Nested Queries (Subqueries)

Table of contents

What shape does the subquery return? 2
Correlated vs non-correlated 4
Subqueries in HAVING 6
Joins or subqueries? 7
"For all" queries 7

A **nested query** (subquery) is a query written inside another query; the containing query is the **outer query**, and nesting can go several levels deep. Subqueries let you compute an intermediate result and feed it straight into a larger query without creating a temporary table — including results, such as aggregates, that the outer query is not allowed to compute for itself. Introduced in 2026-04-13-nested-queries-views-and-generative-ai[1]; practised in week9-tutorial-applied-class-8-nested-queries-and-views[2].

The motivating case is the aggregate that cannot go in a `WHERE` clause (see sql-select-statement[3] for why):

```
-- INVALID: an aggregate cannot appear directly in WHERE
SELECT name, salary
FROM   Employee
WHERE  salary > AVG(salary);

-- Valid: the aggregate is computed by a subquery
SELECT name, salary
FROM   Employee
WHERE  salary > (SELECT AVG(salary) FROM Employee);
```

[1] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf
[2] courses/infs1200/week9-tutorial-applied-class-8-nested-queries-and-views.pdf
[3] courses/infs1200/sql-select-statement.pdf

What shape does the subquery return?

The single most useful question to ask before writing a subquery is *what do I expect this to return* — a value, a list, a table, or a yes/no? The answer decides which clause it can sit in and which operators are legal.

Type	Expected result	Operators	Goes in
Scalar	A single value	`=, <, >, !=`	`WHERE`, `HAVING`
Set	A list of values (one column)	`IN, NOT IN, ANY, ALL`	`WHERE`, `HAVING`
Table	Rows and columns (a relation)	—	`FROM`
Boolean	True/false per outer row	`EXISTS, NOT EXISTS`	`WHERE`

Scalar

A single value, compared directly. The classic failure is a scalar subquery that unexpectedly matches more than one row — the DBMS raises "subquery returned more than one row" rather than picking one.

```
SELECT name, salary
FROM   Employee
WHERE  salary > (SELECT AVG(salary) FROM Employee);
```

Set

A one-column list, tested with `IN`/`NOT IN`/`ANY`/`ALL`. Never use `=` where the subquery could return several rows.

```
-- Departments whose manager is named "Jennifer"
SELECT dName FROM Department
WHERE  mgrSSN IN (SELECT ssn FROM Employee WHERE name LIKE 'Jennifer');

-- ...and those whose manager is not
SELECT dName FROM Department
WHERE  mgrSSN NOT IN (SELECT ssn FROM Employee WHERE name LIKE 'Jennifer');
```

`ANY` succeeds if the comparison holds for **at least one** returned value; `ALL` requires it to hold for **every** returned value. `IN` is exactly `= ANY`, and the course states the equivalence with both forms written out side by side:

```sql
-- These two are the same query
SELECT DISTINCT dName FROM Department
WHERE  mgrSSN IN     (SELECT ssn FROM Employee WHERE name LIKE 'Jennifer');

SELECT DISTINCT dName FROM Department
WHERE  mgrSSN = ANY  (SELECT ssn FROM Employee WHERE name LIKE 'Jennifer');
```

(`ANY` is the course's spelling; the standard's synonym `SOME` does not appear in the material, so write `ANY`.)

```sql
-- Employees earning more than everyone in department 5
SELECT *
FROM   Employee
WHERE  salary > ALL (SELECT salary FROM Employee WHERE dNum = 5);
```

`>= ALL (…)` over a set of group totals is the idiomatic "is this the maximum?" test, and it keeps ties rather than discarding them:

```sql
-- The most common suburb(s), ties included
SELECT   Suburb AS 'Suburb Name', COUNT(*) AS 'Street Count'
FROM     Address
GROUP BY Suburb
HAVING   COUNT(*) >= ALL (SELECT COUNT(*) FROM Address GROUP BY Suburb)
ORDER BY Suburb ASC;
```

`NOT IN` expresses "none of these" — the natural way to write an exclusion:

```sql
-- Employees never granted a role by a 'ProductEngineer' administrator
SELECT E.*
FROM   Employee E
WHERE  EmployeeID NOT IN (
    SELECT DISTINCT E2.EmployeeID
    FROM   Employee E2
    JOIN   RoleGranting RG ON RG.EmployeeID = E2.EmployeeID
    JOIN   AdministrativeEmployee AE ON AE.EmployeeID = RG.AdministrationID
    WHERE  AE.Type = 'ProductEngineer'
);
```

Table

A subquery in the `FROM` clause behaves like a table and **must be given an alias**, after which it can be joined like any other:

```sql
-- Average salary per department, alongside the department name
SELECT d.dName, avgSal.avg_salary
FROM   Department d
JOIN   (SELECT   dNum, AVG(salary) AS avg_salary
        FROM     Employee
        GROUP BY dNum) AS avgSal ON d.dNumber = avgSal.dNum;
```

If the same derived table is needed twice, define it once as a view instead — see sql-views[4].

