN() for similar operations |
| String comparisons are case-insensitive by default | String comparisons are case-insensitive by default unless the collation is set to case-sensitive. |
| Uses the IF() function | Uses the CASE expression for similar logic |
| Uses CREATE TEMPORARY TABLE to create temporary tables.  Ex. CREATE TEMPORARY TABLE temp\_table AS SELECT \* FROM table\_name; | Uses # to denote temporary tables.  Ex. CREATE TABLE #temp\_table AS SELECT \* FROM table\_name; |
| Allows using column aliases in ORDER BY, GROUP BY | Does not allows using column aliases in ORDER BY, GROUP BY |

### Below is an example of a CREATE TABLE statement that includes a variety of constraints: PCFUND

**CREATE TABLE** Employees (

EmployeeID INT **IDENTITY**(1, 1) **PRIMARY KEY**, -- Primary Key Constraint

FullName VARCHAR(50) **NOT NULL**, -- NOT NULL Constraint

Email VARCHAR(100) **UNIQUE**, -- UNIQUE Constraint

PhoneNumber VARCHAR(20) **DEFAULT 'Not Provided'**, -- DEFAULT Constraint

BirthDate DATE **CHECK (BirthDate <= HireDate),** -- CHECK Constraint

DepartmentID INT, -- Foreign Key Reference

Salary DECIMAL(10, 2) CHECK (Salary > 0), -- CHECK Constraint

**CONSTRAINT FK\_Department FOREIGN KEY (DepartmentID) -- FOREIGN KEY Constraint**

**REFERENCES Departments(DepartmentID)**

ON DELETE SET NULL );

*🡺Employee\_id is Auto increment/identity-->values are automatically generated when a new row is inserted into the table.*

**Data Definition Language (DDL) commands in SQL are:** (CARD)  
NOTE:- DDL Command must contains the table keyword and DDL command are auto committed i.e **CREATE, DROP, RENAME, ALTER, TRUNCATE, COMMENT**

1. **CREATE** - Used to create a new database object (e.g., table, index, view).
2. **ALTER** - Used to modify an existing database object (e.g., adding or dropping a column in a table).

**🡺ALTER TABLE** table\_name **ADD** column\_name datatype;

**🡺ALTER TABLE** table\_name **DROP** COLUMN column\_name;

**🡺ALTER TABLE** table\_name **MODIFY** column\_name datatype [constraint];

**🡺ALTER TABLE** EMPLOYEE **ADD PRIMARY KEY**(P\_EMPLOYEE\_ID)

**🡺ALTER TAB LE** EMPOLOYEE **ADD FOREIGN KEY(**F\_KEY**)**

**🡺ALTER TABLE** ProjectDetail **ADD CONSTRAINT** fk\_EmployeeDetailID\_Eid **FOREIGN KEY**(EmployeeDetailID)**REFERENCES**EmployeeDetail (EmployeeID) –setting foreign key references to primary key table

1. **DROP** - Used to delete an existing database object (e.g., table, database, view).

DROP TABLE table\_name;

DROP DATABASE database\_name;

DROP VIEW view\_name;

1. **TRUNCATE** - Used to remove all records from a table, while keeping the table structure. Ex. **Truncate Table** table\_name;
2. **RENAME** - Used to rename an existing database object (e.g., table).

🡺 RENAME TABLE Employees TO Staff;

1. **COMMENT** - Used to add a descriptive comment to a database object (e.g., table, column).

* COMMENT ON TABLE Employees IS 'Table to store employee information';
* COMMENT ON COLUMN Employees.HireDate IS 'Date when the employee was hired';

1. **CREATE INDEX** - Used to create an index on a table for improved query performance.

* **CREATE INDEX** idx\_lastname **ON** Employees (LastName);
* **CREATE CLUSTERED INDEX** IX\_CustomerID **ON** Customers(CustomerID);
* **CREATE NONCLUSTERED INDEX** IX\_CustomerSearch **ON** Customers(CustomerName, City, State);
* **CR EATE NONCLUSTERED INDEX** IX\_CustomerName **ON** Customers(CustomerName);

1. **DROP INDEX** - Used to delete an index from a table.

* DROP INDEX idx\_lastname ON Employees;

1. **CREATE VIEW** - Used to create a virtual table based on the result of a query.

* CREATE VIEW ActiveEmployees AS SELECT FirstName, LastName, HireDate

FROM Employees WHERE Active = 1;

1. **DROP VIEW** - Used to delete a view.

* DROP VIEW ActiveEmployees;

**Data Manipulation Language (DML)—(SIDU)**

* **SELECT**: Retrieve data from the database.
* **INSERT**: Add new records to a table and copy data from another table
* **UPDATE**: Modify existing records in a table.
* **DELETE**: Remove records from a table.
* **SELECT** \* **FROM** Employees;
* **INSERT INTO** Employees (EmployeeID, FirstName, LastName) **VALUES** (1, 'John', 'Doe');
* **INSERT INTO** NewTable (Column1, Column2, ...) **SELECT** Column1, Column2, ... FROM ExistingTable **WHERE** [Condition];

* **UPDATE** Employees **SET** LastName = 'Smith’ **WHERE** EmployeeID = 1;
* **DELETE FROM** Employees **WHERE** EmployeeID = 1;

**Data Control Language (DCL)**

* + **GRANT**: Grant privileges to users.
  + **REVOKE**: Revoke privileges from users

🡪GRANT SELECT, INSERT ON Employees TO User1;

🡪REVOKE INSERT ON Employees FROM User1;

**Transaction Control Language (TCL)**

* + **COMMIT**: Save all changes made during the current transaction.
  + **ROLLBACK**: Undo changes made during the current transaction.
  + **SAVEPOINT**: it sets a savepoint such that all the transactions made till the savepoint cannot be rollbacked, but after ones can be.
  + **SET TRANSACTION**: Specify characteristics for the transaction.

**BEGIN** TRANSACTION;

🡺**INSERT INTO** Employees (EmployeeID, FirstName, LastName) **VALUES** (2, 'Jane', 'Doe');

🡺**SAVEPOINT** Savepoint1;

🡺**UPDATE** Employees **SET** LastName = 'Smith' **WHERE** EmployeeID = 2;

🡺**ROLLBACK TO** Savepoint1;

**COMMIT;**

**Query optimization Techniques:**

1. Instead of **SELECT \*,** select only the columns you need.
2. **Create indexes** on columns used in WHERE, JOIN, ORDER BY, and GROUP BY clauses. Too many indexes can lead to increased INSERT, UPDATE, and DELETE operation times. Remove unused indexes.
3. **Joins** are generally **more efficient** than **sub-queries**, especially in complex queries.
4. **Use inner join instead of left join.**
5. The **EXISTS clause** is typically faster than IN or count() for checking the existence of rows in a subquery.
6. Ensure that the columns used in joins are indexed. Place the smallest result set first in a series of joins. **Use Aggregate Functions on Indexed Columns** and Pre-aggregate data in temporary tables CTE or subqueries, then join or filter on the pre-aggregated result.
7. Apply WHERE conditions as early as possible to reduce the number of rows processed by subsequent operations.
8. Use wildcards (%) at the end of a string in LIKE patterns rather than at the start, as it allows the use of indexes.
9. Don’t use <>, != as index will not be used for that and **full table scan** will take place.
10. Use **varchar or nvarchar** instead of nchar and char.
11. Use numeric fields to store the numeric values instead of varchar.
12. Consider using temp tables for complex queries to break them into smaller, more manageable queries.
13. Partition large tables to improve query performance, as SQL Server can work on smaller subsets of data.
14. Cursors are generally slower than set-based operations. Use set-based operations where possible.
15. Avoid using Unnecessary DISTINCT, use **Group by** instead of that.
16. Replace union with union all where we do **not have any duplicate values**.
17. Use an execution plan to check where the query is taking much time to process.

