# INFS1200 — Week 8 Notes

## Table of contents


**Nested Queries, Views and Generative AI & SQL** **2**
- Today's outline . . . 2
- Nested queries . . . 2
- Relational division . . . 6
- Question 17 — Subquery . . . 7
- Question 18 — Correlated subquery . . . 8
- Question 19 — Division . . . 8
- Question 20 — Division . . . 9
- Views . . . 9
- Question 21 — Views . . . 10
- Generative AI & SQL . . . 11
- Summary . . . 12

**Applied Class 7: DQL Multiple Relation SQL Queries** **13**

**Reference material** **16**
- Relational Division . . . 16
- SQL Joins . . . 20
- Nested Queries (Subqueries) . . . 24
- The SELECT Statement and Its Clause Order . . . 30
- SQL Set Operations . . . 35
- SQL Views . . . 39

# Nested Queries, Views and Generative AI & SQL

Module 3, part 3 — the last SQL lecture. Continues 2026-03-30-basic-sql-ddl-and-dml[1] and 2026-04-06-aggregation-grouping-and-multiple-relation-queries[2], using the same Company and Movie schemas.

## Today's outline

- Nested queries (subqueries) — output types, correlated vs non-correlated, relational division
- Views
- Generative AI & SQL

## Nested queries

A **nested query** (subquery) is a query that appears inside another query — nesting can occur at multiple levels. The query containing the nested query is the **outer query**. Subqueries let you compute an intermediate result and feed it into a larger query, without creating a temporary table.

```
-- INVALID: can't use an aggregate directly in WHERE
SELECT name, salary
FROM   Employee
WHERE  salary > AVG(salary);

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

### Subquery output types

| Type | Expected result | Operators | Example |
|---|---|---|---|
| **Scalar** | A single value | `=, <, >, !=` | `salary > (SELECT AVG(salary) ...)` |
| **Set** | A list of values (1 column) | `IN, NOT IN, ANY, ALL` | `WHERE dept IN (SELECT dNumber ...)` |

[1] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf
[2] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf

| Type | Expected result | Operators | Example |
|---|---|---|---|
| **Table** | Rows & columns (a relation) | used in `FROM` | `FROM (SELECT ...) AS sub` |
| **Boolean** | True/false for a row | `EXISTS, NOT EXISTS` | `WHERE EXISTS (SELECT * FROM ...)` |

Always ask: *what do I expect this subquery to return* — a value, a list, a table, or a yes/no? That determines which clause it can go in and which operators are legal.

**Scalar** — a single value, for direct comparison. Common pitfall: a scalar subquery that unexpectedly returns more than one row raises an error ("subquery returned more than one row").

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

**Set** — a list of values, for `IN`/`NOT IN`/`ANY`/`ALL`. Common pitfall: never use `=` where the subquery can return multiple rows.

```
-- Departments with (not) a manager named "Jennifer"
SELECT dName FROM Department
WHERE  mgrSSN IN (SELECT ssn FROM Employee WHERE name LIKE 'Jennifer');

SELECT dName FROM Department
WHERE  mgrSSN NOT IN (SELECT ssn FROM Employee WHERE name LIKE 'Jennifer');
```

`IN` is equivalent to `= ANY`:

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

`ALL` requires the comparison to hold against *every* value returned:

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

**Table** — used in `FROM`, must be given an alias.

```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;
```

**Boolean** — `EXISTS`/`NOT EXISTS` check for the presence of *any* row; they don't return data, just yes/no.

```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
);
```

### Joins vs subqueries

Many nested queries are equivalent to a plain `JOIN` — but not always.

- **Joins** can display columns from *every* table in the `FROM` clause; a subquery-based query can only display columns from the *outer* query's table(s).
- **Subqueries** can compute an aggregate on the fly and feed it back to the outer query for comparison — an advantage over joins.

**Rule of thumb**: use a join when displaying results from multiple tables; use a subquery when comparing against an aggregate.

### Non-correlated vs correlated subqueries

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

```sql
-- Non-correlated: the AVG is computed once, then 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 outer row
SELECT E1.name, E1.dNum
FROM   Employee E1
WHERE  salary > (
    SELECT AVG(salary) FROM Employee E2 WHERE E1.dNum = E2.dNum
);
```

| | Non-correlated | Correlated |
|---|---|---|
| Semantics | Executed once | Executed once per outer row |
| Dependency | Independent — can run in isolation | Depends on the outer query |
| Runtime | `n + m` rows examined (inner + outer) | `n * m` rows examined |

### Subqueries with GROUP BY / HAVING

```sql
-- WHERE and HAVING both apply to "earning > 20000" (same filtered set)
SELECT   dNum, COUNT(*)
FROM     Employee E1
WHERE    salary > 20000
GROUP BY dNum
HAVING   2 < COUNT(*);

-- HAVING instead checks a DIFFERENT condition, over ALL employees,
-- via a correlated subquery
SELECT   dNum, COUNT(*)
FROM     Employee E1
WHERE    salary > 20000
GROUP BY dNum
HAVING   2 < (SELECT COUNT(*) FROM Employee E2 WHERE E1.dNum = E2.dNum);
```

The first finds departments with >2 employees *earning over 20000*; the second finds departments (among those with employees earning over 20000) that have >2 employees *in total*.

### EXISTS / NOT EXISTS

Subqueries using `EXISTS`/`NOT EXISTS` are always correlated. `EXISTS (subquery)` is true if the subquery's result is non-empty; `NOT EXISTS (subquery)` is true if it's empty.

```
-- 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
);
```

### Relational division

Division answers "for all"/"for every" queries — e.g. *find movie stars who were in **all** movies [produced in a given year].*

Dividing `R1 / R2`: the result has R1's columns *except* R2's, where R1 and R2 must be **division compatible** (R1's last `n` columns identically named to R2's `n` columns, `n` = R2's degree). The result contains a value `t` if `t` appears in R1 in combination with **every** tuple of R2.

