
Subqueries (Inner Queries)

A **subquery** is a SELECT statement nested inside another SQL statement. It's executed first, and its result is used by the outer query. Think of it as a query within a query.

Types of Subqueries

1. **Scalar Subquery**: Returns a single value (one row, one column). It can be used almost anywhere a single value is expected, like in a WHERE clause or a SELECT list.
2. **Multi-row Subquery**: Returns multiple rows. It's typically used with operators like IN, NOT IN, ANY, and ALL in a WHERE clause.
3. **Correlated Subquery**: An inner query that depends on the outer query for its values. It is evaluated once for each row processed by the outer query, which can sometimes be inefficient.

Example Setup

Let's use these two simple tables for our examples:

employees table:

emp_id	first_name	salary	dept_id
101	Alice	90000	1
102	Bob	80000	1
103	Charlie	115000	2
104	David	72000	3

105	Eve	130000	2

departments table:

dept_id	dept_name
1	Sales
2	Engineering
3	HR

Subquery Examples

Example 1: Scalar Subquery

Find all employees who earn more than the average salary.

Here, the subquery (SELECT AVG(salary) FROM employees) calculates the average salary first. This single value ($97,400) is then used by the outer query to filter the employees.

SQL

```sql
SELECT
  first_name,
  salary
FROM
  employees
WHERE
  salary > (SELECT AVG(salary) FROM employees);
```

Result:
first_name	salary
Charlie	115000
Eve	130000

Example 2: Multi-row Subquery with IN

Find all employees who work in the 'Engineering' department.

The subquery (SELECT dept_id FROM departments WHERE dept_name = 'Engineering') returns the dept_id for Engineering (which is 2). The outer query then finds all employees whose dept_id is in that result set.

SQL

```sql
SELECT
  first_name
FROM
  employees
WHERE
  dept_id IN (SELECT dept_id FROM departments WHERE dept_name = 'Engineering');
```

Result:
first_name
Charlie
Eve

Example 3: Correlated Subquery

Find employees whose salary is the highest in their respective departments.

This is a correlated subquery because the inner query (SELECT MAX(salary)...) is linked to the outer query via the condition e2.dept_id = e1.dept_id. For each employee (e1) processed by the outer query, the inner query recalculates the max salary for that specific employee's

department.

SQL

```sql
SELECT
  first_name,
  salary,
  dept_id
FROM
  employees e1
WHERE
  salary = (SELECT MAX(salary) FROM employees e2 WHERE e2.dept_id = e1.dept_id);
```

Result:
first_name	salary	dept_id
Alice	90000	1
Eve	130000	2
David	72000	3

Common Table Expressions (CTEs)

A **Common Table Expression (CTE)** is a temporary, named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. It is defined using the WITH clause before the main query. CTEs make complex queries much more readable and maintainable.

Key Features of CTEs

- **Readability**: Breaks down complex logic into sequential, easy-to-read steps.
- **Reusability**: A CTE can be referenced multiple times in the query that follows it.
- **Recursion**: CTEs can reference themselves, which is essential for querying hierarchical data (e.g., organizational charts).

CTE Examples

Example 1: Basic CTE for Readability

Let's rewrite the "find employees in Engineering" example using a CTE. The logic is identical, but the structure is often considered cleaner.

SQL

```sql
WITH EngineeringDept AS (
    -- This CTE finds the department ID for Engineering
    SELECT dept_id
    FROM departments
    WHERE dept_name = 'Engineering'
)

-- Main query uses the CTE
SELECT
    e.first_name
FROM
    employees e
JOIN
    EngineeringDept ed ON e.dept_id = ed.dept_id;
```

Result:
first_name
Charlie
Eve

Example 2: Multiple CTEs

Calculate the average salary for the 'Sales' and 'Engineering' departments and find the

difference.

This example shows how you can define multiple CTEs, separated by a comma. Each CTE is a logical building block that feeds into the final SELECT statement.

SQL

```sql
WITH SalesAvg AS (
    SELECT AVG(salary) as avg_sales_salary
    FROM employees
    WHERE dept_id = 1
),
EngineeringAvg AS (
    SELECT AVG(salary) as avg_eng_salary
    FROM employees
    WHERE dept_id = 2
)

SELECT
    s.avg_sales_salary,
    e.avg_eng_salary,
    e.avg_eng_salary - s.avg_sales_salary AS difference
FROM
    SalesAvg s, EngineeringAvg e;
```

Result:
avg_sales_salary	avg_eng_salary	difference
85000.00	122500.00	37500.00

Example 3: Recursive CTE (Advanced)

Imagine we add a manager_id to our employees table to create a hierarchy. Let's find all employees who report, directly or indirectly, to Charlie (emp_id 103).

A recursive CTE has two parts:

1. **Anchor Member**: The base case for the recursion (e.g., Charlie himself).
2. **Recursive Member**: The part that references the CTE itself to iterate (e.g., find

employees whose manager is in the set we've already found).

SQL

```sql
-- Let's assume an updated employees table with a manager_id
-- emp_id | first_name | manager_id
-- 103    | Charlie    | NULL
-- 105    | Eve        | 103
-- 101    | Alice      | 105

WITH RECURSIVE Subordinates AS (
    -- 1. Anchor Member: Start with Charlie
    SELECT emp_id, first_name, manager_id, 0 AS level
    FROM employees
    WHERE emp_id = 103

    UNION ALL

    -- 2. Recursive Member: Join employees to the CTE
    SELECT e.emp_id, e.first_name, e.manager_id, s.level + 1
    FROM employees e
    JOIN Subordinates s ON e.manager_id = s.emp_id
)

SELECT * FROM Subordinates;
```

This query would trace the hierarchy down from Charlie, showing who reports to whom.

CTEs vs. Subqueries: When to Use Which 🤔

Feature	Subquery	Common Table Expression (CTE)
Readability	Can become confusing and hard to read when nested	**Excellent**. Breaks logic into clean, sequential steps.

	deeply.	
Placement	Nested *inside* a clause (SELECT, FROM, WHERE).	Defined *before* the main query using a WITH clause.
Reusability	Must be rewritten if you need the same result set multiple times.	**Can be referenced multiple times** within the subsequent query.
Recursion	Cannot perform recursion.	**The only way to do recursion** in standard SQL.
When to Use	✅ Best for simple, one-off lookups where the query is short and clear.	✅ Best for complex queries, multi-step logic, reusability, or recursion.

General Guideline: If you can write a query clearly with a simple subquery, it's fine. For anything more complex, a CTE is almost always the better choice for long-term readability and maintenance. Modern SQL development heavily favors CTEs.