**SSIS Package optimization Techniques:-**

1. Use CTE and temp table instead of sub queries and hitting big tables.
2. Use delta load instead of full load
3. Always apply retention period on data or file extracts.
4. Always run tasks in sequential order instead of parallel.
5. Use query profiling for the query execution plan to identify the slow or inefficient part of query.

### Wildcards in SQL:

* WHERE name LIKE 'A%' — Finds any values that start with "A".
* WHERE name LIKE '%son%' — Finds any values that contain "son" anywhere in the text.
* WHERE name LIKE 'J\_n' — Finds any three-character values that start with "J" and end with "n".
* WHERE name LIKE '[A-C]%' — Finds any values that start with "A", "B", or "C".
* WHERE name LIKE '[^A-C]%' — Finds any values that do not start with "A", "B", or "C".

**1. Aggregate Functions**

* **COUNT()**:SELECT COUNT(\*) FROM Employees;
* **SUM()**:SELECT SUM(Salary) FROM Employees;
* **AVG()**: SELECT AVG(Salary) FROM Employees;
* **MIN()**:SELECT MIN(Salary) FROM Employees;
* **MAX()**:SELECT MAX(Salary) FROM Employees;

**2. String Functions**

* **LEN()**:SELECT LEN(FirstName) FROM Employees;
* **LEFT()**:SELECT LEFT(‘Zaheer’, 3) FROM Employees; --Zah
* **RIGHT()**:SELECT RIGHT(‘Zaheer’, 3) FROM Employees; --eer
* **SUBSTRING()**:SELECT SUBSTRING(‘Zaheer’, 1, 3) FROM Employees; --Zah
* **REPLACE()**:SELECT REPLACE(FirstName, 'John', 'Jonathan') FROM Employees;
* **CHARINDEX()**: Returns the starting position of a substring within a string. SELECT CHARINDEX('Doe', FullName) FROM Employees;
* **CONCAT()**:SELECT CONCAT(‘Zaheer’, ' ', ‘Khan’) FROM Employees; --Zaheer Khan
* **LTRIM()**: Removes **leading spaces** from a string. SELECT LTRIM(**‘ Zaheer’**) FROM Employees; -- Zaheer
* **RTRIM()**: Removes **trailing spaces** from a string. SELECT RTRIM(**‘Zaheer ’**) FROM Employees; --**Zaheer**
* **UPPER()**:SELECT UPPER(‘Shahnawaz’) FROM Employees; --SHAHNAWAZ
* **LOWER()**:SELECT LOWER(‘SHAHnawaz’) FROM Employees; --shahnawaz

**3. Numeric Functions**

* **ABS()**:SELECT ABS(-100) FROM Employees; --100
* **ROUND()**:SELECT ROUND(Salary, 2) FROM Employees; 4521.2500

**🡺**select cast(ROUND(Salary, 2) AS DECIMAL(18,4) from Employees; 4521.25

**🡺**SELECT ROUND(123.4567, 2, 1) AS TruncatedValue; --123.45

* **CEILING()**:SELECT CEILING(1020.45) FROM Employees; --1021
* **FLOOR()**:SELECT FLOOR(1020.45) FROM Employees; --1020
* **RAND()**:SELECT RAND() FROM Employees;
* **POWER()**:-POWER(25,2)

**4. Date and Time Functions**

* **GETDATE()**:SELECT GETDATE();
* **DATEADD()**:SELECT DATEADD(day, 5, GETDATE());
* **DATEDIFF()**:SELECT DATEDIFF(day, StartDate, EndDate) FROM Projects;
* **FORMAT()**:SELECT FORMAT(GETDATE(), 'yyyy-MM-dd') FROM Employees;
* **DATEPART()**:SELECT DATEPART(year, GETDATE()) FROM Employees;
* **DAY()**:SELECT DAY(GETDATE()) FROM Employees;
* **MONTH()**:SELECT MONTH(GETDATE()) FROM Employees;
* **YEAR()**:SELECT YEAR(GETDATE()) FROM Employees;

**5. Conversion Functions:** Both CONVERT and CAST are used to convert data from one data type to another in SQL Server.

* **CAST()**:Simple Syntax and Fully ANSI SQL-compliant.

🡪SELECT CAST(Salary AS VARCHAR) FROM Employees;

* **CONVERT()**:The **style** parameter in CONVERT allows formatting, especially for date/time values and string representations. i.e convert(varchar, getdate(), style)

*🡪SELECT CONVERT(VARCHAR, GETDATE(), 101) FROM Employees;*

**6. Null Functions**

* **ISNULL()**:SELECT ISNULL(Commission, 0) FROM Sales;
* **COALESCE()**: SELECT COALESCE(Commission, Bonus, 0) FROM Sales; returns the not null value of the column if there is no not null value then it returns the last argument as here 0 is last argument.

**[What is the difference between a null field and a blank field?](https://kb.blackbaud.com/articles/Article/39838)**

The value null represents the absence of any object, while the empty string is an object of type String with zero characters.

**7. Logical Functions**

* **IIF()**: Evaluates a condition and returns one of two values based on the result.

SELECT IIF(Salary > 50000, 'High', 'Low') FROM Employees;

* **CASE**: Provides conditional logic within a query.

**SELECT** **CASE WHEN** Salary > 50000 **THEN** 'High' **ELSE** 'Low' **END FROM** Employees;

**8. Window Functions**

* **ROW\_NUMBER()**: Assigns a unique sequential integer to rows within a partition.

SELECT ROW\_NUMBER() OVER (ORDER BY Salary) AS RowNum, FirstName

FROM Employees;

* **RANK()**: Assigns a rank to rows within a partition of a result set. Ex. 122444789

SELECT RANK() OVER (ORDER BY Salary DESC) AS Rank, FirstName FROM Employees;

* **DENSE\_RANK()**: Similar to RANK(), but does not leave gaps in ranking values.

SELECT **DENSE\_RANK()** OVER (ORDER BY Salary DESC) AS DenseRank, FirstName **FROM** Employees; ex. 1122334566777

* **NTILE()**: Divides the result set into a specified number of roughly equal parts.

SELECT NTILE(4) OVER (ORDER BY Salary DESC) AS Quartile, FirstName

FROM Employees;

* Use GROUP BY when you want to aggregate data and return one row per group.
* Use Window Functions (PARTITION BY) when you want to compute aggregate values without collapsing rows.

**Aggregate Functions**: Perform calculations like sum, average, etc., over a specific window of rows.

* SUM()
* AVG()
* MIN()
* MAX()
* COUNT()

select patient\_id,allergies, height,

sum(height) over (partition by allergies) as height\_sum,

avg(height) over (partition by allergies) as height\_avg,

count(height) over (partition by allergies order by height) as height\_count,

min(height) over (partition by allergies order by height) as height\_min,

max(height) over (partition by allergies) as height\_max

from patients where allergies is not null

* if using order by then it will give running value otherwise total value in each row

**Value Functions**: Return specific values from the set of rows.

* **FIRST\_VALUE()**: Returns the first value in an ordered set.
* **LAST\_VALUE()**: Returns the last value in an ordered set.
* **LEAD()**: Provides access to a subsequent row in the same result set(if I’m in 2nd row it will give 3rd row data) without using a self-join.
* **LAG()**: Provides access to a previous row (if I’m in 2nd row it will give 1st row data) in the same result set. Used for calculating commulative sum.