```
Student Degree   Subject          Subject   Student Degree
Anna    BIT      CS114     ÷      CS114  =  Anna    BIT
Anna    BIT      CS115            CS115
Anna    BIT      CS180
Fred    BSc      CS114
Fred    BSc      CS180
```

(Fred is excluded — Fred hasn't taken `CS115`, so Fred doesn't divide evenly by `{CS114, CS115}`.)

MySQL has no `/` division operator for relations — division is expressed via **counting** or **double negation**.

### Division via counting

*"Find the movie star(s) who acted in at least all the movies produced in 1934."* For a candidate star `X`: count how many 1934 movies `X` acted in, and compare that count to the total number of 1934 movies — if they're equal, `X` acted in *all* of them.

```
SELECT   X.starID, X.name
FROM     MovieStar X
JOIN     StarsIn S ON X.starID = S.starID
JOIN     Movie M ON S.movieID = M.movieID
WHERE    M.year = 1934
GROUP BY X.starID
HAVING   COUNT(*) = (SELECT COUNT(*) FROM Movie M2 WHERE M2.year = 1934);
```

### Division via double negation

*"Find X such that there is no 1934 movie which X did not act in."* This reformulates "for all" as "there does not exist an exception":

```
SELECT starID, name
FROM   MovieStar X
WHERE  NOT EXISTS (
    SELECT *
    FROM   StarsIn S
    JOIN   Movie M ON S.movieID = M.movieID
    WHERE  M.year = 1934
      AND  M.movieID NOT IN (
          SELECT movieID FROM StarsIn M2 WHERE M2.starID = X.starID
      )
);
```

Reading from the inside out: the innermost subquery is "all movies `X` acted in"; the middle subquery is "1934 movies `X` did **not** act in"; the outer `NOT EXISTS` is "there is no such movie" — i.e. `X` acted in every 1934 movie.

## Question 17 — Subquery

*Given* `Employee [ID, Name, DepartmentID]` *and* `Department [ID, Department, ManagerID]`*, how do you find all employees managed by Michael Scott?*

A. `WHERE DepartmentID = (SELECT ID FROM Employee WHERE name="Michael Scott")` B. `WHERE DepartmentID IN (SELECT ID FROM Department WHERE ManagerID = (SELECT ID FROM Employee WHERE name="Michael Scott"))` C. `WHERE DepartmentID = (SELECT`

`DepartmentID FROM Employee WHERE name="Michael Scott")` D. `FROM Employee, Department WHERE name="Michael Scott" AND departmentID=ID`

> **Solution**
>
> **B.** We need Michael Scott's `ID` (from `Employee`), then the `Department`(s) *he manages* (`ManagerID` = his `ID`), then employees whose `DepartmentID` matches one of those departments. A incorrectly compares an `Employee.DepartmentID` to an `Employee.ID` (mismatched domains). C and D make the same mistake — they never consult `Department` at all, so they can't find who Michael *manages*, only who shares his own `DepartmentID`.

## Question 18 — Correlated subquery

*Given* `Scores [team, day, opponent, runs]`*, for*

```
SELECT team, day
FROM   Scores S1
WHERE  runs <= ALL (SELECT runs FROM Scores S2 WHERE S1.day = S2.day);
```

*which result(s) are correct?* A. `(Carp, Sun)` B. `(Bay Stars, Sun)` C. `(Swallows, Mon)` D. All of the above E. None of the above

> **Solution**
>
> **D.** The correlated subquery restricts comparison to games played on the *same day* as `S1` — so this returns the team(s) that scored the **fewest runs on their day**. On Sunday, Carp (2 runs) and Bay Stars (2 runs) tie for fewest; both qualify. On Monday, Swallows (0 runs) has the fewest. All three listed answers are genuinely in the result.

## Question 19 — Division

*Given* `R1 [Category, SecondaryCategory, Budget]` *and* `R2 [SecondaryCategory, Budget]` *(values* `Romance/10` *and* `Horror/10`*), what is* `R1 / R2`*?*

> **Solution**
>
> **Category = Drama, Comedy** (a single-column result). Only `Drama` and `Comedy` appear in `R1` paired with **both** `(Romance, 10)` *and* `(Horror, 10)` — `Action` only pairs with `(Horror, 10)`, so it's excluded. The result keeps only the columns of `R1` not in `R2` (just `Category`), not the full `(Category, SecondaryCategory, Budget)` tuples.

### Question 20 — Division

*Given `R1` as above, `R2 = {(Romance,10), (Horror,10)}`, and `R3 = {(Horror, 11)}` — what tuple is in **both** `R1/R2` and `R1/R3`?*

A. `(Drama, Romance, 10)` B. `(Drama)` C. `(Action)` D. `(Comedy, Horror, 10)` E. None of the above

> **Solution**
>
> **B.** From Question 19, `R1/R2 = {Drama, Comedy}`. `R3` has a single tuple `(Horror, 11)`, so `R1/R3` contains every category paired with `(Horror, 11)` in `R1` — both `Drama` (`Drama, Horror, 11`) and `Comedy` (`Comedy, Horror, 11`) qualify, so `R1/R3 = {Drama, Comedy}` too. The intersection of `R1/R2` and `R1/R3` is therefore `{Drama, Comedy}` — `Drama` is in both, which is what option B asserts (`Comedy` isn't offered as a choice here). A and D are the wrong shape (division results keep only the `Category` column, degree 1 — not the full 3-column tuple); C (`Action`) never appears in either division, since `Action` only pairs with `(Horror, 10)`, not `(Romance, 10)` or `(Horror, 11)`.

### Views

A **view** is a single table derived from other tables (base tables or other views).

- **Virtual** — does not physically exist on disk; recomputed each time it's queried.
- **Materialized** — physically stored, and must be refreshed when base tables change.

```
CREATE VIEW <view name> (<column name> {, <column name>}) AS <select statement>;
```

```
-- Count, gender and average salary of employees, per department
CREATE VIEW DepEmpStatus AS
SELECT   dNumber, dName, sex, COUNT(*) AS employeeNumber, AVG(salary) AS avgSalary
FROM     Department AS D
JOIN     Employee AS E ON D.dNumber = E.dNum
GROUP BY dNum, sex;

SELECT * FROM DepEmpStatus;
```

Given `DepEmpStatus` but *not* `Department`/`Employee` directly, a user can see aggregate departmental statistics without access to confidential per-employee data (address, salary).

### Benefits of views

- **Simplification** — hide the complexity of underlying tables.
- **Security** — hide sensitive columns from some users.
- **Computed columns** — computed on the fly.
- **Logical data independence** — users/programs querying the view are insulated from changes to the underlying logical schema.

### View updates and dropping

Updates to a view must ultimately happen on the base table(s) — this can be ambiguous or difficult in general, so DBMSs restrict updates to simple, single-table **updatable views** (e.g. a view of "employees from department X" that a department manager can edit directly).

