**SQL Query Syntaxes**

**Creating EMP Table:**

CREATE TABLE EMP (

EmpID INT IDENTITY(1,1) PRIMARY KEY,

FirstName NVARCHAR(50) NOT NULL,

LastName NVARCHAR(50) NOT NULL,

JobTitle NVARCHAR(100),

Department NVARCHAR(50),

HireDate DATE,

Salary DECIMAL(10, 2),

Phone NVARCHAR(15),

Email NVARCHAR(100) UNIQUE,

ManagerID INT,

IsActive BIT DEFAULT 1,

CreatedAt DATETIME DEFAULT GETDATE(),

UpdatedAt DATETIME NULL

);

**Example Foreign Key Constraint for ManagerID referencing the same table**

ALTER TABLE EMP

ADD CONSTRAINT FK\_ManagerID FOREIGN KEY (ManagerID) REFERENCES EMP (EmpID);

**Explanation of Columns:**

1. **EmpID**: Auto-incrementing primary key for unique identification of employees.
2. **FirstName** and **LastName**: Store the employee's first and last names.
3. **JobTitle**: The role or title of the employee.
4. **Department**: The department where the employee works.
5. **HireDate**: The date the employee was hired.
6. **Salary**: The salary of the employee.
7. **Phone**: Contact number of the employee.
8. **Email**: Unique email address.
9. **ManagerID**: References the employee's manager; can be null for top-level managers.
10. **IsActive**: Indicates whether the employee is active (1) or inactive (0).
11. **CreatedAt** and **UpdatedAt**: Timestamps for record tracking.

**Insert Values:**

INSERT INTO EMP (FirstName, LastName, JobTitle, Department, HireDate, Salary, Phone, Email, ManagerID)

VALUES

('John', 'Doe', 'Software Engineer', 'IT', '2023-01-15', 60000, '123-456-7890', 'john.doe@example.com', NULL),

('Jane', 'Smith', 'Project Manager', 'IT', '2020-06-10', 90000, '987-654-3210', 'jane.smith@example.com', 1),

('Alice', 'Johnson', 'HR Manager', 'HR', '2018-03-25', 75000, '456-789-0123', 'alice.johnson@example.com', NULL);

**Query with WHERE Clause**

**Retrieve all employees in the "IT" department earning a salary greater than $50,000:**

SELECT EmpID, FirstName, LastName, JobTitle, Department, Salary

FROM EMP

WHERE Department = 'IT' AND Salary > 50000;

**.Query with GROUP BY Clause**

**Find the total salary paid to employees grouped by their department:**

sql

Copy code

SELECT Department, SUM(Salary) AS TotalSalary

FROM EMP

GROUP BY Department;

**Using GROUP BY with HAVING**

**Retrieve departments where the total salary exceeds $100,000:**

sql

Copy code

SELECT Department, SUM(Salary) AS TotalSalary

FROM EMP

GROUP BY Department

HAVING SUM(Salary) > 100000;

**Combining WHERE and GROUP BY**

**Find the number of employees in each department hired after 2020:**

sql

Copy code

SELECT Department, COUNT(EmpID) AS EmployeeCount

FROM EMP

WHERE HireDate > '2020-01-01'

GROUP BY Department;

**Query with Aggregate and GROUP BY**

**Retrieve the highest-paid employee in each department:**

sql

Copy code

SELECT Department, MAX(Salary) AS MaxSalary

FROM EMP

GROUP BY Department;

Here are examples of RANK and DENSE\_RANK queries for the EMP table:

**RANK Query**

**Retrieve a ranked list of employees based on their salaries within each department. Use RANK to assign ranks, allowing gaps in ranking for ties:**

sql

Copy code

SELECT

EmpID,

FirstName,

LastName,

Department,

Salary,

RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS Rank

FROM EMP;

**Explanation:**

* PARTITION BY Department: Groups employees by department.
* ORDER BY Salary DESC: Orders salaries in descending order within each department.
* RANK(): Assigns a rank; tied salaries will have the same rank, but the next rank will skip numbers (e.g., 1, 2, 2, 4).