They returns a complete column.

SELECT EmployeeID, Department, Salary,

LEAD(Salary) OVER (PARTITION BY Department ORDER BY Salary DESC) AS NextHigherSalary,

LAG(Salary) OVER (PARTITION BY Department ORDER BY Salary DESC) AS PreviousSalary,

FIRST\_VALUE(Salary) OVER (PARTITION BY Department ORDER BY Salary DESC) AS HighestSalary,

LAST\_VALUE(Salary) OVER (PARTITION BY Department ORDER BY Salary DESC) AS LowestSalary

FROM Employees;

#### How do you handle NULL values in SQL while performing aggregations?

* SELECT COUNT(\*) FROM table\_name; --*Use \* in place of column name to include null value*
* SELECT SUM(COALESCE(column\_name, 0)) FROM table\_name; --use coalesce to convert null to 0
* SELECT avg(COALESCE(col\_name, 0)) FROM table\_n; --use coalesce to convert null to 0 and include the row.

**Finding median value from the list of Lat\_n:-**

SELECT CAST(ROUND(LAT\_N,4) AS DECIMAL(18,4)) FROM

(SELECT LAT\_N,ROW\_NUMBER() OVER (ORDER BY LAT\_N DESC) AS R,

COUNT(\*) OVER() AS TOTAL

FROM STATION) AS P

WHERE R=TOTAL/2+1

**Write a query to calculate a running total using a window function.**

SELECT EmployeeID, SaleDate, SaleAmount,

SUM(SaleAmount) OVER (ORDER BY SaleDate) AS RunningTotal

FROM Sales;

This query calculates a cumulative sum of SaleAmount ordered by SaleDate.

**How would you use window functions to find the top N records in each partition?**

WITH RankedSales AS (

SELECT EmployeeID, SaleDate, SaleAmount,

ROW\_NUMBER() OVER (PARTITION BY EmployeeID ORDER BY SaleAmount DESC) AS RowNum FROM Sales

)

SELECT \* FROM RankedSales WHERE RowNum <= 3;

* This query finds the top 3 sales for each employee by ranking the sales and filtering the results.

**Explain how the NTILE() function works and provide an example of its usage.**

NTILE() distributes rows into a specified number of approximately equal-sized groups (or buckets) and assigns a bucket number to each row. The function can be useful for distributing data into quartiles, deciles, etc.

SELECT

EmployeeID, Salary,

NTILE(4) OVER (ORDER BY Salary DESC) AS Quartile

FROM Employees;

* This query divides the employees into four quartiles based on their salary.

**How can you use window functions to remove duplicate rows while keeping the first occurrence?**

WITH RowNumbered AS (

SELECT EmployeeID, Salary,

ROW\_NUMBER() OVER (PARTITION BY EmployeeID, Salary ORDER BY EmployeeID) AS RowNum

FROM

Employees

)