```
DROP VIEW [IF EXISTS] view_name [, view_name] ... [RESTRICT | CASCADE];
```

`DROP TABLE ... RESTRICT` refuses to drop a table with views defined on it; `DROP TABLE ... CASCADE` drops the table *and* recursively drops any dependent views.

## Question 21 — Views

*Given* `R [a, b, c]`,

```
CREATE VIEW V AS SELECT a+b AS d, c FROM R;
SELECT d, SUM(c) FROM V GROUP BY d HAVING COUNT(*) <> 1;
```

*which tuple is returned?* A. `(2,3)` B. `(3,12)` C. `(5,9)` D. All are correct E. None are correct

**Solution**

**C.** Computing `d = a+b` for every row of `R [a,b,c] = (1,1,3), (1,2,3), (2,1,4), (2,3,5), (2,4,1), (3,2,4), (3,3,6)` gives view rows `(d=2,c=3), (d=3,c=3), (d=3,c=4), (d=5,c=5), (d=6,c=1), (d=5,c=4), (d=6,c=6)`. Grouping by `d`:

| d | rows (c values) | COUNT(*) | SUM(c) |
|---|---|---|---|
| 2 | 3 | 1 | 3 |
| 3 | 3, 4 | 2 | 7 |
| 5 | 5, 4 | 2 | 9 |
| 6 | 1, 6 | 2 | 7 |

`HAVING COUNT(*) <> 1` excludes `d=2` (only one row), leaving `(3,7), (5,9), (6,7)`. `(5, 9)` is genuinely in the result; `(2,3)` was excluded by the `HAVING`, and `(3,12)` doesn't match `d=3`'s actual sum of 7.

### Views as a cleaner alternative to nested subqueries

```sql
-- Ugly: nested subquery re-derives the per-department total twice
SELECT   d.dName
FROM     Department d,
         (SELECT dNum, SUM(salary) AS departmentWage
          FROM   Employee GROUP BY dNum) AS Temp
WHERE    d.dNumber = Temp.dNum
  AND    Temp.departmentWage = (
      SELECT MAX(departmentWage)
      FROM   (SELECT SUM(salary) AS departmentWage
              FROM   Employee GROUP BY dNum) AS Temp2
  );

-- Cleaner: define the intermediate result as a view, once
CREATE VIEW Temp AS
SELECT   dNum, SUM(salary) AS departmentWage
FROM     Employee
GROUP BY dNum;

SELECT *
FROM   Temp
WHERE  departmentWage IN (SELECT MAX(departmentWage) FROM Temp);
```

## Generative AI & SQL

Generative AI uses machine learning to produce new content (text, images, code, music) from a prompt. It can draft code/queries, explain concepts, summarise data, and much more — but it is not infallible.

### Risks to keep in mind

- **Hallucination** — confident-sounding but false/misleading output: fake sources, incorrect facts, invented code/data/people.
- **Deepfakes** — digitally altered video/image/audio depicting someone saying or doing something they didn't; hard to detect, can spread misinformation or damage reputations.

- **Bias** — AI trained on human-generated data can repeat or amplify existing biases (gender/racial bias in language or image models, cultural bias from uneven data representation, confirmation bias in recommendations).

**Five tips for responsible use**: remember AI is a tool, not a person; critically evaluate its output; investigate anything that "feels off"; keep private information private; use AI to *elevate* your skills, not replace them.

### Three GenAI-assisted SQL workflows

1. **Full query workflow (schema + data)** — share both the schema and sample data; the AI can generate a full query *and* the expected result. Good for rapid prototyping and teaching SQL end-to-end. Caution: verify the output manually, since results can still hallucinate.
2. **Schema-only query generation** — share only the schema (useful when the data itself is privacy-sensitive). Good for planning queries before you have data access. Caution: without data, the AI may make incorrect assumptions — review generated queries for correctness.
3. **Query explanation & debugging** — paste an existing query and ask the AI to explain it, or help debug syntax errors, logic problems, or unexpected results. Caution: explanations can still contain inaccuracies — cross-check against the schema and expected logic.

### Learning SQL yourself, while building AI confidence

This course's priority is that **you** learn to write SQL independently — you'll be tested on this without AI tools (the final exam, and the Assignment 2 oral interview). At the same time, you're encouraged to use AI *in* Assignment 2 to test/check queries and deepen your understanding — the goal is to use AI as a supportive partner, not a substitute for your own skills. The RiPPLE weekly activities are scaffolded to build AI literacy progressively: designing your own questions grounded in course concepts, practising prompt engineering, critical/metacognitive reflection, and structured peer review — moving from passive AI use to critical, ethical, effective collaboration with AI tools.

## Summary

You should now be able to write nested (correlated and non-correlated) subqueries, use relational division (via counting or double negation), create/drop views, and use generative AI tools for

SQL responsibly. This completes the Module 3 lecture content — see week8-tutorial-applied-class-7-multiple-relation-queries[3] and week9-tutorial-applied-class-8-nested-queries-and-views[4] for practice.

## Applied Class 7: DQL Multiple Relation SQL Queries

Practice for 2026-04-06-aggregation-grouping-and-multiple-relation-queries[5]. Same BestTechLtd schema as week6-tutorial-applied-class-5-basic-sql-ddl-and-dml[6] and week7-tutorial-applied-class-6-aggregation-and-grouping[7]:

```
Employee               [EmployeeID, FirstName, LastName, DOB, PasswordHash, PasswordSalt]
AdministrativeEmployee [EmployeeID, Level, Type]
Role                   [RoleID, Name, Description]
RoleGranting           [EmployeeID, RoleID, AdministrationID, Timestamp]
Permission             [WebsiteURI, RoleID, GrantType, Description]