**DENSE\_RANK Query**

**Retrieve a dense-ranked list of employees based on their salaries within each department. Use DENSE\_RANK to assign ranks without gaps for ties:**

sql

Copy code

SELECT

EmpID,

FirstName,

LastName,

Department,

Salary,

DENSE\_RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS DenseRank

FROM EMP;

**Explanation:**

* **Similar to RANK(), but DENSE\_RANK() does not skip ranks for ties. For example, if two employees tie for rank 2, the next rank will be 3 (e.g., 1, 2, 2, 3).**

**Comparison of RANK and DENSE\_RANK**

If the EMP table contains the following sample data:

| **EmpID** | **FirstName** | **Department** | **Salary** |
| --- | --- | --- | --- |
| 1 | John | IT | 70000 |
| 2 | Jane | IT | 60000 |
| 3 | Alice | IT | 60000 |
| 4 | Bob | HR | 75000 |

* **RANK** Result:

| **EmpID** | **Department** | **Salary** | **Rank** |
| --- | --- | --- | --- |
| 1 | IT | 70000 | 1 |
| 2 | IT | 60000 | 2 |
| 3 | IT | 60000 | 2 |
| 4 | HR | 75000 | 1 |

* **DENSE\_RANK** Result:

| **EmpID** | **Department** | **Salary** | **DenseRank** |
| --- | --- | --- | --- |
| 1 | IT | 70000 | 1 |
| 2 | IT | 60000 | 2 |
| 3 | IT | 60000 | 2 |
| 4 | HR | 75000 | 1 |

**Set Oprators:**

**Set operators in SQL (such as UNION, UNION ALL, INTERSECT, and EXCEPT) allow combining results from two or more queries. Here's how you can apply them to the EMP table:**

**1. UNION**

**Combine results from two queries and eliminate duplicates.**

**Example: Retrieve unique departments from employees earning more than $60,000 and from managers.**

sql

Copy code

SELECT Department

FROM EMP

WHERE Salary > 60000

UNION

SELECT Department

FROM EMP

WHERE ManagerID IS NOT NULL;

**2. UNION ALL**

**Combine results from two queries without removing duplicates.**

**Example: Retrieve all departments from employees earning more than $60,000 and from managers (including duplicates).**

sql

Copy code

SELECT Department

FROM EMP

WHERE Salary > 60000

UNION ALL

SELECT Department

FROM EMP

WHERE ManagerID IS NOT NULL;

**3. INTERSECT**

**Retrieve common rows from two queries.**

**Example: Find departments with employees earning more than $60,000 and also having a manager.**

sql

Copy code

SELECT Department

FROM EMP

WHERE Salary > 60000

INTERSECT

SELECT Department

FROM EMP

WHERE ManagerID IS NOT NULL;

**4. EXCEPT**

**Retrieve rows from the first query that do not appear in the second query.**

**Example: Find departments with employees earning more than $60,000 but no managers.**

**s**ql

Copy code

SELECT Department

FROM EMP

WHERE Salary > 60000

EXCEPT

SELECT Department

FROM EMP

WHERE ManagerID IS NOT NULL;

**JOINS**

**Key Notes:**

* **Column Alignment:** The number of columns and their data types in both queries must match when using set operators.
* **UNION vs UNION ALL:** Use UNION when you want unique results and UNION ALL to include duplicates.
* **Performance:** UNION ALL is generally faster than UNION because it skips duplicate elimination.

To demonstrate **all types of joins** and **multiple table joins**, let’s consider the following database schema:

### Example Tables

1. **EMP** (Employees Table)

CREATE TABLE EMP (

EmpID INT PRIMARY KEY,

FirstName NVARCHAR(50),

LastName NVARCHAR(50),

DepartmentID INT,

ManagerID INT,

Salary DECIMAL(10, 2)

);

1. **DEPARTMENT** (Departments Table)

CREATE TABLE DEPARTMENT (

DepartmentID INT PRIMARY KEY,

DepartmentName NVARCHAR(50)

);

1. **PROJECT** (Projects Table)

CREATE TABLE PROJECT (

ProjectID INT PRIMARY KEY,

ProjectName NVARCHAR(50),

DepartmentID INT

);

### Queries Demonstrating All Joins

#### 1. ****INNER JOIN****

Retrieve employees and their departments (only matching records):

SELECT

E.EmpID,

E.FirstName,

E.LastName,

D.DepartmentName

FROM EMP E

INNER JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

#### 2. ****LEFT JOIN****

Retrieve all employees and their departments (including employees without a department):

