**CREATE TABLE Department (**

**DeptID INT PRIMARY KEY,**

**DeptName VARCHAR(50),**

**Location VARCHAR(50)**

**);**

**-- Employees Table**

**CREATE TABLE Employees (**

**EmpID INT PRIMARY KEY,**

**EmpName VARCHAR(50) NOT NULL,**

**Gender CHAR(1),**

**Salary DECIMAL(10,2),**

**DeptID INT,**

**EmailID VARCHAR(50) UNIQUE,**

**FOREIGN KEY (DeptID) REFERENCES Department(DeptID)**

**);**

**-- Insert Department Data**

**INSERT INTO Department VALUES**

**(1, 'HR', 'Delhi'),**

**(2, 'IT', 'Delhi'),**

**(3, 'Finance', 'Mumbai'),**

**(4, 'Marketing', 'Bangalore'),**

**(5, 'Operations', 'Chennai');**

**-- Insert Employees Data**

**INSERT INTO Employees VALUES**

**(101, 'Amit', 'M', 50000, 1, 'amit@abc.com'),**

**(102, 'Meena', 'F', 60000, 2, 'meena@abc.com'),**

**(103, 'John', 'M', 55000, 2, 'john@abc.com'),**

**(104, 'Rita', 'F', 45000, 3, 'rita@abc.com'),**

**(105, 'Suresh', 'M', 70000, 2, 'suresh@abc.com'),**

**(106, 'Neha', 'F', 40000, 4, 'neha@abc.com'),**

**(107, 'Vikram', 'M', 65000, 5, 'vikram@abc.com'),**

**(108, 'Pooja', 'F', 30000, NULL, 'pooja@abc.com'); -- employee without department**

**Now we have:**

**5 departments**

**8 employees (1 has no department, useful for OUTER JOIN demo)**

**4.6 JOINS**

**1) INNER JOIN (only matching rows)**

**SELECT EmpName, DeptName, Salary**

**FROM Employees E**

**INNER JOIN Department D**

**ON E.DeptID = D.DeptID;**

**Output → Only employees with a department will appear (Pooja excluded).**

**Example: Meena | IT | 60000**

**2) LEFT OUTER JOIN (all employees, even without dept)**

**SELECT EmpName, DeptName, Salary**

**FROM Employees E**

**LEFT JOIN Department D**

**ON E.DeptID = D.DeptID;**

**Output → Pooja also appears but DeptName = NULL.**

**3) RIGHT OUTER JOIN (all departments, even without employees)**

**SELECT EmpName, DeptName**

**FROM Employees E**

**RIGHT JOIN Department D**

**ON E.DeptID = D.DeptID;**

**Output → If a department has no employee, it will still show up with NULL employee.**

**4) FULL OUTER JOIN (all employees + all departments)**

**SELECT EmpName, DeptName**

**FROM Employees E**

**FULL OUTER JOIN Department D**

**ON E.DeptID = D.DeptID;**

**Output → Combines LEFT + RIGHT join results.**

**5) CROSS JOIN (Cartesian product)**

**SELECT EmpName, DeptName**

**FROM Employees**

**CROSS JOIN Department;**

**Output → Every employee paired with every department (8×5 = 40 rows).**

**6) SELF JOIN (employee comparison)**

**SELECT E1.EmpName AS Employee, E2.EmpName AS Colleague, E1.DeptID**

**FROM Employees E1**

**JOIN Employees E2**

**ON E1.DeptID = E2.DeptID AND E1.EmpID <> E2.EmpID;**

**Output → Lists colleagues in the same department (Meena ↔ John, Meena ↔ Suresh, etc).**

**7) LATERAL JOIN (fetching top N per dept)**

**Using CROSS APPLY (SQL Server equivalent of LATERAL):**

**SELECT D.DeptName, E.EmpName, E.Salary**

**FROM Department D**

**CROSS APPLY (**

**SELECT TOP 1 EmpName, Salary**

**FROM Employees**

**WHERE Employees.DeptID = D.DeptID**

**ORDER BY Salary DESC**

**) E;**

**Explanation:**

* **CROSS APPLY in SQL Server = LATERAL JOIN in PostgreSQL.**
* **TOP 1 replaces LIMIT 1 to fetch only the highest-paid employee per department.**
* **The ORDER BY Salary DESC ensures we get the maximum salary.**