AdministrativeEmployee.EmployeeID references Employee.EmployeeID
RoleGranting.EmployeeID references Employee.EmployeeID
RoleGranting.RoleID references Role.RoleID
RoleGranting.AdministrationID references AdministrativeEmployee.EmployeeID
Permission.RoleID references Role.RoleID
```

### Question 1

*Return the distinct role ID, name and descriptions of all roles which permit access to commercial websites (URI ends with `.com`).*

> **Working**
>
> ```sql
> SELECT DISTINCT R.RoleID, R.Name, R.Description
> FROM   Role R
> JOIN   Permission P ON P.RoleID = R.RoleID
> WHERE  P.WebsiteURI LIKE '%.com';
> ```

[3] courses/infs1200/week8-tutorial-applied-class-7-multiple-relation-queries.pdf
[4] courses/infs1200/week9-tutorial-applied-class-8-nested-queries-and-views.pdf
[5] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[6] courses/infs1200/week6-tutorial-applied-class-5-basic-sql-ddl-and-dml.pdf
[7] courses/infs1200/week7-tutorial-applied-class-6-aggregation-and-grouping.pdf

### Question 2

*Return the First Name, Last Name, Level, Type and Salary of all Administrative Employees.*

Working

```
SELECT FirstName, LastName, Level, Type, Salary
FROM   AdministrativeEmployee AE
JOIN   Employee E ON E.EmployeeID = AE.EmployeeID;
```

### Question 3

*For every role, find the number of websites it gives access to. Two columns:* `RoleId`*, number of website URIs.*

Working

```
SELECT     R.RoleID, COUNT(DISTINCT WebsiteURI)
FROM       Role R
LEFT JOIN  Permission P ON P.RoleId = R.RoleID
GROUP BY   R.RoleID;
```

A `LEFT JOIN` (not an inner join) is needed here so that roles with **zero** permissions still appear in the result, with a count of 0.

### Question 4

*For Employee Sofia Gonzalez, return the distinct name and description of the role(s) she was granted, along with the description of the associated permissions.*

Working

```
SELECT DISTINCT R.Name, R.Description, P.Description
FROM   Employee E
JOIN   RoleGranting RG ON E.EmployeeID = RG.EmployeeID
JOIN   Role R ON RG.RoleID = R.RoleID
LEFT JOIN Permission P ON P.RoleID = RG.RoleID
WHERE  E.FirstName = 'Sofia' AND E.LastName = 'Gonzalez';
```

### Question 5

*The co-founders of BestTechLtd can be identified as either: (A) Administrative Employees with a* `'LegacyEngineer'` *type; (B) Employees granted the* `FinancialPerformanceOverview` *role; or (C) Employees with permission to* view* (but not edit) `https://grafana.besttechltd.com`. Return the EmployeeID, first and last name of all co-founders.*

*Restriction: use at least one set operation.*

> **i** Working

```sql
SELECT E.EmployeeID, E.FirstName, E.LastName
FROM   Employee E
JOIN   AdministrativeEmployee AE ON AE.EmployeeID = E.EmployeeID
WHERE  AE.Type = 'LegacyEngineer'
UNION
SELECT E.EmployeeID, E.FirstName, E.LastName
FROM   Employee E
JOIN   RoleGranting RG ON RG.EmployeeID = E.EmployeeID
JOIN   Role R ON R.RoleID = RG.RoleID
WHERE  R.Name = 'FinancialPerformanceOverview'
UNION
SELECT E.EmployeeID, E.FirstName, E.LastName
FROM   Employee E
JOIN   RoleGranting RG ON RG.EmployeeID = E.EmployeeID
JOIN   Permission P ON P.RoleID = RG.RoleID
WHERE  P.WebsiteURI = 'https://grafana.besttechltd.com' AND P.GrantType =
 ↪  'View';
```

Three independent conditions, each a join producing the same `(EmployeeID, FirstName, LastName)` shape, combined with `UNION` (which also conveniently de-duplicates employees matching more than one condition).

### Question 6

*Return all employees who have been granted at least two roles,* ***and*** *can edit at least four distinct website URIs.*

Working

```sql
SELECT   EmployeeID
FROM     RoleGranting RG
GROUP BY EmployeeID
HAVING   COUNT(RoleID) >= 2
INTERSECT
SELECT   EmployeeID
FROM     RoleGranting RG
JOIN     Permission P ON P.RoleID = RG.RoleID
WHERE    P.GrantType = 'Edit'
GROUP BY EmployeeID
HAVING   COUNT(DISTINCT WebsiteURI) >= 4;
```

`INTERSECT` combines two independently-grouped-and-filtered employee lists — "granted 2 roles" **and** "can edit 4 distinct websites" — rather than trying to express both conditions in one `GROUP BY/HAVING` (which would conflate the two different counts). Note `INTERSECT` isn't implemented in MySQL — see 2026-04-06-aggregation-grouping-and-multiple-relation-queries[8] for how to rewrite it as a self-join.

# Reference material

## Relational Division

**Division** is the operation that answers *"for all" / "for every"* questions — find the X that are related to **every** Y, such as the movie stars who appeared in *all* the movies produced in a given year, or the employees who can access *all* the websites some other employee can. It is the one common query shape that neither a join nor a simple aggregate expresses directly, which is why it gets its own name and its own two standard SQL patterns. Introduced in 2026-04-13-nested-queries-views-and-generative-ai[9]; applied in week9-tutorial-applied-class-8-nested-queries-and-views[10].

### The operation

Dividing $R1/R2$ requires the operands to be **division compatible**: R1's last $n$ columns must be identically named to R2's $n$ columns, where $n$ is R2's degree. The result has R1's columns

[8] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[9] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf
[10] courses/infs1200/week9-tutorial-applied-class-8-nested-queries-and-views.pdf

*except* R2's, and contains a value $t$ only if $t$ appears in R1 in combination with **every** tuple of R2.