DELETE FROM Employees

WHERE EmployeeID IN (

SELECT EmployeeID FROM RowNumbered WHERE RowNum > 1

);

```

- This query assigns a unique row number to each duplicate row and then deletes all but the first occurrence.

**Difference between nchar, varchar and char?**

**Unicode:-** nchar, nvarchar. Takes 2 byte for each character. Limited to **ASCII characters or single-byte character sets**.

**Non-Unicode:-** char, varchar. Takes 1 byte for each character. Handles special and multilingual characters.

* **Nchar:-** Fixed length, only Unicode supports. Takes 2 byte for each character.
* **Nvarchar:-** Only stores actual data length, only Unicode supports. Takes 2 byte for each character.
* **Char:-** Fixed length non-Unicode supports. Takes 1 byte for each character.
* **Varchar:-** Only stores actual data length, non-Unicode supports. Takes 1 byte for each character.

## What is a surrogate key in SQL

A surrogate key is defined as a unique identifier for some record or object in a table. It is similar to a primary key, but with a significant difference: it is not derived from the table data – the object generates this key itself.

characteristics of surrogate key are following:-

* It holds a unique value for all records
* It is generated automatically
* It can’t be modified by the user or the application
* It can be used only in the CRUD (Create, Read, Update, Delete) operations.

SQL keywords are reserved words that have specific meanings and functions in SQL queries and statements. Here’s a comprehensive list of SQL keywords along with examples for each:

1. ALL:- Returns true if all of the subquery values meet the condition
2. ALTER:- Adds, deletes, or modifies columns in a table, or changes the data type of a column in a table
3. ALTER COLUMN:- Changes the data type of a column in a table
4. ALTER TABLE:- Adds, deletes, or modifies columns in a table
5. AND:- Only includes rows where both conditions is true
6. ANY:- Returns true if any of the subquery values meet the condition
7. BACKUP DATABASE:- Creates a back up of an existing database
8. BETWEEN:- Selects values within a given range
9. CREATE:- Creates a database, index, view, table, or procedure
10. CREATE DATABASE:- Creates a new SQL database
11. CREATE INDEX:- Creates an index on a table (allows duplicate values)

i.e CREATE INDEX idx\_salary ON Employees (Salary);

1. CREATE OR REPLACE VIEW:- Updates a view
2. CREATE TABLE:-Creates a new table in the database
3. CREATE PROCEDURE:-Creates a stored procedure
4. CREATE UNIQUE INDEX:- Creates a unique index on a table (no duplicate values)
5. CREATE VIEW:-Creates a view based on the result set of a SELECT statement
6. DROP CONSTRAINT:-Deletes a UNIQUE, PRIMARY KEY, FOREIGN KEY, or CHECK constraint
7. DROP DATABASE:-Deletes an existing SQL database
8. DROP DEFAULT:-Deletes a DEFAULT constraint
9. DROP INDEX:-Deletes an index in a table

i.e DROP INDEX idx\_salary ON Employees;

1. **DROP TABLE**:-Deletes an existing table in the database
2. DROP VIEW:- Deletes a view
3. EXEC:-Executes a stored procedure
4. EXISTS:-Tests for the existence of any record in a subquery
5. FOREIGN KEY:-A constraint that is a key used to link two tables together
6. FROM:-Specifies which table to select or delete data from
7. FULL OUTER JOIN:-Returns all rows when there is a match in either left table or right table
8. GROUP BY:-Groups the result set (used with aggregate functions: COUNT, MAX, MIN, SUM, AVG)
9. HAVING:-Used instead of WHERE with aggregate functions
10. IN:-Allows you to specify multiple values in a WHERE clause
11. INDEX:-Creates or deletes an index in a table
12. INNER JOIN:-Returns rows that have matching values in both tables
13. INSERT INTO:- Inserts new rows in a table
14. INSERT INTO SELECT:- Copies data from one table into another table
15. LEFT JOIN:- Returns all rows from the left table, and the matching rows from the right table
16. LIKE:- Searches for a specified pattern in a column
17. LIMIT:- Specifies the number of records to return in the result set
18. OUTER JOIN:- Returns all rows when there is a match in either left table or right table
19. PRIMARY KEY:- A constraint that uniquely identifies each record in a database table
20. PROCEDURE:- A stored procedure
21. RIGHT JOIN:- Returns all rows from the right table, and the matching rows from the left table
22. ROWNUM:- Specifies the number of records to return in the result set
23. UNION:- Combines the result set of two or more SELECT statements (only distinct values)
24. UNION ALL:- Combines the result set of two or more SELECT statements (allows duplicate values)
25. UNIQUE:- A constraint that ensures that all values in a column are unique

**JOIN**

**JOIN:-** Combines rows from two or more tables based on a related column.

**INNER JOIN:-** Returns records that have matching values in both tables.

**LEFT JOIN (or LEFT OUTER JOIN):-** Returns all records from the left table and matched records from the right table.

**RIGHT JOIN (or RIGHT OUTER JOIN):-** Returns all records from the right table and matched records from the left table.

**FULL JOIN (or FULL OUTER JOIN):-** Returns records when there is a match in either left or right table.

**UNION:-** Combines the result sets of two or more SELECT statements.

**INTERSECT:-** Returns the common records from two SELECT statements.

SELECT FirstName FROM Employees

INTERSECT

SELECT FirstName FROM Contractors;

**EXCEPT:-**Returns records from the first SELECT statement that are not in the second SELECT statement.

SELECT FirstName FROM Employees

**EXCEPT** SELECT FirstName FROM Contractors;

* **retrieve all distinct FirstName values from the Employees table that do not exist in the Contractors table**

**GROUP BY:-** Group rows that have the same values into summary rows.

SELECT DepartmentID, COUNT(\*) AS EmployeeCount

FROM Employees

GROUP BY DepartmentID;

**HAVING:-** Filters groups based on a condition (used with GROUP BY).

SELECT DepartmentID, COUNT(\*) AS EmployeeCount

FROM Employees

GROUP BY DepartmentID

HAVING COUNT(\*) > 5;

**NULLIF:-** Returns NULL if two expressions are equal, otherwise returns the first expression.

SELECT NULLIF(Commission, 0) FROM Sales;

**EXISTS:-** Checks if a subquery returns any rows.

SELECT \* FROM Employees

WHERE EXISTS (SELECT 1 FROM Departments WHERE Departments.DepartmentID = Employees.DepartmentID);

**30. ANY and ALL**

* **ANY**: This retrieves all employees where their salary is greater than at least one salary in the subquery.

SELECT \* FROM Employees

WHERE Salary > ANY (SELECT Salary FROM Employees WHERE DepartmentID = 1);

* ALL: Compares a value to all values in a list or result set. The query will return employees whose salary is **greater than every single salary** in department 1.

SELECT \* FROM Employees

WHERE Salary > ALL (SELECT Salary FROM Employees WHERE DepartmentID = 1);

**31. WITH (Common Table Expressions)**

* Defines a temporary result set that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement.

WITH DepartmentCTE AS (

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY DepartmentID

)

SELECT \* FROM DepartmentCTE

WHERE AvgSalary > 50000;

Comparison between **CTE (Common Table Expression)** and **Temp Table**:

| **Aspect** | **CTE (Common Table Expression)** | **Temp Table** |
| --- | --- | --- |
| **Scope** | Limited to the single query or statement where it is defined | Available throughout the session or until explicitly dropped |
| **Lifetime** | Exists only during the execution of the query | Exists for the session or until explicitly dropped |
| **Data Storage** | Data is not physically stored, materialized only during query execution | **Physically stored in the tempdb** database |
| **Reuse** | Cannot be reused across multiple queries | Can be reused across multiple queries within the same session |
| **Performance** | May be faster for small datasets and in-memory operations | Better for handling large datasets; can be indexed for performance |
| **Indexing** | Cannot create indexes directly | Indexes can be created to improve performance |
| **Recursive Queries** | Supports recursion (ideal for hierarchical data) | Does not directly support recursion |
| **Modification (Insert/Update/Delete)** | Cannot modify data directly in the CTE | Can perform INSERT, UPDATE, DELETE operations |