SELECT

E.EmpID,

E.FirstName,

E.LastName,

D.DepartmentName

FROM EMP E

LEFT JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

#### 3. ****RIGHT JOIN****

Retrieve all departments and their employees (including departments without employees):

SELECT

E.EmpID,

E.FirstName,

E.LastName,

D.DepartmentName

FROM EMP E

RIGHT JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

#### 4. ****FULL OUTER JOIN****

Retrieve all employees and all departments, showing matches where possible:

SELECT

E.EmpID,

E.FirstName,

E.LastName,

D.DepartmentName

FROM EMP E

FULL OUTER JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

#### 5. ****CROSS JOIN****

Pair every employee with every department (cartesian product):

sql

Copy code

SELECT

E.EmpID,

E.FirstName,

E.LastName,

D.DepartmentName

FROM EMP E

CROSS JOIN DEPARTMENT D;

#### 6. ****Multiple Joins****

Retrieve employees, their departments, and the projects their departments are working on:

sql

Copy code

SELECT

E.EmpID,

E.FirstName,

E.LastName,

D.DepartmentName,

P.ProjectName

FROM EMP E

INNER JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID

INNER JOIN PROJECT P

ON D.DepartmentID = P.DepartmentID;

#### 7. ****Self Join****

Retrieve employees and their managers:

sql

Copy code

SELECT

E.EmpID AS EmployeeID,

E.FirstName AS EmployeeName,

M.EmpID AS ManagerID,

M.FirstName AS ManagerName

FROM EMP E

LEFT JOIN EMP M

ON E.ManagerID = M.EmpID;

### Tips:

* Use **ON** to specify join conditions.
* For large datasets, prefer **INNER JOIN** where applicable for performance.
* Combine **JOINs** with **WHERE** or **GROUP BY** for filtering and aggregation.
* Here are examples of different types of **subqueries** that can be applied to the EMP table. Subqueries can be used in various parts of SQL queries, including SELECT, FROM, and WHERE clauses.

**SUB QUERIES**

* **1. Subquery in SELECT Clause**
* **Retrieve each employee’s salary and show the average salary of all employees as a column.**
* SELECT
* EmpID,
* FirstName,
* LastName,
* Salary,
* (SELECT AVG(Salary) FROM EMP) AS AvgSalary
* FROM EMP;
* **2. Subquery in WHERE Clause**
* **Find employees earning above the average salary.**
* SELECT EmpID, FirstName, LastName, Salary
* FROM EMP
* WHERE Salary > (SELECT AVG(Salary) FROM EMP);
* **3. Subquery in FROM Clause**
* **Use a subquery to calculate the total salary of each department, then join it with the DEPARTMENT table.**
* SELECT
* D.DepartmentName,
* T.TotalSalary
* FROM DEPARTMENT D
* INNER JOIN (
* SELECT DepartmentID, SUM(Salary) AS TotalSalary
* FROM EMP
* GROUP BY DepartmentID
* ) T
* ON D.DepartmentID = T.DepartmentID;
* **4. Correlated Subquery**
* **Find employees whose salaries are higher than the average salary of their respective department.**
* SELECT EmpID, FirstName, LastName, DepartmentID, Salary
* FROM EMP E1
* WHERE Salary > (
* SELECT AVG(Salary)
* FROM EMP E2
* WHERE E2.DepartmentID = E1.DepartmentID
* );
* **5. Subquery with EXISTS**
* **Find departments with at least one employee.**
* SELECT DepartmentName
* FROM DEPARTMENT D
* WHERE EXISTS (
* SELECT 1
* FROM EMP E
* WHERE E.DepartmentID = D.DepartmentID
* );
* **6. Subquery with IN**
* **Find employees who work in a department managed by a specific manager.**
* SELECT EmpID, FirstName, LastName, DepartmentID
* FROM EMP
* WHERE DepartmentID IN (
* SELECT DepartmentID
* FROM EMP
* WHERE ManagerID = 1
* );
* 7. Subquery with NOT IN
* Find employees who are not managers.
* SELECT EmpID, FirstName, LastName
* FROM EMP
* WHERE EmpID NOT IN (
* SELECT DISTINCT ManagerID
* FROM EMP
* WHERE ManagerID IS NOT NULL
* );
* **8. Subquery with ALL**
* **Find employees whose salaries are greater than all employees in department 2.**
* SELECT EmpID, FirstName, LastName, Salary
* FROM EMP
* WHERE Salary > ALL (
* SELECT Salary
* FROM EMP
* WHERE DepartmentID = 2
* );
* **9. Subquery with ANY**
* **Find employees whose salaries are greater than any employee in department 2.**
* SELECT EmpID, FirstName, LastName, Salary
* FROM EMP
* WHERE Salary > ANY (
* SELECT Salary
* FROM EMP
* WHERE DepartmentID = 2
* );
* **10. Subquery in INSERT Statement**
* **Insert a new employee in the department with the highest total salary.**
* INSERT INTO EMP (FirstName, LastName, DepartmentID, Salary)
* VALUES (
* 'New', 'Employee',
* (SELECT TOP 1 DepartmentID
* FROM EMP
* GROUP BY DepartmentID
* ORDER BY SUM(Salary) DESC),
* 80000
* );
* **11. Subquery in UPDATE Statement**
* **Update the salary of employees who earn less than the average salary.**
* UPDATE EMP
* SET Salary = Salary + 5000
* WHERE Salary < (SELECT AVG(Salary) FROM EMP);
* **12. Subquery in DELETE Statement**
* **Delete employees in departments with no employees assigned.**
* DELETE FROM EMP
* WHERE DepartmentID IN (
* SELECT DepartmentID
* FROM DEPARTMENT
* WHERE DepartmentID NOT IN (SELECT DISTINCT DepartmentID FROM EMP)
* );

