**Subqueries in SQL**

# Agenda

1. Introduction to Subqueries

2. Types of Subqueries (by result)

* Single‑Row Subquery
* Multi‑Row Subquery
* Scalar Subquery
* Correlated Subquery

3. Nested Subqueries

4. Use of EXISTS / NOT EXISTS with Subqueries

5. Comparison of Subqueries vs. JOINs

# 1) Introduction to Subqueries in SQL

A subquery (also called an inner or nested query) is a query inside another SQL statement. Subqueries are enclosed in parentheses and can return values used by the outer (main) query.

Typical comparison / subqueries operators with subqueries: =, <, <=, >, >=, <>, IN, ANY/SOME, ALL, EXISTS, NOT EXISTS.

**Beginner tip:** Read subqueries from the inside out — evaluate the inner query first, then see how the outer query uses its result.

## Use Cases at a Glance

* **Filtering Data:** use a subquery in WHERE/HAVING to filter rows based on other data.
* **Dynamic Calculations**: compute values on the fly in SELECT (e.g., averages by group).
* **Conditional Updates/Deletes:** update or delete rows conditionally using related data.
* **Aggregations:** compute MAX/AVG/COUNT in a subquery then compare in the outer query.
* **Existence checks:** test if related rows exist (EXISTS / NOT EXISTS).

# 2) Where Subqueries Can Appear

Subqueries commonly appear in three places:

* SELECT list (returns a single value per outer row → scalar subquery).
* FROM clause (called a derived table or inline view; returns a virtual table).
* WHERE/HAVING (to filter rows using comparisons to inner results).

**-- SELECT list (scalar)**

|  |
| --- |
| SELECT e.EmployeeID,  (SELECT AVG(Salary) FROM Employee WHERE DepartmentID = e.DepartmentID) AS DeptAvg FROM Employee e; |

**-- FROM clause (derived table)**

|  |
| --- |
| SELECT e.EmployeeID, e.Name, d.AvgSalary FROM Employee e JOIN (  SELECT DepartmentID, AVG(Salary) AS AvgSalary  FROM Employee  GROUP BY DepartmentID ) d ON e.DepartmentID = d.DepartmentID; |

**-- WHERE clause**

|  |
| --- |
| SELECT Name FROM Employee WHERE Salary > (  SELECT AVG(Salary) FROM Employee  ); |

# 3) Types of Subqueries (by Result)

1. **Single‑Row Subquery:** returns exactly one row (often one column).
2. **Multi‑Row Subquery:** returns multiple rows (one column).
3. **Scalar Subquery:** returns exactly one value (one row, one column).
4. **Correlated Subquery:** inner query depends on the current row of the outer query.

# I) Single‑Row Subquery

A single‑row subquery returns exactly one row (typically one column). Use comparison operators like =, <, <=, >, >=, <>. If the subquery returns more than one row, most databases raise an error.

**Q: Employees earning more than a specific employee’s salary?**

|  |
| --- |
| SELECT EmployeeID, Name FROM Employee WHERE Salary > (  SELECT Salary FROM Employee  WHERE EmployeeID = 15 ); |

**Example Error:**

If there’s any chance the Inner query / Subquery could Return multiple rows, constrain it deterministically by using [LIMIT]:

|  |
| --- |
| SELECT EmployeeID, Name FROM Employee WHERE Salary > (  SELECT Salary FROM Employee  WHERE DepartmentID = 10  ORDER BY Salary DESC  LIMIT 1 ); |

# II) Multi‑Row Subquery

A multi‑row subquery returns multiple rows. Use IN, ANY/SOME, or ALL to compare a single value with a set of values.

**Examples:**

-- **IN:** Used to check if a value matches any value in a list or subquery result.

|  |
| --- |
| SELECT EmployeeID, Name FROM Employee WHERE DepartmentID IN (  SELECT DepartmentID FROM Department  WHERE LocationID = 2 ); |

**-- ANY / SOME:** Compare against each value; True if true for at least one.

**Q. Employees earning more than ANY salary in Dept 101 (i.e., more than the minimum there)?**

|  |
| --- |
| SELECT EmployeeID, Name FROM Employee WHERE Salary > ANY (  SELECT Salary FROM Employee WHERE DepartmentID = 101 ); |

**-- ALL:** Condition must be True against every returned value.

**Q. Employees earning more than ALL salaries in Dept 101 (i.e., greater than the maximum there)?**

|  |
| --- |
| SELECT EmployeeID, Name FROM Employee WHERE Salary > ALL (  SELECT Salary FROM Employee WHERE DepartmentID = 101 ); |

# III) Scalar Subquery

A scalar subquery returns exactly one value (one row, one column). It’s common in the SELECT list to compute a per‑row value.

**Q. Show each employee and the average salary of their department?**

|  |
| --- |
| SELECT e.EmployeeID, e.Name,  (SELECT AVG(Salary) FROM Employee WHERE DepartmentID = e.DepartmentID) AS DeptAvg FROM Employee e; |

**Q. Company‑wide maximum salary repeated on each row?**

|  |
| --- |
| SELECT e.EmployeeID, e.Name,  (SELECT MAX(Salary) FROM Employee) AS MaxCompanySalary FROM Employee e; |

If a scalar subquery returns more than one row, the database raises an error. Make sure grouping/filters guarantee a single result.

# IV) Correlated Subquery

A correlated subquery references columns from the outer query, so it runs once per outer row. This is powerful but can be slower on large tables.