| **Memory Usage** | Typically lower memory usage | Higher memory usage due to physical storage |
| **Best Use Case** | Simplifying complex queries, recursive queries | Handling large datasets, repeated access, data modification |
| **Creation Time** | Defined within the query itself, no explicit creation | Requires explicit creation with CREATE TABLE or SELECT INTO |
| **Drop Required** | Not needed (disappears after query execution) | Needs to be explicitly dropped or auto-dropped after session ends |
| **Transaction Context** | Exists within the transaction that runs the query | Can be used across multiple transactions within the same session |

**WHAT IS A STORED PROCEDURE?**

A stored procedure is a precompiled collection of SQL statements that can be executed as a single unit. They are used to encapsulate logic and are especially handy for frequently performed tasks.

**HOW DO YOU CREATE A SMALL STORED PROCEDURE?**

To create a small stored procedure, you can use the CREATE OR REPLACE PROCEDURE statement in PL/SQL. You define the procedure's name, parameters, and the logic it should execute.

**FUNCTIONS IN PL/SQL:-**

Functions are also crucial in PL/SQL. They are similar to stored procedures but return a single value. Functions are often **used to calculate and return** specific results.

**DIFFERENCE BETWEEN STORED PROCEDURE AND FUNCTIONS?**

Ans. Stored procedures don't necessarily return a value, while functions must return a single value. Functions are typically **used for calculations**, and procedures are for **performing actions**.

**VIEWS WITH EXAMPLE ?**

Ans. Views are virtual tables created from one or more base tables. They provide a way to simplify complex queries.

**WILL THE VIEW UPDATE AUTOMATICALLY IF MAIN TABLES CHANGE?**

Ans. Views are based on the underlying tables. If there are changes in the main tables used in the views, the view's data will reflect those changes automatically when queried.

**What is normalization?**

Normalization is the process of organizing data in a database to minimize redundancy and dependency by dividing large tables into smaller tables and defining relationships between them.

**What is denormalization?**

Denormalization is the process of adding redundant data to a normalized database to improve read performance by reducing the number of joins needed to retrieve data.

**Explain ACID properties in the context of database transactions.**

ACID stands for **Atomicity, Consistency, Isolation, and Durability**. Atomicity ensures that a transaction is treated as a single unit of work, Consistency ensures that the database remains in a consistent state before and after the transaction, Isolation ensures that the transaction is isolated from other transactions, and Durability ensures that the changes made by a transaction are permanent.

**What is indexing in sql server?**

Index is a database object, which can be created on one or more columns. Indexes help SQL Server retrieve the data quicker. When creating the index will read the column(s) and forms a relevant data structure to minimize the number of data comparisons. The index will improve the performance of data retrieval and adds some overhead on data modification such as create, delete and modify. So it depends on how much data retrieval can be performed on table versus how much of DML (Insert, Delete and Update) operations.

**What is the difference between a clustered and a non-clustered index?**

A **clustered index** determines the physical order of rows in a table and is created on the primary key column(s) by default.

*CREATE CLUSTERED INDEX tblEmployee ON [SCHEMA]. tblEmployee (gender asc,salary desc)*

#### Key Differences:

* By default, If table has a primary key, it becomes clustered index and data gets sorted.
* Only one clustered index is allowed per table.
* The clustered index directly affects the physical ordering of data.
* It offers faster retrieval but may slow down insert and update operations.
* You can have only one clustered index in one table, but you can have one clustered index on multiple columns, and that type of index is called a composite index.

A **non-clustered** index does not affect the physical order of rows and is stored separately from the table data.

***CREATE NONCLUSTERED INDEX NC\_EMP\_NAME ON HUMANRESOURCES.EMP(NAME);***

***Key Differences:***

* Multiple non-clustered indexes are allowed per table.
* Non-clustered indexes store data pointers, not the data itself.
* They offer flexibility but may result in slower retrieval compared to clustered indexes.

**Disadvantage of Index?**

Indexes in SQL Server (and other databases) significantly improve query performance by optimizing data retrieval. However, they are not without drawbacks. Below are the **disadvantages of indexes**:

* Indexes require **additional disk space** to store the index structures.
* **Slower Data Modification (INSERT, UPDATE, DELETE)**:- Indexes must be updated whenever data in the indexed columns is modified.
* Having too many indexes can confuse the query optimizer, potentially leading to suboptimal execution plans.
* Indexes are part of the database structure and are included in backups, **increasing backup size and duration.**

**What is the difference between DELETE and TRUNCATE commands?**

DELETE is a DML command used to remove rows from a table based on a specified condition, while TRUNCATE is a DDL command used to remove all rows from a table **without generating a log of individual row deletions.**

**What is a trigger?**

A trigger is a database object that automatically executes a predefined set of SQL statements in response to certain events on a table or view, such as INSERT, UPDATE, or DELETE. A trigger can be delete using a drop as object are deleted by using DROP.

**What are the types of Triggers in SQL?**

* **Before Trigger:** Executes before the triggering event (like before an INSERT, UPDATE, or DELETE operation).
* **After Trigger:** Executes after the triggering event.
* **Instead Of Trigger:** Executes instead of the triggering event, typically used with views.
* **DDL Trigger:** Fires in response to Data Definition Language (DDL) events such as CREATE, ALTER, or DROP.
* **Logon Trigger:** Fires in response to logon events.

**Explain the concept of INSTEAD OF Triggers.**

An INSTEAD OF trigger allows you to define custom actions that should occur instead of the default operation for INSERT, UPDATE, or DELETE statements, often used with views to update underlying base tables.

**What is the difference between a BEFORE and AFTER Trigger?**

* **BEFORE Trigger:** Executes before the database action, allowing you to validate or modify data before it is inserted, updated, or deleted.
* **AFTER Trigger:** Executes after the database action, often used to audit changes or enforce complex integrity rules.

**How do you prevent recursive triggers in SQL?**

Recursive triggers occur when a trigger fires and causes another trigger to fire, potentially leading to an infinite loop. To prevent this:

* + Use the DISABLE TRIGGER option for specific sessions or triggers.
  + Add logic within the trigger to check for recursion, such as using a session variable.

**What is a Cursor in SQL?**

A cursor is a database object used to retrieve, manipulate, and navigate through a result set one row at a time. Cursors are typically used in SQL when **row-by-row processing** is required.

**Usecases of Cursors in SQL?**

Cursors are used to perform row-by-row operations that are not possible with set-based operations. This is useful for tasks that require processing each row individually:-

1. send an email in a sequential manner using row-by-row data
2. Updating a column in each row based on the calculation of running totals.
3. generate dynamic SQL commands or execute stored procedures based on values from each row.
4. When rows need to be processed in batches to avoid locking large numbers of rows or to break large transactions into smaller units.

**1. Declare the Cursor**

DECLARE cursor\_name CURSOR FOR SELECT column1, column2, ...

FROM table\_name WHERE condition;

**2. Open the Cursor**

OPEN cursor\_name;

**3. Fetch the First Row from the Cursor**

FETCH NEXT FROM cursor\_name INTO @variable1, @variable2, ...;

**4. Loop Through Rows WHILE @@FETCH\_STATUS = 0 BEGIN**

Process the data (using @variable1, @variable2, ...)