```
Student Degree   Subject          Subject   Student Degree
Anna    BIT      CS114     ÷      CS114  =  Anna    BIT
Anna    BIT      CS115            CS115
Anna    BIT      CS180
Fred    BSc      CS114
Fred    BSc      CS180
```

Fred is excluded: he hasn't taken `CS115`, so he doesn't divide evenly by `{CS114, CS115}`. Two things about the *shape* of the result are easy to get wrong:

- The result is **narrower** than R1 — it keeps only the columns not contributed by R2. Dividing a 3-column relation by a 2-column one gives a 1-column answer, not whole 3-column tuples.
- Extra rows in R1 are harmless; only the *required* combinations matter. Anna still qualifies despite also having taken `CS180`, which isn't in the divisor at all.

Dividing the same relation by successively larger divisors shows how the demand tightens. With `A [sno, pno]` holding `s1`→`{p1,p2,p3,p4}`, `s2`→`{p1,p2}`, `s3`→`{p2}`, `s4`→`{p2,p4}`:

| Divisor | Requires | `A / B` |
|---|---|---|
| `B1 = {p2}` | `p2` | `s1, s2, s3, s4` |
| `B2 = {p2, p4}` | `p2` **and** `p4` | `s1, s4` |
| `B3 = {p1, p2, p4}` | all three | `s1` |

Every divisor row is an extra hurdle, so the result can only shrink as the divisor grows — a useful sanity check on an exam answer. And an empty divisor would be satisfied by everybody, which is the degenerate end of the same pattern.

### In SQL

The first move on any division question is to name the dividend and the divisor before writing a line of SQL. "Find the movie star(s) who acted in *at least all* the movies produced in the year 1934" resolves to

```
StarsIn  ÷  (the movieIDs of Movies from 1934)  =  the qualifying stars
```

— the dividend is the relationship table, the divisor is the filtered set of things that must all be covered, and the result is the dividend's remaining column.

SQL has no `/` operator for relations, so division is expressed either by **counting** or by **double negation**. Both are worth having ready: the counting form is shorter, while the double-negation form is a direct transcription of the logical reading of the question.

The two are the same picture read from opposite sides. Draw the movies produced in 1934 as one circle and the movies X acted in as another; X qualifies exactly when the first circle sits wholly inside the second.

| Form | What it checks |
|---|---|
| **Counting** | The **overlap** is as large as the whole first circle — X acted in as many 1934 movies as there are 1934 movies |
| **Double negation** | The part of the first circle **outside** the second is **empty** — there is no 1934 movie X did not act in |

### Via counting

For a candidate X, count how many of the required Ys it is related to, and compare that against the total number of Ys. Equal counts mean X is related to all of them.

```sql
-- Star(s) who acted in at least all the movies produced in 1934
SELECT   X.starID, X.name
FROM     MovieStar X
JOIN     StarsIn S ON X.starID = S.starID
JOIN     Movie M ON S.movieID = M.movieID
WHERE    M.year = 1934
GROUP BY X.starID
HAVING   COUNT(*) = (SELECT COUNT(*) FROM Movie M2 WHERE M2.year = 1934);
```

The `WHERE M.year = 1934` restricts the count to the relevant movies, and the scalar subquery in `HAVING` supplies the target (sql-aggregation-and-grouping[11], sql-nested-queries[12]).

### Via double negation

Reformulate "X is related to every Y" as "there does **not exist** a Y that X is not related to":

[11] courses/infs1200/sql-aggregation-and-grouping.pdf
[12] courses/infs1200/sql-nested-queries.pdf

```sql
-- "Find X such that there is no 1934 movie which X did not act in"
SELECT starID, name
FROM   MovieStar X
WHERE  NOT EXISTS (
    SELECT *
    FROM   StarsIn S
    JOIN   Movie M ON S.movieID = M.movieID
    WHERE  M.year = 1934
      AND  M.movieID NOT IN (
          SELECT movieID FROM StarsIn M2 WHERE M2.starID = X.starID
      )
);
```

Read it inside out: the innermost subquery is "all the movies X acted in"; the middle one is "1934 movies X did **not** act in"; the outer `NOT EXISTS` says "there is no such movie" — i.e. X acted in every one. The correlation on `X.starID` is what makes the inner queries mean "this star" rather than "any star".

### Recognising it in the wild

The giveaway phrasings are *all*, *every*, and *at least all* — the last of which is division rather than equality: X may reach more than the divisor requires and still qualify. The same skeleton, with a subquery supplying the divisor, answers "employees with access to at least all the websites `E0007` has access to": there is no website `E0007` can reach that this employee cannot.

```sql
SELECT E.FirstName, E.LastName
FROM   Employee E
WHERE  NOT EXISTS (
    SELECT P.WebsiteURI
    FROM   Employee E2
    JOIN   RoleGranting RG1 ON RG1.EmployeeID = E2.EmployeeID
    JOIN   Permission P ON P.RoleID = RG1.RoleID
    WHERE  E2.EmployeeID = 'E0007'
      AND  P.WebsiteURI NOT IN (
          SELECT P2.WebsiteURI
          FROM   RoleGranting RG2
          JOIN   Permission P2 ON P2.RoleID = RG2.RoleID
          WHERE  RG2.EmployeeID = E.EmployeeID
      )
);
```

## SQL Joins

A **join** combines related tuples from two (or more) relations into a single wider result relation. Joins are how a normalised schema is put back together at query time: the foreign keys created by er-to-relational-mapping[13] are precisely the paths a join follows. Introduced in 2026-04-06-aggregation-grouping-and-multiple-relation-queries[14]; practised in week8-tutorial-applied-class-7-multiple-relation-queries[15].

### Renaming and qualification

With more than one table in scope, attribute names can be ambiguous. Two ways to resolve it: qualify the name (`table.attribute`), or declare an alias with `AS` (which may be omitted) and qualify with that. Aliasing is not just cosmetic — it is what makes a **self-join** expressible at all, since a table joined to itself needs two distinct names.

```
SELECT E.name, salary, 1.17 * salary AS 'includingSuper'
FROM   Employee E
WHERE  dNum = 4;
```

### Cartesian product

`R1 × R2` pairs *every* row of `R1` with *every* row of `R2`; the result schema is the concatenation of both schemas, and if $|R1| = m$ and $|R2| = n$ then $|R1 \times R2| = m \times n$. A join, written $\bowtie$, is built on top of that product: it is the product restricted to the pairs that satisfy a condition.