**Q. Employees earning above their own department's average?**

|  |
| --- |
| SELECT e.EmployeeID, e.Name, e.Salary FROM Employee e WHERE e.Salary > (  SELECT AVG(e2.Salary) FROM Employee e2  WHERE e2.DepartmentID = e.DepartmentID ); |

**Q. Employees earning above the highest salary in their department (besides themselves)?**

|  |
| --- |
| SELECT e.EmployeeID, e.Name, e.Salary FROM Employee e WHERE e.Salary > (  SELECT MAX(e2.Salary) FROM Employee e2  WHERE e2.DepartmentID = e.DepartmentID  AND e2.EmployeeID <> e.EmployeeID ); |

Often, you can rewrite correlated subqueries with JOINs to let the optimizer do the heavy lifting once.

# V) Nested Subqueries

A nested subquery is a subquery inside another subquery, enabling multi‑step filtering.

**Q. Employees working in departments whose locations are named 'New York'?**

|  |
| --- |
| SELECT EmployeeID, Name FROM Employee WHERE DepartmentID IN (  SELECT DepartmentID FROM Department  WHERE LocationID IN (  SELECT LocationID FROM Locations  WHERE LocationName = 'New York'  ) ); |

# VI) EXISTS / NOT EXISTS

EXISTS returns true if the subquery returns at least one row; NOT EXISTS is true if it returns none. These are great for semi‑joins (filter when related rows exist) and anti‑joins (filter when related rows do not exist).

**Q. Employees whose department is in location 3?**

|  |
| --- |
| SELECT e.EmployeeID, e.Name FROM Employee e WHERE EXISTS (  SELECT 1 FROM Department d  WHERE d.DepartmentID = e.DepartmentID  AND d.LocationID = 3 ); |

**Q. Departments that have no employees (anti-join)?**

|  |
| --- |
| SELECT d.DepartmentName FROM Department d WHERE NOT EXISTS (  SELECT 1 FROM Employee e  WHERE e.DepartmentID = d.DepartmentID ); |

**Tip:** Prefer NOT EXISTS over NOT IN when NULLs may be present in the subquery result, because NOT IN with NULL returns no rows.

# VII) Comparison: Subqueries vs. JOINs

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Subqueries** | **JOINs** |
| Use Case | Dynamic filtering, single values, existence tests. | Combining related rows/columns across tables. |
| Performance | Correlated subqueries can be slow on large data. | Often faster; optimizers handle joins well. |
| Complexity | Can nest deeply and get hard to read. | Typically clearer relationships. |
| Readability | Good for 'inside‑out' logic. | Good for 'relational' thinking. |
| When to prefer | When you need a single scalar or an existence check. | When you need multiple columns or to reuse an intermediate result. |

# VIII) Common Errors & NULL Gotchas

* **“Subquery returns more than one row”:** your single‑row or scalar subquery produced multiple rows → tighten filters, aggregate, or LIMIT/TOP 1.
* **“Operand should contain 1 column(s)”** : scalar context requires one column but your subquery returns multiple columns.
* **NOT IN vs NULL:** If the subquery returns any NULL, NOT IN yields UNKNOWN for all comparisons → usually returns zero rows. Prefer NOT EXISTS.
* **Data type Mismatches:** ensure the compared columns share compatible types (e.g., integers vs text).
* **Order of Execution:** ORDER BY inside a subquery only matters with LIMIT/TOP; outer ORDER BY sorts the final result.

# IX) Performance Tips & When to Prefer JOINs or CTEs

* Index the columns used for joining/filtering (e.g., DepartmentID, LocationID).
* Rewrite correlated subqueries as JOINs when possible, especially under heavy data volumes.
* Use derived tables (FROM subqueries) or CTEs to compute expensive aggregates once and reuse them.
* Use EXISTS/NOT EXISTS for existence checks; they often short‑circuit and avoid materializing full sets.
* Aggregate first in an inner query, then join results to the base table (reduces repeated work).

# X) Solved Practice Queries

## 1) Retrieve Managers of Employees?

|  |
| --- |
| SELECT DISTINCT e.Name FROM Employee e WHERE e.EmployeeID IN (  SELECT ManagerID FROM Employee  WHERE ManagerID IS NOT NULL ); |

## 2) Employees in the 'Sales' Department?

|  |
| --- |
| SELECT Name FROM Employee WHERE DepartmentID = (  SELECT DepartmentID FROM Department  WHERE DepartmentName = 'Sales' ); |

## 3) Departments with a High Earner (Earning more than 70,000)?

|  |
| --- |
| SELECT DISTINCT d.DepartmentName FROM Department d WHERE EXISTS (  SELECT 1 FROM Employee e  WHERE e.DepartmentID = d.DepartmentID  AND e.Salary > 70000 ); |

## 4) Hired After the Highest‑Paid Employee?

|  |
| --- |
| SELECT Name FROM Employee WHERE HireDate > (  SELECT HireDate FROM Employee  ORDER BY Salary DESC  LIMIT 1 ); |

## 5) Locations with Departments?

|  |
| --- |
| SELECT DISTINCT l.LocationName FROM Locations l WHERE EXISTS (  SELECT 1 FROM Department d  WHERE d.LocationID = l.LocationID ); |

## 6) Departments with No Employees?

|  |
| --- |
| SELECT d.DepartmentName FROM Department d WHERE NOT EXISTS (  SELECT 1 FROM Employee e  WHERE e.DepartmentID = d.DepartmentID ); |

## 7) Employees in Departments Located in 'New York'?

|  |
| --- |
| SELECT e.Name FROM Employee e WHERE e.DepartmentID IN (  SELECT d.DepartmentID FROM Department d  WHERE d.LocationID IN (  SELECT l.LocationID FROM Locations l  WHERE l.LocationName = 'New York'  ) ); |