Fetch the next row FETCH NEXT FROM cursor\_name INTO @variable1, @variable2, ...; END;

**5. Close the Cursor**

CLOSE cursor\_name;

**6. Deallocate the Cursor**

DEALLOCATE cursor\_name;

**What are the types of Cursors in SQL?**

* **Implicit Cursor:** Automatically created by SQL Server whenever a SQL statement is executed.
* **Explicit Cursor:** Manually defined by the programmer to handle specific rows returned by a query.
* **Static Cursor:** The result set is fixed when the cursor is opened.
* **Dynamic Cursor:** The result set reflects changes made to the database while the cursor is open.
* **Forward-Only Cursor:** The cursor can only move forward through the result set.
* **Keyset-Driven Cursor:** The set of keys that define the result set is fixed, but the data can change.

### Explain the life cycle of a Cursor in SQL?

* **Declare:** Define the cursor with a SQL statement.
* **Open:** Execute the SQL statement and populate the cursor with rows.
* **Fetch:** Retrieve rows from the cursor one at a time.
* **Close:** Release the cursor when done to free up resources.
* **Deallocate:** Remove the cursor definition and reclaim memory.

1. **What are the advantages and disadvantages of using Cursors?**

* **Advantages:**
  + Allows row-by-row processing.
  + Useful for complex logic that cannot be achieved with set-based operations.
* **Disadvantages:**
  + Slower performance compared to set-based operations.
  + More resource-intensive, leading to potential issues with memory and locking.

1. **How can you improve the performance of Cursors in SQL?**
   * Minimize the use of cursors by opting for set-based operations when possible.
   * Use the least restrictive cursor type, such as a forward-only cursor.
   * Limit the number of rows returned by the cursor’s query.
   * Avoid using cursors in large data sets.
2. **What is the difference between a Static Cursor and a Dynamic Cursor?**
   * **Static Cursor:** The result set is fixed at the time the cursor is opened. Changes to the underlying data after the cursor is opened are not visible.
   * **Dynamic Cursor:** Reflects changes made to the data while the cursor is open. If a row is updated, deleted, or inserted, these changes are visible.
3. **What happens if you do not close or deallocate a Cursor?**
   * If you do not close or deallocate a cursor, it can lead to memory leaks and resource locking, potentially causing performance issues in the database system. Always close and deallocate cursors after their use to release resources.
4. **Can you use a Cursor in a Stored Procedure?**
   * Yes, cursors can be declared, opened, fetched, and closed within a stored procedure. They are often used in stored procedures when a set of operations needs to be performed on a row-by-row basis.
5. **Explain how to handle errors in Cursors.**
   * Error handling in cursors can be managed using TRY...CATCH blocks in SQL Server. If an error occurs during any cursor operation (such as fetching rows), the error can be caught and handled appropriately within the CATCH block.
6. **What is the difference between a Forward-Only Cursor and a Keyset-Driven Cursor?**
   * **Forward-Only Cursor:** The cursor can only move forward through the result set; you cannot scroll backward.
   * **Keyset-Driven Cursor:** The keys that define the rows in the result set are fixed when the cursor is opened, but the data in the result set can be updated as changes occur in the database.

**When to Use Cursor:**

* **Complex Calculations:** When each row requires a different and complex calculation.
* **Sequential Processing:** When operations need to be done in a specific order, such as updating records in a specific sequence.
* **Logging or Auditing:** When each operation must be logged or audited for compliance purposes.

**Transactions and ACID Properties**

* **Transaction**: A sequence of database operations that must be executed as a single unit.
* **ACID Properties**:
  + **Atomicity**: Transactions are all-or-nothing.
  + **Consistency**: Ensures data integrity before and after the transaction.
  + **Isolation**: Transactions are isolated from each other, preventing conflicts.
  + **Durability**: Once a transaction is committed, it is permanent.

**Database Normalization**

* **Normalization**: Normalization is the process of organizing a database in such a way that it reduces redundancy and dependency by dividing large tables into smaller, more manageable ones.
  + **1NF (First Normal Form)**: Ensures that all values in a column are **atomic** and All entries in a column must be of the same data type. If any column contains more than one value then equal no. of row will be created. Ex. Multiple mobile number should be arranged in different rows.
  + **2NF (Second Normal Form)**: Ensure that the table is **in 1NF and remove partial dependencies.** In 2NF, every non-key attribute must depend on the whole primary key, not just part of it.student\_id🡪 student\_name, Course\_id🡪 instructor name then it should be like student\_id,course\_id🡪 instructor name
  + **3NF (Third Normal Form)**: Ensure the table is in **2NF and No transitive dependencies**. In 3NF, all non-key attributes should be directly dependent on the primary key, and no non-key attribute should depend on another non-key attribute. If EMP\_ID🡪 EMP\_NAME,DEPT, DEPT\_HEAD THEN EMP\_ID🡪 EMP\_NAME,DEPT AND DEPT🡺 DEPT\_HEAD
  + **BCNF (Boyce-Codd Normal Form)**: Ensure the table is in 3NF and Every determinant **must be a candidate key** i.e no non-key attribute should depend on another non-key attribute. For every functional dependency (A → B) in the relation, A must be a superkey.

**Superkey:** A **superkey** is any set of attributes (one or more) that can uniquely identify a row in a table. It can include unnecessary or additional attributes beyond what is required for unique identification. A table can have many superkeys, including the primary key itself.

**Primary Key:** A **primary key** is a minimal superkey, meaning it is the smallest set of attributes that uniquely identifies a record in a table.It cannot have any unnecessary or redundant attributes.There can only be **one primary key** per table.

**Composite Key:** A **composite key** is a combination of two or more columns in a table that together uniquely identify each row.

A **candidate key** is a set of one or more columns (attributes) in a table that can uniquely identify a record (row) in that table. It is a minimal superkey, meaning that no subset of the candidate key can uniquely identify the record.

**Benefits of Normalization**

1. **Reduced Data Redundancy**: Avoids storing the same data in multiple places, reducing storage needs.
2. **Improved Data Integrity**: Prevents data anomalies such as inconsistencies or incomplete data.
3. **Enhanced Data Maintainability**: Easier to update, insert, or delete data without causing issues.
4. **Optimized Queries**: Smaller, well-structured tables can improve query performance.

**Challenges of Normalization**

1. **Complex Queries**: As data is split across multiple tables, joins become more frequent, which can sometimes degrade performance.
2. **Performance Trade-offs**: In highly normalized databases, read performance may suffer, which is why **denormalization** is sometimes applied for optimization in read-heavy applications (e.g., data warehouses).

A **deadlock** in a Database Management System (DBMS) occurs when two or more transactions are waiting for each other to release locks on resources, creating a cycle of dependency where none of the transactions can proceed. This situation leads to a standstill where each transaction is blocked indefinitely unless the deadlock is resolved.

### ****How Deadlocks Occur in DBMS****

Deadlocks typically occur when multiple transactions are competing for the same set of resources, such as database rows or tables, and they acquire locks in an inconsistent order. For example:

* **Transaction A** locks **Resource X** and waits for **Resource Y** to be released.
* **Transaction B** locks **Resource Y** and waits for **Resource X** to be released.
* Neither transaction can proceed because each is waiting for a resource held by the other, leading to a deadlock.