```
SELECT *
FROM   MovieStar, StarsIn;
```

This is almost never what you want — a join is a Cartesian product with a condition that keeps only the meaningfully related pairs. Forgetting the condition is exactly how a query silently returns thousands of rows.

### Two syntaxes for the same thing

[13] courses/infs1200/er-to-relational-mapping.pdf
[14] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[15] courses/infs1200/week8-tutorial-applied-class-7-multiple-relation-queries.pdf

```
SELECT <attribute list>
FROM   <table> {<type of join> JOIN <table to join to> ON <join attributes>}
[WHERE search condition]

-- equivalently, as a Cartesian product plus a join condition:
SELECT <attribute list>
FROM   <table list of more than one table>
[WHERE join condition AND search condition]
```

```
-- Names of the managers of each department, older comma form
SELECT E.name, D.dName
FROM   Department AS D, Employee AS E
WHERE  D.mgrSSN = E.ssn;

-- ...and the explicit JOIN form
SELECT E.name, D.dName
FROM   Department AS D
JOIN   Employee AS E ON D.mgrSSN = E.ssn;
```

`JOIN/ON` is generally preferred, because it can be more efficiently optimised by the DBMS's query optimiser and it keeps the join condition visually separate from the filtering condition — but in the context of this course the two forms are equivalent. These are also the *only* two join syntaxes the course uses: there is no `NATURAL JOIN` and no `USING (…)` anywhere in the material, so a join condition is always written out in full.

### Kinds of join

| Kind | Join condition |
|---|---|
| **Equi-join** | Tuples must *agree* on a pair of attributes — the condition uses only `=` |
| **Theta-join** | The most general form: the condition uses any $\theta \in \{=, \neq, <, >, \leq, \geq\}$ |

A theta-join is worth reaching for whenever the relationship between the rows is an inequality rather than a match. It is also easy to misread: `SELECT DISTINCT S1.sName, S1.age FROM Student S1 JOIN Student S2 ON S1.age > S2.age` returns **all the students who are older than the youngest student(s)** — every student for whom *some* younger student exists. That is everybody except the youngest, not "the oldest student". The `DISTINCT` matters too: a student older than three others is paired three times by the join and would otherwise appear three times.

### Inner vs outer joins

- **Inner join** — a tuple appears in the result only if matching tuples exist in *both* relations; unmatched rows are dropped silently. Using the `JOIN` keyword on its own specifies an inner join, so plain `JOIN` and `INNER JOIN` mean the same thing.
- **Outer join** — the inner join result *plus* the unmatched rows, with `NULL` filling the columns that had no match:
  - `LEFT JOIN` — all rows from the first table.
  - `RIGHT JOIN` — all rows from the second table.
  - `FULL OUTER JOIN` — all rows from both. **Not implemented in MySQL.**

The four are easiest to hold in mind on one small instance. Given `R [A, B]` = `{(1,2), (3,3)}` and `S [B, C]` = `{(2,4), (4,6)}`, joined on `B`, only `(1,2)` and `(2,4)` match:

| Join | Result `[A, B, C]` |
|---|---|
| `R JOIN S ON R.B = S.B` | `(1, 2, 4)` |
| `R LEFT JOIN S ON R.B = S.B` | `(1, 2, 4)`, `(3, 3, NULL)` |
| `R RIGHT JOIN S ON R.B = S.B` | `(1, 2, 4)`, `(NULL, 4, 6)` |
| `R FULL OUTER JOIN S ON R.B = S.B` | `(1, 2, 4)`, `(3, 3, NULL)`, `(NULL, 4, 6)` |

The unmatched row is *padded*, not dropped: `(3, 3)` keeps its own `A` and `B` values and takes a null only in `S`'s column.

```
-- Misses departments that have no manager:
SELECT    D.dName, E.name
FROM      Department AS D
JOIN      Employee AS E ON D.mgrSSN = E.ssn;

-- Includes them, with NULL for the manager's name:
SELECT    D.dName, E.name
FROM      Department AS D
LEFT JOIN Employee AS E ON D.mgrSSN = E.ssn;
```

The choice matters most when **counting** (see sql-aggregation-and-grouping[16]): an inner join makes the zero-cases disappear entirely rather than counting them as zero.

```
-- Every role, including those granting access to no websites at all
SELECT    R.RoleID, COUNT(DISTINCT WebsiteURI)
FROM      Role R
LEFT JOIN Permission P ON P.RoleId = R.RoleID
GROUP BY  R.RoleID;
```

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

The `NULL`s an outer join introduces are the query's own doing, not missing data in the base tables — see sql-nulls[17].

### Self-joins

A recursive relationship (an employee's manager is also an employee) is queried by joining a table to itself under two aliases:

```sql
-- Employees who earn less than their manager
SELECT A.name AS employee, A.salary AS employeeSalary,
       B.name AS manager,  B.salary AS managerSalary
FROM   Employee A
JOIN   Employee B ON A.mgrSSN = B.ssn
WHERE  A.salary < B.salary;
```

The same trick chains a relation to itself to traverse two hops — e.g. pairing each flight with the flights departing from where it lands, to count one-stopover connections:

```sql
SELECT   F1.Origin, F2.Destination, COUNT(*)
FROM     Flight F1, Flight F2
WHERE    F1.Destination = F2.Origin
GROUP BY F1.Origin, F2.Destination;
```

### Joining more than two tables

Each additional `JOIN … ON` extends the row further:

```sql
-- Stars in "Gone with the Wind"
SELECT S.starID, MS.name, S.role, M.title
FROM   StarsIn S
JOIN   MovieStar MS ON S.starID = MS.starID
JOIN   Movie M ON S.movieID = M.movieID
WHERE  M.title LIKE 'Gone with the Wind';
```

`StarsIn` here is a pure link table: it exists only to connect `Movie` to `MovieStar`, and a query that needs attributes from both ends must pass through it.

[17] courses/infs1200/sql-nulls.pdf

### Choosing the join condition

Most wrong-join bugs are semantic, not syntactic — the query runs and returns plausible rows that answer a different question. The condition should almost always match a **foreign key to the primary key it references** (foreign-keys[18]). For "employees and the department they work in", that is `Employee.dNum = Department.dNumber`; joining `Employee.ssn` to `Department.mgrSSN` instead answers "departments this employee *manages*", and joining `Employee.mgrSSN` to `Department.mgrSSN` matches managers to managers. Before writing `ON`, name the relationship the join is meant to follow.

Joins are not the only way to combine relations: sql-set-operations[19] stack result sets as new *rows* rather than new columns, and sql-nested-queries[20] feed one query's result into another.

The number of joins a query needs is also the running cost of a well-normalised schema — the trade-off normal-forms[21] discusses under denormalisation, and the operation decomposition[22] must keep lossless.

## Nested Queries (Subqueries)

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[23]; practised in week9-tutorial-applied-class-8-nested-queries-and-views[24].

The motivating case is the aggregate that cannot go in a `WHERE` clause (see sql-select-statement[25] 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
```

[18] courses/infs1200/foreign-keys.pdf
[19] courses/infs1200/sql-set-operations.pdf
[20] courses/infs1200/sql-nested-queries.pdf
[21] courses/infs1200/normal-forms.pdf
[22] courses/infs1200/decomposition.pdf
[23] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf
[24] courses/infs1200/week9-tutorial-applied-class-8-nested-queries-and-views.pdf
[25] courses/infs1200/sql-select-statement.pdf

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

### 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:

```
-- 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`.)