Boolean

`EXISTS`/`NOT EXISTS` test whether the subquery returns *any* row at all; they return no data themselves, only yes/no, so the subquery's target list is irrelevant (`SELECT *` is conventional).

```sql
-- Employees in a department that currently has a manager assigned
SELECT name, dNum
FROM   Employee e
WHERE  EXISTS (
    SELECT * FROM Department d
    WHERE  d.dNumber = e.dNum AND d.mgrSSN IS NOT NULL
);
```

Correlated vs non-correlated

- **Non-correlated** — the inner query is self-contained and evaluated once, independently of the outer query ("inside out"), like calling a parameterless function and reusing the result.
- **Correlated** — the inner query references an attribute of the outer query's relation, so the outer query supplies a value the inner one needs, and the subquery is conceptually re-evaluated once per outer row.

```sql
-- Non-correlated: one AVG over the whole company, reused for every row
SELECT name, dNum, salary
FROM   Employee
WHERE  salary > (SELECT AVG(salary) FROM Employee);

-- Correlated: E1.dNum ties the inner query to the current outer row,
-- so each employee is compared to their OWN department's average
SELECT E1.name, E1.dNum
FROM   Employee E1
WHERE  salary > (
```

[4] courses/infs1200/sql-views.pdf

```
    SELECT AVG(salary) FROM Employee E2 WHERE E1.dNum = E2.dNum
);
```

	Non-correlated	Correlated
Semantics	Executed once	Executed once per outer row
Dependency	Independent — runs in isolation	Depends on the outer query
Rows examined	$n + m$	$n \times m$

Because a correlated subquery re-runs per outer row it is the more expensive form, but it is also the only way to express "compared with the rest of *its own* group". `EXISTS`/`NOT EXISTS` subqueries are **always** correlated — an uncorrelated `EXISTS` would give the same answer for every row.

```
-- Employees who share a surname with some OTHER employee
SELECT COUNT(*) AS employees_with_shared_surname
FROM   Employee E1
WHERE  LastName IS NOT NULL
  AND  EXISTS (
      SELECT * FROM Employee E2
      WHERE  E1.LastName = E2.LastName AND E1.EmployeeID != E2.EmployeeID
  );
```

The inequality on the key is what makes "some *other* row" mean what it says; without it every row matches itself and the test is trivially true. The course's cleanest illustration is a matched pair of queries that differ only in the negation:

```
-- Movies that were the ONLY movie produced that year
SELECT * FROM Movie M1
WHERE  NOT EXISTS (
    SELECT * FROM Movie M2
    WHERE  M1.movieID <> M2.movieID AND M1.year = M2.year);

-- Movies that were NOT the only movie produced that year
SELECT * FROM Movie M1
WHERE  EXISTS (
    SELECT * FROM Movie M2
    WHERE  M1.movieID <> M2.movieID AND M1.year = M2.year);
```

The correlated ALL idiom

Correlating an `ALL` comparison to the outer row is how "the extreme *within its own group*" is written without any `GROUP BY` at all. The subquery is restricted to the outer row's group, and the comparison then asks whether this row beats every member of it:

```
-- The team(s) that scored the fewest runs on each day
SELECT team, day
FROM   Scores S1
WHERE  runs <= ALL (
    SELECT runs FROM Scores S2 WHERE S1.day = S2.day);
```

`S1.day = S2.day` is the whole trick: drop it and the query returns the lowest-scoring teams overall rather than per day. `<=` (rather than `<`) keeps every team tied on the day's minimum, exactly as `>= ALL` keeps the tied maxima.

Subqueries in HAVING

A subquery inside `HAVING` lets a group (sql-aggregation-and-grouping[5]) be tested against a population the `WHERE` clause has already filtered away:

```
-- Departments with >2 employees EARNING OVER 20000
SELECT   dNum, COUNT(*)
FROM     Employee E1
WHERE    salary > 20000
GROUP BY dNum
HAVING   2 < COUNT(*);

-- Departments (among those with employees earning over 20000)
-- that have >2 employees IN TOTAL
SELECT   dNum, COUNT(*)
FROM     Employee E1
WHERE    salary > 20000
GROUP BY dNum
HAVING   2 < (SELECT COUNT(*) FROM Employee E2 WHERE E1.dNum = E2.dNum);
```

Another idiom worth memorising: comparing against `MAX`/`MIN` with `>=` rather than `=` returns **all** the tied extremes rather than one arbitrary row.

```
-- Every youngest employee, ties included
SELECT E.DOB, E.FirstName, E.LastName
FROM   Employee E
WHERE  DOB >= (SELECT MAX(DOB) FROM Employee E2);
```

[5] courses/infs1200/sql-aggregation-and-grouping.pdf

Joins or subqueries?

Many nested queries can be rewritten as a plain join (sql-joins[6]) — but not all, and the two have different strengths:

- A **join** can display columns from *every* table in its `FROM` clause; a subquery-based query can only display columns from the **outer** query's tables.
- A **subquery** can compute an aggregate on the fly and feed it back to the outer query for comparison, which a join cannot.

Rule of thumb: use joins when you are displaying results from multiple tables; use subqueries when you need to compare aggregates to other values.

Nesting is also the standard fallback for the set operators. "In this result but not that one" is `EXCEPT`, and the course notes that **any `EXCEPT` query can be implemented using nested queries** — a `NOT IN` or `NOT EXISTS` subquery says the same thing and runs on dialects that lack the operator (sql-set-operations[7]).

"For all" queries

The one common shape subqueries express that neither joins nor plain aggregates do is *"related to **every** …"* — the movie stars who were in all of a year's movies, the employees who can reach all of another employee's websites. Both standard patterns for it (counting, and `NOT EXISTS` double negation) are nested-query patterns: see relational-division[8].

[6] courses/infs1200/sql-joins.pdf
[7] courses/infs1200/sql-set-operations.pdf
[8] courses/infs1200/relational-division.pdf