**Stored Procedure**

CREATE PROCEDURE ManageEmployee

@Action NVARCHAR(10),

@EmpID INT = NULL,

@FirstName NVARCHAR(50) = NULL,

@LastName NVARCHAR(50) = NULL,

@DepartmentID INT = NULL,

@ManagerID INT = NULL,

@Salary DECIMAL(10, 2) = NULL

AS

BEGIN

SET NOCOUNT ON;

IF @Action = 'INSERT'

BEGIN

INSERT INTO EMP (FirstName, LastName, DepartmentID, ManagerID, Salary)

VALUES (@FirstName, @LastName, @DepartmentID, @ManagerID, @Salary);

END

ELSE IF @Action = 'UPDATE'

BEGIN

UPDATE EMP

SET

FirstName = ISNULL(@FirstName, FirstName),

LastName = ISNULL(@LastName, LastName),

DepartmentID = ISNULL(@DepartmentID, DepartmentID),

ManagerID = ISNULL(@ManagerID, ManagerID),

Salary = ISNULL(@Salary, Salary)

WHERE EmpID = @EmpID;

END

ELSE IF @Action = 'DELETE'

BEGIN

DELETE FROM EMP

WHERE EmpID = @EmpID;

END

ELSE IF @Action = 'SELECT'

BEGIN

SELECT \*

FROM EMP

WHERE

(EmpID = @EmpID OR @EmpID IS NULL) AND

(FirstName = @FirstName OR @FirstName IS NULL) AND

(LastName = @LastName OR @LastName IS NULL) AND

(DepartmentID = @DepartmentID OR @DepartmentID IS NULL) AND

(ManagerID = @ManagerID OR @ManagerID IS NULL);

END

END;

**Without Parameters Create Stored Procedured:**

CREATE PROCEDURE EmployeeDetails

BEGIN

AS

SELECT \* FROM

END;

**INDEXERS**

### 1. ****Clustered Index****

A **clustered index** determines the physical order of data in the table. Each table can have only one clustered index.

#### Create Clustered Index on EmpID

Since EmpID is the primary key, it's common to create a clustered index on this column.

CREATE CLUSTERED INDEX IDX\_EmpID

ON EMP(EmpID);

### 2. ****Non-Clustered Index****

A **non-clustered index** creates a separate structure for the index while leaving the data in the table unsorted.

#### Non-Clustered Index on DepartmentID

To optimize queries filtering by DepartmentID, create a non-clustered index:

CREATE NONCLUSTERED INDEX IDX\_DepartmentID

ON EMP(DepartmentID);

#### Non-Clustered Index on Salary

If queries frequently sort or filter by Salary, create a non-clustered index:

CREATE NONCLUSTERED INDEX IDX\_Salary

ON EMP(Salary);

### 3. ****Composite Non-Clustered Index****

A composite index involves multiple columns and is useful for optimizing queries with multiple conditions.