### ****Conditions for Deadlock (Coffman’s Conditions)****

Deadlocks occur when the following four conditions are met simultaneously:

1. **Mutual Exclusion**: A resource can be held by only one transaction at a time.
2. **Hold and Wait**: Transactions holding resources can request additional resources.
3. **No Preemption**: Resources cannot be forcibly taken away from a transaction; the transaction must release them voluntarily.
4. **Circular Wait**: A circular chain of transactions exists, where each transaction is waiting for a resource held by the next transaction in the chain.

### ****Deadlock Handling Techniques****

There are several strategies to handle deadlocks in DBMS:

#### 1. ****Deadlock Prevention****

* The system ensures that at least one of the four conditions for deadlock cannot occur, thereby preventing deadlock.
  + **Wait-Die Scheme**: A transaction is allowed to wait only if it has a higher priority (older transaction) than the transaction holding the requested resource. Otherwise, it is aborted and restarted later.
  + **Wound-Wait Scheme**: A transaction with a higher priority can preempt (forcefully abort) the transaction holding the resource. The aborted transaction is restarted later.
* Other techniques:
  + **No Hold and Wait**: Require transactions to request all resources upfront, ensuring they either get all the resources at once or none at all.
  + **Prevention of Circular Wait**: Impose an order on resource acquisition (e.g., each transaction must request resources in a specific order to prevent circular waits).

#### 2. ****Deadlock Avoidance****

* **Wait-for Graph (WFG)**: The DBMS uses a directed graph where each node represents a transaction, and an edge from T1 to T2 means T1 is waiting for a resource held by T2. If the graph contains a cycle, a deadlock exists.
* **Banker’s Algorithm**: A resource allocation strategy that checks whether granting a resource request will leave the system in a safe state. If it leads to a potential deadlock, the request is denied.

#### 3. ****Deadlock Detection****

* **Deadlock Detection** allows deadlocks to occur but periodically checks for them and resolves them by aborting one or more transactions.
* The system periodically checks for deadlocks by constructing a **wait-for graph** and searching for cycles.
* When a deadlock is detected, the system breaks the cycle by aborting one or more transactions (known as **victim selection**). The aborted transaction(s) release their locks, allowing other transactions to proceed.

#### 4. ****Deadlock Recovery****

* Once a deadlock is detected, the system must recover by aborting one or more transactions. The **victim** is chosen based on:
  + The transaction with the fewest resources held.
  + The transaction that has done the least amount of work (least cost to abort and restart).
  + The transaction with the lowest priority.
* After aborting a transaction, the system may either:
  + **Restart the transaction** after a delay or after all other transactions have completed.
  + **Rollback the transaction** to an earlier savepoint before the deadlock occurred, if supported by the DBMS.

### ****Deadlock Prevention vs. Deadlock Detection****

* **Deadlock Prevention** tries to ensure that deadlocks never occur by designing the system to avoid circular waits or by imposing an ordering on resource acquisition.
* **Deadlock Detection** allows deadlocks to occur but periodically checks for them and resolves them by aborting one or more transactions.

|  |  |  |
| --- | --- | --- |
| Player-1 | Player-2 | Winner |
| Rock | Scissor | Rock |
| Paper | Scissor | Scissor |
| Rock | Paper | Paper |
|  |  |  |

Rock beats scissors, scissors beats paper, paper beats rocks

**What is Stored Procedures?**

A Stored Procedure is a sequence statement in PL/SQL. It can do some data processing. You can use a stored procedure as a schema object in the Oracle database. You can invoke them by triggers, nested, invoked, or parameterized.

**What is the default syntax of the stored procedures?**

CREATE OR REPLACE PROCEDURE <your\_procedure>  
(  
<parameterl IN/OUT <datatype>  
...  
)  
[ IS | AS ]<declaration\_part>  
BEGIN  
<execution part>  
EXCEPTION  
<exception handling part>  
END;

**What do you know about stored functions?**

You can call it also User Function or User Defined Function. Simply it is a set of PL/SQL statements that you can call by name. They are stored and compiled in a database. This function will return a value to the environment in which it is called.

**What is the default syntax of the stored functions?**

CREATE OR REPLACE FUNCTION <your\_procedure>  
(  
<parameterl IN/OUT <datatype>  
)  
RETURN <datatype>  
[ IS | AS ]<declaration\_part>  
BEGIN  
<execution part>   
EXCEPTION  
<exception handling part>  
END;

**How many modes are available to pass parameters?**

There are three types of modes are available:

* IN: It is the default parameter that will receive input value from the program.
* OUT: It will send output value to the program.
* IN OUT: It is the combination of both IN and OUT. Thus, it receives from, as well as sends a value to the program.

**How many stored procedures are there?**

There are two types of stored procedures:

* System Stored Procedures
* User Defined Stored Procedures

**What is purpose of recursive stored procedure?**

It can be called by itself in SQL server by the User. This particular store procedure helps to solve repetitive problems. You can go next up to 32 levels.

**What do you know about System Stored Procedures?**

It is a very useful one of the stored procedures. It helps to extract the definition and all dependencies of the stored procedures. There are three types of System stored procedures.

* sp\_help to show all details
* sp\_helptext  to show the definition
* sp\_depends to show all dependencies

Exec sp\_helptext tblEmployee;

**Different ways to call a particular stored procedure?**

You can call them in three different ways:

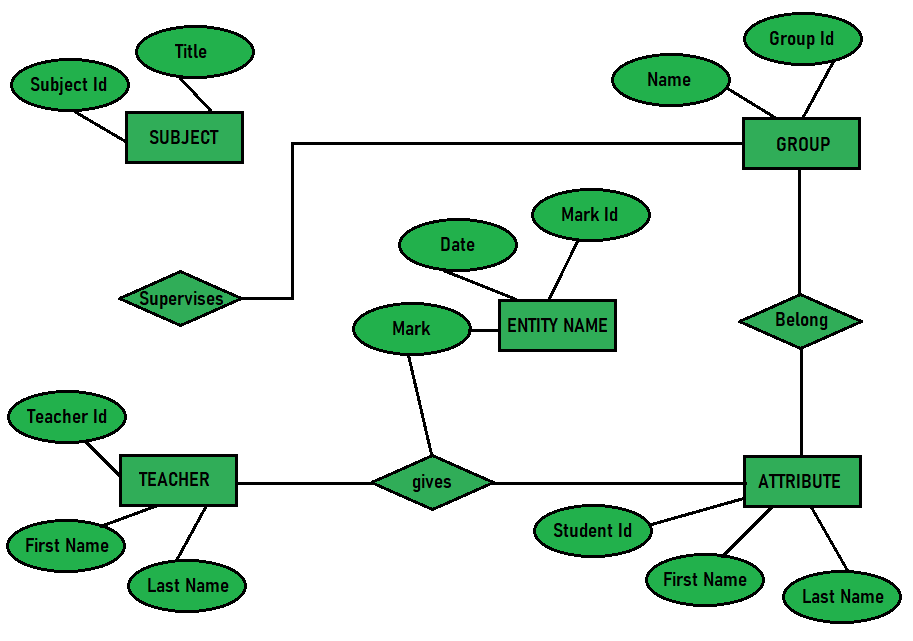
1. By using EXECUTE in SQL prompt
2. Using CALL statement in SQL prompt
3. By using within another subprogram

**Difference between a function and a stored procedure in SQL** presented in a tabular form:

| **Feature** | **Function** | **Stored Procedure** |
| --- | --- | --- |
| **Purpose** | Used for computations and returning values. | Used for executing business logic and operations. |
| **Returns Value** | Always returns a value (scalar, table, or composite). | May return zero or more values via OUTPUT parameters. |
| **Invocation** | Called from SQL statements (e.g., SELECT, WHERE, FROM). | Called using EXEC or EXECUTE command. |
| **Input Parameters** | Must have input parameters. | Can have input and output parameters. |
| **Output Parameters** | Not supported. | Supported. |
| **Data Modification** | Cannot perform INSERT, UPDATE, or DELETE (except with inline table-valued functions). | Can perform INSERT, UPDATE, DELETE, and other operations. |
| **Transactions** | Cannot manage transactions (e.g., BEGIN TRAN, COMMIT). | Can manage transactions. |
| **Deterministic Behavior** | Should be deterministic (same output for the same input). | May include non-deterministic operations (e.g., random values, logging). |
| **Side Effects** | No side effects (does not alter database state). | Can have side effects (modifies database state). |
| **Usage in SQL Queries** | Can be used in SQL queries like SELECT, WHERE, or JOIN. | Cannot be directly used in queries. |
| **Error Handling** | Limited error handling. | Supports full error handling using TRY-CATCH. |
| **Performance** | Lightweight and faster for computations. | Suitable for more complex logic and bulk operations. |
| **Example Syntax** | SELECT dbo.FunctionName(params) | EXEC ProcedureName params |

**Differences Between DFD and ERD**

| **DFD** | **ERD** |
| --- | --- |
| It stands for [Data Flow Diagram.](https://www.geeksforgeeks.org/what-is-dfddata-flow-diagram/) | It stands for [Entity Relationship Diagram](https://www.geeksforgeeks.org/introduction-of-er-model/) or Model. |
| Main objective is to represent the processes and data flow between them. | Main objective is to represent the data object or entity and relationship between them. |
| It explains the flow and process of data input, data output, and storing data. | It explains and represent the relationship between entities stored in a database. |