**Output: Highest-paid employee per department (same as in PostgreSQL).**

**SQL Server – using OFFSET … FETCH NEXT**

**SELECT D.DeptName, E.EmpName, E.Salary**

**FROM Department D**

**CROSS APPLY (**

**SELECT EmpName, Salary**

**FROM Employees**

**WHERE Employees.DeptID = D.DeptID**

**ORDER BY Salary DESC**

**OFFSET 0 ROWS FETCH NEXT 1 ROWS ONLY**

**) E;**

**Explanation:**

* **OFFSET 0 ROWS → skip 0 rows (start from the top).**
* **FETCH NEXT 1 ROWS ONLY → return just the top 1 row.**
* **Works exactly like LIMIT 1 in PostgreSQL.**

**Both this query and the earlier TOP 1 version give the same result: highest-paid employee per department.**

**8) COALESCE (replace NULL values)**

**SELECT EmpName, COALESCE(DeptName, 'No Department') AS Department**

**FROM Employees E**

**LEFT JOIN Department D**

**ON E.DeptID = D.DeptID;**

**Output → Pooja will show "No Department".**

**9) CASE (conditional output)**

**SELECT EmpName, Salary,**

**CASE**

**WHEN Salary > 60000 THEN 'High Earner'**

**WHEN Salary BETWEEN 40000 AND 60000 THEN 'Mid Earner'**

**ELSE 'Low Earner'**

**END AS Category**

**FROM Employees;**

**Output → Employees categorized into High, Mid, Low earners.**

**10) CONCAT (string joining)**

**SELECT CONCAT(EmpName, ' works in ', COALESCE(DeptName, 'No Dept')) AS Info**

**FROM Employees E**

**LEFT JOIN Department D**

**ON E.DeptID = D.DeptID;**

**Output → "Amit works in HR", "Pooja works in No Dept".**

**11) Recursive CTE (hierarchy)**

**WITH RECURSIVE Numbers AS (**

**SELECT 1 AS n**

**UNION ALL**

**SELECT n+1 FROM Numbers WHERE n < 5**

**)**

**SELECT \* FROM Numbers;**

**Output → Generates numbers 1 to 5 (shows recursive queries).**

**To create hierarchy, we need to know who reports to whom.  
So, let’s add a column ManagerID (where null = top-level manager).**