#### Create Composite Index on DepartmentID and Salary

To optimize queries filtering or sorting by both DepartmentID and Salary:

CREATE NONCLUSTERED INDEX IDX\_Department\_Salary

ON EMP(DepartmentID, Salary);

### 4. ****Unique Non-Clustered Index****

A unique index ensures that the indexed column(s) contain only unique values.

#### Unique Index on LastName and FirstName

To ensure that no two employees have the same first and last name:

CREATE UNIQUE NONCLUSTERED INDEX IDX\_Unique\_FullName

ON EMP(LastName, FirstName);

### 5. ****Filtered Non-Clustered Index****

Filtered indexes optimize queries that target specific subsets of data.

#### Filtered Index for High-Salary Employees

If queries often filter employees with salaries greater than 50,000:

CREATE NONCLUSTERED INDEX IDX\_Filtered\_HighSalary

ON EMP(Salary)

WHERE Salary > 50000;

### 6. ****Drop an Index****

To remove an existing index:

DROP INDEX IDX\_DepartmentID ON EMP;

### Query Performance Scenarios:

1. **Clustered Index**:
   * Used automatically when searching by EmpID.
   * Example:

SELECT \* FROM EMP WHERE EmpID = 101;

1. **Non-Clustered Index**:
   * Index on DepartmentID will speed up:

SELECT \* FROM EMP WHERE DepartmentID = 2;

1. **Composite Index**:
   * Optimizes queries like:

SELECT \* FROM EMP WHERE DepartmentID = 2 AND Salary > 60000;

1. **Filtered Index**:
   * Optimizes specific range queries like:

SELECT \* FROM EMP WHERE Salary > 50000;

### Key Notes:

* **Clustered Index**: Best for primary keys or columns frequently queried for ranges.
* **Non-Clustered Index**: Best for columns often used in WHERE, ORDER BY, or JOIN conditions.
* **Composite Index**: Best for multi-condition queries.
* **Filtered Index**: Reduces index size and improves performance for specific subsets.

Here are examples of **SQL clauses** that can be used with the EMP table for various operations, such as filtering, sorting, grouping, limiting results, and more.

**1. WHERE Clause**

Filters records based on a condition.

**Example: Find employees with a salary greater than 50,000.**

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

WHERE Salary > 50000;

**2. ORDER BY Clause**

Sorts the result set by one or more columns.

**Example: Sort employees by salary in descending order.**

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

ORDER BY Salary DESC;

**3. GROUP BY Clause**

Groups rows that have the same values into summary rows, often used with aggregate functions.

**Example: Get the average salary per department.**

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM EMP

GROUP BY DepartmentID;

**4. HAVING Clause**

Filters groups created by the GROUP BY clause. It is used to apply a condition on aggregated data.

**Example: Get departments with an average salary greater than 60,000.**

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM EMP

GROUP BY DepartmentID

HAVING AVG(Salary) > 60000;

**5. JOIN Clause**

Used to combine rows from two or more tables based on a related column.

**Example: Get employees along with their department names (using INNER JOIN).**

SELECT E.EmpID, E.FirstName, E.LastName, D.DepartmentName

FROM EMP E

INNER JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

**6. LEFT JOIN Clause**

Returns all records from the left table (EMP), and the matched records from the right table (DEPARTMENT). If no match, NULL values are returned for columns from the right table.

**Example: Get all employees and their department names (including employees with no department).**

SELECT E.EmpID, E.FirstName, E.LastName, D.DepartmentName

FROM EMP E

LEFT JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

**7. RIGHT JOIN Clause**

Returns all records from the right table (DEPARTMENT), and the matched records from the left table (EMP). If no match, NULL values are returned for columns from the left table.

**Example: Get all departments and their employees (including departments with no employees).**

SELECT E.EmpID, E.FirstName, E.LastName, D.DepartmentName

FROM EMP E

RIGHT JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

**8. FULL OUTER JOIN Clause**

Returns all records when there is a match in either the left (EMP) or right (DEPARTMENT) table. It returns NULL for non-matching rows in both tables.

**Example: Get all employees and all departments (including unmatched rows).**

SELECT E.EmpID, E.FirstName, E.LastName, D.DepartmentName

FROM EMP E

FULL OUTER JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

**9. DISTINCT Clause**

Removes duplicate rows from the result set.