```
-- 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:

```
-- 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:

```
-- 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:

```
-- 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[26].

### 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).

```
-- 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.

```
-- 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,
```

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

```sql
-- so each employee is compared to their OWN department's average
SELECT E1.name, E1.dNum
FROM   Employee E1
WHERE  salary > (
    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.

```sql
-- 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:

```sql
-- 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[27]) 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);
```

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

### Joins or subqueries?

Many nested queries can be rewritten as a plain join (sql-joins[28]) — 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[29]).

### "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[30].

## The SELECT Statement and Its Clause Order

`SELECT` is the retrieval half of SQL, and the reason SQL is called a **declarative** language: you specify what the result should look like and the DBMS decides the execution plan. The result of any SQL query is itself a table, which is what makes queries composable — it can be joined (sql-joins[31]), nested inside another query (sql-nested-queries[32]), stacked with another result set (sql-set-operations[33]), or stored as a view (sql-views[34]). Introduced in 2026-03-30-basic-sql-ddl-and-dml[35] and extended across 2026-04-06-aggregation-grouping-and-multiple-relation-queries[36] and 2026-04-13-nested-queries-views-and-generative-ai[37].

[28] courses/infs1200/sql-joins.pdf
[29] courses/infs1200/sql-set-operations.pdf
[30] courses/infs1200/relational-division.pdf
[31] courses/infs1200/sql-joins.pdf
[32] courses/infs1200/sql-nested-queries.pdf
[33] courses/infs1200/sql-set-operations.pdf
[34] courses/infs1200/sql-views.pdf
[35] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf
[36] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[37] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf

### The full clause skeleton

```
SELECT [DISTINCT] <target list>
FROM   <table list>
[WHERE search condition]
[GROUP BY <grouping attributes>]
[HAVING <group conditions>]
[ORDER BY column [ASC|DESC] {, column [ASC|DESC]}];
```

Only `SELECT` and `FROM` are compulsory; the clauses must appear in this order when present.

### Logical evaluation order

The order the clauses are *written* is not the order they take effect. Reading a query as a pipeline — each stage handing a table to the next — explains most of SQL's otherwise arbitrary-looking rules:

| # | Clause | Operates on | Produces |
|---|---|---|---|
| 1 | `FROM` | The named tables (joined, if more than one) | One combined table |
| 2 | `WHERE` | Individual **rows** | The rows that satisfy the condition |
| 3 | `GROUP BY` | The surviving rows | One **group** per distinct combination of grouping attributes |
| 4 | `HAVING` | Whole **groups** | The groups that satisfy the condition |
| 5 | `SELECT` | Each group (or each row, if no grouping) | The output columns |
| 6 | `ORDER BY` | The finished result table | The same rows, sorted |

Three of the course's recurring rules fall straight out of this:

- **An aggregate cannot appear in `WHERE`.** At stage 2 no groups exist yet, so there is nothing to aggregate over — `WHERE salary > AVG(salary)` fails with *invalid use of group function*, and the fix is to compute the aggregate in a subquery instead (see sql-nested-queries[38]).

[38] courses/infs1200/sql-nested-queries.pdf

- **`HAVING` may use aggregates, `WHERE` may not**, because `HAVING` runs at stage 4, after the groups exist.
- **`WHERE` filters rows, `HAVING` filters groups.** They are not interchangeable, and a query can sensibly use both — see sql-aggregation-and-grouping[39].

The placement of `SELECT` before `ORDER BY` is not a guess: a sort key may be given as a *column position in the target list* (`ORDER BY 2 DESC`), which is only meaningful if the target list has already been built. Correspondingly, the two clauses that must be able to see raw rows — `WHERE` and `GROUP BY` — can only name attributes of the tables in `FROM`, never an alias invented by `SELECT`.

### Projection and selection

The course names the two halves of a simple query after the relational operations they perform:

| Clause | Operation | Cuts the table |
|---|---|---|
| `SELECT` | **Projection** — choose the attributes | **Vertically** |
| `WHERE` | **Selection** — choose the tuples | **Horizontally** |

Note the terminological trap: the relational operation called *selection* is the `WHERE` clause, **not** the `SELECT` clause.

### Projection, duplicates and bag semantics

```
SELECT [DISTINCT] (<attribute list> | *)
FROM   <table list>
[WHERE <condition>];
```

SQL relations are **bags** (multisets), not sets: projection does **not** eliminate duplicates by default, unlike the pure relational-model[40]. `DISTINCT` restores set semantics; `*` is the wildcard for "all columns".

```
-- All the years a movie was produced, with and without duplicates
SELECT year FROM Movie;
SELECT DISTINCT year FROM Movie;
```

[39] courses/infs1200/sql-aggregation-and-grouping.pdf
[40] courses/infs1200/relational-model.pdf

The consequence to watch for: projecting away the columns that made rows distinct leaves *duplicate* rows behind. Projecting `(score1, score2)` from a table of match results keeps two separate `(5, 3)` rows if two different fixtures both ended 5–3; `SELECT DISTINCT` collapses them into one and the row count drops accordingly.

### Expressions and aliases

The target list can contain arithmetic (`+ - * /`) over numeric attributes, string/date functions, and aggregates — each optionally renamed with `AS`:

```
-- Names, salaries, and salaries with a 17% loading, for dept 6
SELECT name, salary, 1.17 * salary AS 'includingSuper'
FROM   Employee
WHERE  dNum = 6;

-- Build an employee code FirstName-LastName-yyyy
SELECT CONCAT(FirstName, '-', LastName, '-', YEAR(DOB))
FROM   Employee;
```

Aliases also disambiguate attributes across tables and enable self-joins — see sql-joins[41].