| Symbols used in DFD are: rectangles (represent the data entity), circles (represent the process), arrows (represent the flow of data), ovals or parallel lines (represent data storing). | Symbols used in ERD are: rectangles (represent the entity), diamond boxes (represent relationship), lines and standard notations (represent [cardinality](https://www.geeksforgeeks.org/cardinality-in-dbms/)). |
| Rule followed by DFD is that at least one data flow should be there entering into and leaving the process or store. | Rule followed by ERD is that all entities must represent the set of similar things. |
| It models the flow of data through a system. | It model entities like people, objects, places and events for which data is stored in a system. |



**E-R DIAGRAM**

1. **How to delete two tables using one Drop command?**

Ans: DROP TABLE table1, table2 CASCADE;

#*If there are foreign key constraints or other dependencies between tables, you might need to drop constraints first or use CASCADE to remove dependencies*

1. **How do you copy data from one table to another table?**

ANS:- INSERT INTO target\_table (column1, column2, ...) SELECT column1, column2, ... FROM source\_table WHERE condition;

1. **How do you create new table from another table?**

ANS:- CREATE TABLE employees\_backup AS SELECT \* FROM employees;

1. **What is nolock hint in sql server?**

Ans:- SELECT column1, column2 FROM table\_name WITH (NOLOCK);

#*Queries with NOLOCK do not block other processes, No Locks Acquired*

1. **How to rename Database?**

**ANS:-** ALTER DATABASE OldTestDB MODIFY NAME = NewTestDB;

While renaming database, Make sure no users are connected to the database to avoid conflicts. Set the database to **single-user mode** if necessary: Steps are following:-

1. ALTER DATABASE OldDatabaseName SET SINGLE\_USER WITH ROLLBACK IMMEDIATE;
2. ALTER DATABASE OldDatabaseName MODIFY NAME = NewDatabaseName;
3. ALTER DATABASE NewDatabaseName SET MULTI\_USER;
4. Can we store Image, MP3 and binary data in SQL Server?

Yes, you can store **images**, **MP3 files**, and other **binary data** in SQL Server. These types of data are typically stored using **VARBINARY(MAX)** data types designed for handling binary content.

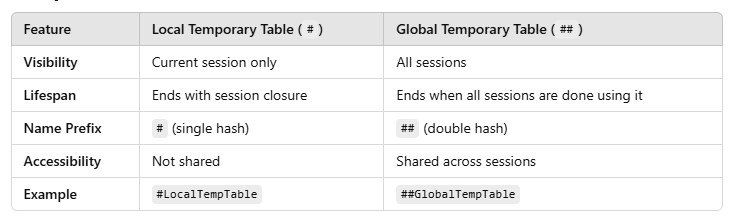
CREATE TABLE MediaStore (

MediaID INT PRIMARY KEY,

MediaName NVARCHAR(255),

MediaData VARBINARY(MAX));

1. **Local Temp Table vs Global Temp Table:-** Both types are stored in the **tempdb** database, Dropped manually using DROP TABLE.



1. **What are the different types of System tables?**

|  |  |
| --- | --- |
|  | **System Catalog Views**   1. **Database Information**: sys.databases, sys.tables, sys.views 2. **Column and Index Information**: sys.columns, sys.indexes 3. **Security**: sys.database\_principals, sys.server\_principals |
|  | **Dynamic Management Views (DMVs) and Functions**   1. **Performance**: sys.dm\_exec\_query\_stats, sys.dm\_exec\_requests 2. **Index Usage**: sys.dm\_db\_index\_usage\_stats 3. **System Health**: sys.dm\_os\_performance\_counters |
|  | **Get a List of All User Tables:-** SELECT name FROM sys.tables WHERE type = 'U'; |
|  | **Get Column Information for a Table:-** SELECT COLUMN\_NAME, DATA\_TYPE FROM INFORMATION\_SCHEMA.COLUMNS WHERE TABLE\_NAME = 'YourTableName'; |
|  | **Check Database Properties:-** SELECT name, state\_desc, recovery\_model\_desc FROM sys.databases; |
|  | 1. **sys.server\_principals**:- Lists all server-level principals (logins, users, and roles). 2. **sys.sql\_logins:-**Contains information about SQL Server logins, including password policies. 3. **sys.database\_principals:-** Lists database-level principals, such as database users and roles. |
|  | 1. **sys.dm\_exec\_sessions**:- Shows **all Active sessions** connected to the SQL Server instance.   🡪 *SELECT session\_id, login\_name, status, last\_request\_start\_time FROM sys.dm\_exec\_sessions;*   1. **sys.dm\_exec\_connections**:-Provides details about **all active connections**.   *SELECT session\_id, client\_net\_address, connect\_time FROM sys.dm\_exec\_connections;*   1. **sys.dm\_exec\_requests**:- Displays information about **executing queries** for active sessions.   *SELECT session\_id, status, command, start\_time, sql\_text.text FROM sys.dm\_exec\_requests* |
|  | 1. **sys.server\_audits:-** Lists server-level audit definitions. 2. **sys.database\_audit\_specifications:-** Lists audit specifications for each database. |
|  | **Monitor query performance:-** SELECT sql\_handle, execution\_count, total\_worker\_time, total\_logical\_reads FROM **sys.dm\_exec\_query\_stats** ORDER BY total\_worker\_time DESC; |

1. **What are different Types of Sub-Queries?**
2. **Single-Row Sub-Query:-** Returns exactly one row with one or more columns, Often used in the WHERE or HAVING.

**🡪** *SELECT name, salary FROM employees WHERE salary > ( SELECT AVG(salary) FROM employees );*

1. **Multi-Row Sub-Query:-** Returns multiple rows but only one column, Use IN, ANY, ALL, or comparison operators.

🡪SELECT name FROM employees WHERE department\_id IN ( SELECT department\_id FROM departments WHERE location = 'New York' );

1. **Multi-Column Sub-Query:-** Returns multiple rows and multiple columns. Typically used in WHERE or FROM clauses.

🡪SELECT name FROM employees WHERE (department\_id, job\_id) IN ( SELECT department\_id, job\_id FROM job\_openings);

1. **Correlated Sub-Query:-** A sub-query that references columns from the outer query. It is executed once for each row of the outer query.

🡪SELECT name, salary FROM employees e1 WHERE salary > ( SELECT AVG(salary) FROM employees e2 WHERE e1.department\_id = e2.department\_id );

1. **Non-Correlated Sub-Query:-** A sub-query independent of the outer query. It is executed once and the result is used in the outer query.

🡪 SELECT name FROM employees WHERE department\_id = ( SELECT department\_id FROM departments WHERE department\_name = 'HR' );

### ****What is an Execution Plan?****

An **execution plan** is a detailed map generated by the database engine that shows how a SQL query will be executed. It includes the steps and methods the database uses to retrieve, filter, sort, and join data.

### ****Why Use an Execution Plan?****

Execution plans help in:

1. **Performance Tuning**: Identifying inefficiencies, such as full table scans or suboptimal joins.
2. **Index Optimization**: Determining whether indexes are being used properly.
3. **Query Troubleshooting**: Understanding why a query is running slowly.
4. **Identifying Bottlenecks**: Revealing operations like high-cost joins, sorts, or scans.

### ****How to View the Execution Plan in SQL Server?****

SQL Server provides multiple ways to view execution plans:

* **Estimated Execution Plan** (Static):
  + Shows the plan without executing the query.
  + Use when you want to analyze a query without impacting the database.
  + Command:

SET SHOWPLAN\_ALL ON;

GO

SELECT \* FROM your\_table;

SET SHOWPLAN\_ALL OFF;

GO

* **Or, in SQL Server Management Studio (SSMS**): Press Ctrl + L or click on Display Estimated Execution Plan.
* **Actual Execution Plan** (Dynamic):
  + Shows the plan after executing the query, with runtime statistics.
  + Use for analyzing actual resource usage.
  + Command:

SET STATISTICS XML ON;

SELECT \* FROM your\_table;

SET STATISTICS XML OFF;

* **Or, in SSMS:-** Enable Include Actual Execution Plan (Ctrl + M) and run the query.

### ****Understanding Execution Plan Components****

Key elements in an execution plan include:

1. **Table Scan**:- A full scan of the table (inefficient for large datasets).
2. **Index Scan/Seek**:- Indicates use of an index (faster and preferred).
3. **Joins**:- Methods like nested loops, hash joins, or merge joins.
4. **Sorting**:- Sorting steps, which can be costly without proper indexes.
5. **Estimated and Actual Costs**:- Indicates resource usage, such as CPU, memory, and I/O.

### ****When Would You Use It?****

1. **Slow Query Diagnosis**: If a query is taking too long to execute.
2. **Index Optimization**: To determine if indexes are being used effectively.
3. **Understanding Query Logic**: To see how the database interprets and executes your query.
4. **Resource Bottleneck Identification**: To spot high-cost operations or excessive resource consumption.