**Example: Get distinct department IDs.**

SELECT DISTINCT DepartmentID

FROM EMP;

**10. LIMIT Clause (SQL Server uses TOP)**

Limits the number of rows returned.

**Example: Get the top 5 highest-paid employees.**

SELECT TOP 5 EmpID, FirstName, LastName, Salary

FROM EMP

ORDER BY Salary DESC;

**11. IN Clause**

Checks if a value is present in a list of values.

**Example: Find employees in departments 1, 2, and 3.**

SELECT EmpID, FirstName, LastName, DepartmentID

FROM EMP

WHERE DepartmentID IN (1, 2, 3);

**12. BETWEEN Clause**

Selects values within a range (inclusive).

**Example: Find employees with a salary between 40,000 and 60,000.**

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

WHERE Salary BETWEEN 40000 AND 60000;

**13. LIKE Clause**

Searches for a specified pattern in a column.

**Example: Find employees whose first name starts with "J".**

SELECT EmpID, FirstName, LastName

FROM EMP

WHERE FirstName LIKE 'J%';

**14. IS NULL Clause**

Checks if a value is NULL.

**Example: Find employees who have no manager.**

SELECT EmpID, FirstName, LastName

FROM EMP

WHERE ManagerID IS NULL;

**15. IS NOT NULL Clause**

Checks if a value is not NULL.

**Example: Find employees who have a manager.**

SELECT EmpID, FirstName, LastName

FROM EMP

WHERE ManagerID IS NOT NULL;

**16. EXISTS Clause**

Checks if a subquery returns any records.

**Example: Find departments that have employees (using EXISTS).**

SELECT DepartmentName

FROM DEPARTMENT D

WHERE EXISTS (

SELECT 1

FROM EMP E

WHERE E.DepartmentID = D.DepartmentID

);

**17. ANY Clause**

Compares a value to any value in another result set or subquery.

**Example: Find employees with a salary greater than any employee in department 2.**

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

WHERE Salary > ANY (

SELECT Salary

FROM EMP

WHERE DepartmentID = 2

);

**18. ALL Clause**

Compares a value to all values in another result set or subquery.

**Example: Find employees whose salary is greater than all employees in department 2.**

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

WHERE Salary > ALL (

SELECT Salary

FROM EMP

WHERE DepartmentID = 2

);

**19. UNION Clause**

Combines the result sets of two or more SELECT statements (removes duplicates).

**Example: Get a list of all unique departments from two different queries.**

SELECT DepartmentID FROM EMP

UNION

SELECT DepartmentID FROM DEPARTMENT;

**20. UNION ALL Clause**

Combines the result sets of two or more SELECT statements (does not remove duplicates).

**Example: Get a list of all departments, including duplicates.**

SELECT DepartmentID FROM EMP

UNION ALL

SELECT DepartmentID FROM DEPARTMENT;

**Conclusion:**

* Use **WHERE**, **ORDER BY**, and **GROUP BY** for filtering and sorting.
* **HAVING** is used for filtering after aggregation.
* **JOIN** clauses allow combining related data from multiple tables.
* **DISTINCT**, **IN**, **LIKE**, **BETWEEN**, and other operators help refine data extraction based on specific conditions.
* **EXISTS**, **ANY**, and **ALL** allow more complex comparisons with subqueries.

A **view** in SQL is essentially a virtual table that stores a query for easy reuse. It can be used to simplify complex queries, restrict access to certain columns or rows, or join multiple tables. Below are some examples of **views** based on the EMP table.

**1. Simple View: Employee Details**

Create a view to retrieve basic employee details (e.g., employee ID, first name, last name, and salary).

CREATE VIEW EmployeeDetails AS

SELECT EmpID, FirstName, LastName, Salary

FROM EMP;

**Usage**:  
You can now query the view like a table to get employee details.

SELECT \* FROM EmployeeDetails;

**2. View with Filtered Data: Employees with Salary Above 50,000**

Create a view to retrieve only employees with salaries greater than 50,000.

CREATE VIEW HighSalaryEmployees AS

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

WHERE Salary > 50000;

**Usage**: Query the view to get employees with high salaries.

SELECT \* FROM HighSalaryEmployees;

**3. View with Aggregation: Average Salary by Department**

Create a view to show the average salary for each department.