**-- Alter Employees table to add ManagerID**

**ALTER TABLE Employees ADD ManagerID INT NULL;**

**-- Update data to create a hierarchy**

**UPDATE Employees SET ManagerID = NULL WHERE EmpID = 101; -- Amit = HR Head**

**UPDATE Employees SET ManagerID = 101 WHERE EmpID = 104; -- Rita reports to Amit**

**UPDATE Employees SET ManagerID = NULL WHERE EmpID = 102; -- Meena = IT Head**

**UPDATE Employees SET ManagerID = 102 WHERE EmpID = 103; -- John reports to Meena**

**-- Add a few more employees**

**INSERT INTO Employees (EmpID, EmpName, Gender, Salary, DeptID, EmailID, ManagerID) VALUES**

**(105, 'Raj', 'M', 40000, 2, 'raj@abc.com', 103), -- Raj reports to John**

**(106, 'Priya', 'F', 35000, 3, 'priya@abc.com', 104); -- Priya reports to Rita**

**Step 2: Recursive CTE query**

**Now let’s build a recursive CTE to show employee hierarchy.**

**WITH RECURSIVE EmployeeHierarchy AS (**

**-- Anchor member: Select top-level employees (no manager)**

**SELECT**

**EmpID, EmpName, ManagerID, DeptID, 1 AS Level**

**FROM Employees**

**WHERE ManagerID IS NULL**

**UNION ALL**

**-- Recursive member: Find employees reporting to previous level**

**SELECT**

**e.EmpID, e.EmpName, e.ManagerID, e.DeptID, eh.Level + 1**

**FROM Employees e**

**INNER JOIN EmployeeHierarchy eh**

**ON e.ManagerID = eh.EmpID**

**)**

**SELECT**

**eh.Level, e.EmpName AS Employee, m.EmpName AS Manager, d.DeptName**

**FROM EmployeeHierarchy eh**

**LEFT JOIN Employees e ON eh.EmpID = e.EmpID**

**LEFT JOIN Employees m ON eh.ManagerID = m.EmpID**

**LEFT JOIN Department d ON eh.DeptID = d.DeptID**

**ORDER BY eh.Level, Employee;**

**Step 3: Explanation of Output**

**Expected Output (sample):**

| **Level** | **Employee** | **Manager** | **Department** |
| --- | --- | --- | --- |
| **1** | **Amit** | **NULL** | **HR** |
| **2** | **Rita** | **Amit** | **Finance** |
| **3** | **Priya** | **Rita** | **Finance** |
| **1** | **Meena** | **NULL** | **IT** |
| **2** | **John** | **Meena** | **IT** |
| **3** | **Raj** | **John** | **IT** |

**What happened here?**

* **Level 1 = Department heads (Amit & Meena).**
* **Level 2 = Direct reports (Rita reports to Amit, John reports to Meena).**
* **Level 3 = Deeper hierarchy (Priya under Rita, Raj under John).**

**Recursive CTE is super useful for hierarchies like org charts, category trees, folder structures, bill of materials.**

**4.7 SET OPERATORS**

**1) UNION (combine unique)**

**SELECT DeptID FROM Employees**

**UNION**

**SELECT DeptID FROM Department;**

**Output → List of all DeptIDs without duplicates.**

**2) UNION ALL (combine all, with duplicates)**

**SELECT DeptID FROM Employees**

**UNION ALL**

**SELECT DeptID FROM Department;**

**Output → Same as above but duplicates included.**

**3) INTERSECT (common values)**

**SELECT DeptID FROM Employees**

**INTERSECT**

**SELECT DeptID FROM Department;**

**Output → DeptIDs that exist in both tables.**

**4) EXCEPT (difference)**

**SELECT DeptID FROM Department**

**EXCEPT**

**SELECT DeptID FROM Employees;**

**Output → Departments that have no employees.**

**4.8 Temporary Tables**

**CREATE TEMP TABLE TempEmp AS**

**SELECT EmpName, Salary FROM Employees WHERE Salary > 50000;**

**Creates a temporary subset of employees with salary > 50k.**

**4.10 Table Variables (SQL Server / PL/pgSQL)**

**DECLARE @TempEmp TABLE (**

**EmpName VARCHAR(50),**

**Salary DECIMAL(10,2)**

**);**

**INSERT INTO @TempEmp**

**SELECT EmpName, Salary FROM Employees WHERE Salary > 50000;**

**SELECT \* FROM @TempEmp;**

**Works like a temporary dataset stored in memory.**

**5.1 Understanding SQL functions – what do they do?**

**SQL functions are predefined routines that take input values (arguments), perform operations (calculation, formatting, transformation), and return a value.  
They save time, reduce repetitive code, and help perform complex operations on data quickly.**

**Example: UPPER(EmpName) → converts employee names to uppercase.**

**5.2 Scalar functions**

**Operate on a single value and return a single value.**

**Example – Convert names to uppercase and get length of email ID:**

**SELECT EmpName,**

**UPPER(EmpName) AS UpperName,**

**LEN(EmailID) AS EmailLength**

**FROM Employees;**

**Output explanation:**

* **Amit → AMIT, Email amit@abc.com has length 12.**
* **Each row is transformed individually.**

**5.3 Aggregate functions**

**Operate on a set of rows and return a single summarized value.**

**Example – Find average salary per department:**

**SELECT d.DeptName,**

**AVG(e.Salary) AS AvgSalary,**

**COUNT(e.EmpID) AS TotalEmployees**

**FROM Employees e**

**JOIN Department d ON e.DeptID = d.DeptID**

**GROUP BY d.DeptName;**

**Output explanation:**

* **HR → AvgSalary: 50000, Employees: 1**
* **IT → AvgSalary: (60000+55000)/2 = 57500, Employees: 2**
* **Finance → AvgSalary: 45000, Employees: 1**