CREATE VIEW AvgSalaryByDepartment AS

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM EMP

GROUP BY DepartmentID;

**Usage**: You can query the view to see the average salary by department.

SELECT \* FROM AvgSalaryByDepartment;

**4. View with Join: Employees and Their Departments**

Create a view that joins the EMP table with the DEPARTMENT table to show employees along with their department names.

CREATE VIEW EmployeeDepartmentDetails AS

SELECT E.EmpID, E.FirstName, E.LastName, D.DepartmentName

FROM EMP E

INNER JOIN DEPARTMENT D

ON E.DepartmentID = D.DepartmentID;

**Usage**: Query this view to get employees along with their respective department names.

SELECT \* FROM EmployeeDepartmentDetails;

**5. View with Conditional Data: Employees Managed by a Specific Manager**

Create a view to show employees who are managed by a specific manager (e.g., ManagerID = 1).

CREATE VIEW EmployeesByManager AS

SELECT EmpID, FirstName, LastName, ManagerID

FROM EMP

WHERE ManagerID = 1;

**Usage**: You can query this view to get the employees managed by manager with ManagerID = 1.

SELECT \* FROM EmployeesByManager;

**6. View for Employees with Full Name**

Create a view that combines the first and last names of employees into a single column for easier display.

CREATE VIEW EmployeeFullNames AS

SELECT EmpID,

FirstName + ' ' + LastName AS FullName

FROM EMP;

**Usage**: Query the view to get employees' full names.

SELECT \* FROM EmployeeFullNames;

**7. View with Multiple Filters: Employees in a Specific Department with High Salary**

Create a view to show employees who are in department 2 and have a salary greater than 60,000.

CREATE VIEW HighEarnersInDept2 AS

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

WHERE DepartmentID = 2 AND Salary > 60000;

**Usage**: Query the view to get high-earning employees in department 2.

SELECT \* FROM HighEarnersInDept2;

**8. View with Nested Queries: Employees with Salaries Above Average**

Create a view to show employees whose salary is above the average salary for all employees.

CREATE VIEW EmployeesAboveAverageSalary AS

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

WHERE Salary > (SELECT AVG(Salary) FROM EMP);

**Usage**: You can query this view to see employees who earn more than the average salary.

SELECT \* FROM EmployeesAboveAverageSalary;

**9. View with Calculated Columns: Salary and Bonus**

Create a view to show employee salaries and calculate their bonus (e.g., 10% of salary).

CREATE VIEW EmployeeSalaryWithBonus AS

SELECT EmpID, FirstName, LastName, Salary, (Salary \* 0.10) AS Bonus

FROM EMP;

**Usage**: Query the view to see employees’ salary and their bonus.

SELECT \* FROM EmployeeSalaryWithBonus;

**10. Dropping a View**

If you need to drop an existing view, use the following command:

DROP VIEW EmployeeDetails;

**Notes:**

1. **Views and Performance**: While views make complex queries easier to manage and use, be mindful that they can sometimes impact performance, especially if they involve complex joins or aggregations.
2. **Updatable Views**: In some cases, you can update a view directly (if it meets certain criteria), but views involving joins or aggregations are typically read-only.
3. **Permissions**: You can use views to restrict access to certain columns or rows of data in a table.

**Difference between Common Table Expressions (CTEs) and Temporary Tables**

Here’s a quick comparison between **Common Table Expressions (CTEs)** and **Temporary Tables** in SQL:

| **Feature** | **Common Table Expression (CTE)** | **Temporary Table** |
| --- | --- | --- |
| **Definition** | A CTE is a temporary result set defined within the execution scope of a SELECT, INSERT, UPDATE, or DELETE query. | A temporary table is a table that exists only during the session in which it is created, and its data is stored in the tempdb system database. |
| **Scope** | CTEs exist only within the execution of a single query or statement. | Temporary tables exist for the duration of a session or until explicitly dropped. |
| **Persistence** | A CTE is not stored; it is just a temporary result used within the query. | A temporary table exists independently of a query and can be queried multiple times within a session. |
| **Usage** | Used to simplify complex queries, especially with recursive queries. | Useful when needing to store intermediate results or work with large datasets across multiple queries. |
| **Performance** | Generally faster for small datasets and simple queries. | Temporary tables might be slower for large datasets but can be indexed and are more persistent during the session. |
| **Creation Syntax** | Defined using the WITH clause. | Defined using CREATE TABLE with a # (local) or ## (global) prefix. |
| **Indexes** | Cannot have indexes. | Can have indexes, constraints, and even primary keys. |
| **Recursive Capability** | Can be recursive (with RECURSIVE keyword in CTE). | Cannot be recursive. |
| **Session Scope** | Limited to a single query or a part of a query. | Can be accessed across multiple queries within a session. |

**1. Common Table Expression (CTE) Example with EMP Table**

A **CTE** is defined using the WITH keyword. Below is an example of using a CTE to find the average salary by department, and then use that CTE in a query to find employees with salaries above the average.

**Example: CTE to Find Employees with Salary Above Average by Department**

WITH AvgSalaryByDepartment AS (

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM EMP

GROUP BY DepartmentID

)

SELECT E.EmpID, E.FirstName, E.LastName, E.Salary, E.DepartmentID

FROM EMP E

JOIN AvgSalaryByDepartment AS A

ON E.DepartmentID = A.DepartmentID

WHERE E.Salary > A.AvgSalary;

* In this query:
  + The AvgSalaryByDepartment CTE calculates the average salary per department.
  + The main query joins this CTE with the EMP table to find employees whose salary is above the department's average.

**Note**: A CTE is temporary and can be used only within the execution of the query it is defined in.

**2. Temporary Table Example with EMP Table**

A **temporary table** is created using the CREATE TABLE statement with the # prefix (local temporary table). It can be used across multiple queries within the same session.

**Example: Create Temporary Table to Store High-Salary Employees**

-- Create a temporary table

CREATE TABLE #HighSalaryEmployees (

EmpID INT,

FirstName NVARCHAR(50),

LastName NVARCHAR(50),

Salary DECIMAL(18, 2)

);

-- Insert data into the temporary table

INSERT INTO #HighSalaryEmployees (EmpID, FirstName, LastName, Salary)

SELECT EmpID, FirstName, LastName, Salary

FROM EMP

WHERE Salary > 50000;

-- Query the temporary table

SELECT \* FROM #HighSalaryEmployees;

-- Drop the temporary table (optional, will be dropped automatically when the session ends)

DROP TABLE #HighSalaryEmployees;

* In this query:
  + A temporary table #HighSalaryEmployees is created to store employees with a salary greater than 50,000.
  + The INSERT INTO statement populates the temporary table.
  + You can query the temporary table as you would with a regular table.

**Note**: Temporary tables are stored in the tempdb database, and they are automatically dropped when the session ends, or you can explicitly drop them using DROP TABLE.

**Key Differences in Usage:**

1. **CTEs**:
   * Use CTEs when you need a temporary result set that only exists within a specific query.
   * Ideal for recursive queries or breaking down complex queries into simpler parts.
   * Cannot persist data between queries or across multiple sessions.
2. **Temporary Tables**:
   * Use temporary tables when you need to store intermediate results over multiple queries or need to perform operations like indexing or adding constraints.
   * Can persist across multiple queries in the same session.
   * Useful for working with large data sets or for complex manipulations across queries.

**3. Recursive CTE Example with EMP Table**

CTEs also support recursion. Below is an example using a recursive CTE to find all employees who report to a specific manager in the EMP table.

**Example: Recursive CTE to Find Employees Reporting to a Manager**

WITH EmployeeHierarchy AS (

-- Base case: Get employees directly reporting to ManagerID = 1

SELECT EmpID, FirstName, LastName, ManagerID

FROM EMP

WHERE ManagerID = 1

UNION ALL

-- Recursive case: Get employees reporting to employees found in previous step

SELECT E.EmpID, E.FirstName, E.LastName, E.ManagerID

FROM EMP E

INNER JOIN EmployeeHierarchy EH

ON E.ManagerID = EH.EmpID

)

SELECT EmpID, FirstName, LastName

FROM EmployeeHierarchy;

* In this query:
  + The CTE EmployeeHierarchy recursively selects employees who report to a manager with ManagerID = 1.
  + The UNION ALL clause combines the base case (direct reports) with the recursive case (employees reporting to previous employees).
  + The result is a list of employees who report to manager 1, including indirect reports.

**Conclusion:**

* **CTEs** are more useful for simplifying complex queries and for cases where the result set is needed only within a single query. CTEs are defined with the WITH keyword and are temporary.
* **Temporary Tables** are useful when you need to store data that persists across multiple queries within the same session. They can also be indexed, have constraints, and be queried multiple times.