**Aggregate functions collapse rows into groups.**

**5.4 Functions on Numbers, Strings, and Dates**

**Example – Mix of numeric, string, and date functions:**

**SELECT EmpName,**

**ROUND(Salary \* 1.1, 0) AS BonusSalary, -- numeric**

**LOWER(EmailID) AS LowerEmail, -- string**

**GETDATE() AS Today, -- date**

**YEAR(GETDATE()) - 5 AS PastYear -- date math**

**FROM Employees;**

**Output explanation:**

* **Salary with 10% bonus is rounded.**
* **Email is shown in lowercase.**
* **Current system date is returned.**
* **Year is calculated dynamically.**

**5.5 Inline SQL functions & Inline SQL queries**

**Inline SQL = queries written directly inside SELECT.**

**Example – Inline query to find highest salary in IT department:**

**SELECT EmpName, Salary**

**FROM Employees**

**WHERE Salary = (SELECT MAX(Salary)**

**FROM Employees**

**WHERE DeptID = 2);**

**Output explanation:**

* **In IT dept, highest salary is 60000 (Meena).**
* **Inline query (SELECT MAX(Salary)…) runs inside the main query.**

**1. What are Inline SQL Functions?**

* An **inline function** is similar to a **view**, but with parameters.
* It returns a **table** as output, not just a single value.
* It’s very useful when you want to encapsulate a frequently used query with input parameters.

*“A function that returns a query result (table) instead of just one value.”*

**2. Syntax**

CREATE FUNCTION FunctionName (@parameter datatype)

RETURNS TABLE

AS

RETURN

(

SELECT ... FROM ... WHERE column = @parameter

);

**3. Example with Employees Table**

**Step 1 – Create Inline Function**

Let’s create a function that returns all employees of a given **Department ID**:

CREATE FUNCTION fn\_GetEmployeesByDept(@DeptID INT)

RETURNS TABLE

AS

RETURN

(

SELECT EmpID, EmpName, Salary, DeptID

FROM Employees

WHERE DeptID = @DeptID

);

**Step 2 – Use the Function**

SELECT \* FROM fn\_GetEmployeesByDept(2);

**Output Explanation:**

If DeptID = 2 (IT Department), the result will be:

EmpID EmpName Salary DeptID

102 Meena 60000 2

103 John 55000 2

105 Neha 70000 2 -- (if we add new employee in IT)

Here, the function behaves like a parameterized view.  
Instead of writing a WHERE DeptID=2 query every time, we just call the function.

**4. Another Inline Function Example – Salary Above Threshold**

CREATE FUNCTION fn\_GetHighEarners(@MinSalary DECIMAL(10,2))

RETURNS TABLE

AS

RETURN

(

SELECT EmpName, Salary, DeptID

FROM Employees

WHERE Salary > @MinSalary

);

Usage:

SELECT \* FROM fn\_GetHighEarners(55000);

Output will list all employees earning more than 55,000.

**5. Difference from Scalar Functions**

* **Scalar function** → returns **one value** (e.g., max salary, count, etc.).
* **Inline function** → returns a **table** (multiple rows & columns).

So inline functions = **reusable, parameterized queries that return a table.**

**5.6 General functions**

General functions are **useful utilities** like ISNULL, COALESCE, NULLIF.

Example – Handling null values in EmailID:  
(Let’s add one more row with NULL email)

INSERT INTO Employees VALUES (105, 'Suresh', 'M', 40000, 1, NULL);

SELECT EmpName,

COALESCE(EmailID, 'No Email') AS EmailStatus

FROM Employees;

**Output explanation**:

* For Suresh, instead of NULL, it shows No Email.
* COALESCE picks the **first non-null value**.

**5.7 Duplicate functions**

These help to **remove or identify duplicates**.

Example – Add a duplicate record intentionally:

INSERT INTO Employees VALUES (106, 'John', 'M', 55000, 2, 'john@abc.com');

SELECT EmpName, Salary, COUNT(\*) AS Occurrence

FROM Employees

GROUP BY EmpName, Salary

HAVING COUNT(\*) > 1;

**Output explanation**:

* Finds duplicate entries (like John appears twice with same salary).
* COUNT > 1 means duplicates exist.