### The WHERE clause

The `<search condition>` is a conditional (Boolean) expression identifying the tuples to retrieve. It may contain:

| Form | Use |
|---|---|
| `=`, `<>`, `<`, `<=`, `>`, `>=` | Compare an attribute to a value or another attribute |
| `AND`, `OR`, `NOT` | Combine conditions — bracket them, the grouping is the whole meaning |
| `LIKE` | Substring matching: `%` is zero or more arbitrary characters, `_` is exactly one |
| `IN` | Membership in a list: `lastName IN ('Jones', 'Wong', 'Harrison')` |
| `IS` | Null-checking — `dNum IS NULL`; `=` does **not** work here, see sql-nulls[42] |

[41] courses/infs1200/sql-joins.pdf
[42] courses/infs1200/sql-nulls.pdf

| Form | Use |
|---|---|
| `+ - * /`, date/time functions | Arithmetic on the row's own values: `salary * 2 > 50000`, `YEAR(GETDATE() - dob) > 55` |
| `BETWEEN … AND …` | Inclusive range: `salary BETWEEN 10000 AND 30000` |

```sql
-- Names of employees in dept 4 earning > 25000, or dept 5 earning > 30000
SELECT name
FROM   Employee
WHERE  (dNum = 4 AND salary > 25000) OR (dNum = 5 AND salary > 30000);

-- Permissions to edit commercial (.com) websites
SELECT WebsiteURI
FROM   Permission
WHERE  GrantType = 'Edit' AND WebsiteURI LIKE '%.com';
```

A disjunction of conjunctions is exactly as literal as it looks: a row qualifies only if it satisfies *one whole bracket*. Rows with impressive values in the right columns but the wrong combination — 8 runs scored but 12 conceded, against `(runsFor >= 6 AND runsAgainst <= 4)` — do not qualify.

Dates are compared either as a quoted `'YYYY-MM-DD'` literal or as the bare number in that order; both forms appear in the course and both work, because the ordering is the same either way:

```sql
SELECT name, dob FROM Employee WHERE dob < '1965-01-01';
SELECT name, dob FROM Employee WHERE dob < 19650101;
```

### Grouping the connectives

The course brackets its compound conditions rather than leaning on precedence, and that is the habit to copy: `(A AND B) OR (C AND D)` written out is unambiguous to you and to the marker. Where precedence still decides the meaning, it is **`NOT` first, then `AND`, then `OR`** — so `NOT` binds to the single comparison next to it, not to the rest of the condition. sql-dml[43] works that case through on a `DELETE`, where misreading it costs the whole table rather than a few rows.

Written without brackets, `A AND B OR C AND D` groups as `(A AND B) OR (C AND D)` — which is usually what was meant, but relying on that is how a condition ends up meaning something you did not intend. Bracket it.

[43] courses/infs1200/sql-dml.pdf

### ORDER BY

```
[ORDER BY column [ASC|DESC] {, column [ASC|DESC]}]
```

Sorting is the last stage, applied to the finished result. A sort key is either a **column name** or a **column position in the target list**; multiple keys are applied left to right, each with its own `ASC` (default) or `DESC`.

```
-- By department ascending, then salary descending within each department
SELECT dNum, name, salary
FROM   Employee
ORDER BY dNum ASC, salary DESC;

-- Sorting a grouped result by its count, greatest first: the count is
-- the 2nd column of the target list
SELECT   VehiclePurpose, COUNT(*)
FROM     Vehicle
GROUP BY VehiclePurpose
ORDER BY 2 DESC;
```

The positional form is what makes a computed column sortable at all — the aggregate itself belongs to `SELECT` and `HAVING` (sql-aggregation-and-grouping[44]), and `ORDER BY` reaches it by pointing at the column it produced.

Later keys only break ties left by earlier ones, so `ORDER BY c DESC, b ASC` puts *every* row with a larger `c` ahead of `(5, 5, 5)` regardless of its `b`. Where two rows tie on *all* the sort keys, their relative order is unspecified — never rely on it.

## SQL Set Operations

A relation in the relational-model[45] is a *set* of tuples — no duplicates, no order — so the ordinary set operators apply directly to whole query results. These are the one corner of SQL that keeps that set discipline by default, where plain `SELECT` returns a bag (sql-select-statement[46]). Where a join (sql-joins[47]) combines tables into wider **rows**, a set operation stacks two result sets into more **rows** of the same shape. Introduced in 2026-04-06-aggregation-grouping-and-multiple-relation-queries[48]; practised in week8-tutorial-applied-class-7-multiple-relation-queries[49].

[44] courses/infs1200/sql-aggregation-and-grouping.pdf
[45] courses/infs1200/relational-model.pdf
[46] courses/infs1200/sql-select-statement.pdf
[47] courses/infs1200/sql-joins.pdf
[48] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[49] courses/infs1200/week8-tutorial-applied-class-7-multiple-relation-queries.pdf

### The three operators

| Operator | Relational algebra | Result |
|---|---|---|
| `UNION` | $\cup$ | Tuples appearing only in `R1`, only in `R2`, or in both |
| `INTERSECT` | $\cap$ | Tuples appearing in both `R1` and `R2` |
| `EXCEPT` (a.k.a. `MINUS`) | $-$ | Tuples appearing in `R1` but not in `R2` |

```
SELECT ...
UNION [ALL] SELECT ...
[UNION [ALL] SELECT ...]
```

### Union compatibility

Both operands must be **union compatible**: the same number of columns, with pair-wise compatible domains. The result takes its column names from the first `SELECT`, so the second query's names are irrelevant — only its column *order* matters.

```
-- Error: 3 columns UNION 2 columns
SELECT * FROM Table1
UNION
SELECT a, b FROM Table1 WHERE b = 'X';
```

A mismatch is an error, not a silently odd result — which makes it one of the easier SQL mistakes to catch.

### Duplicates: set vs multiset semantics

Each operator eliminates duplicates by default. Appending `ALL` keeps multiset semantics instead. If a tuple occurs $m$ times in $r$ and $n$ times in $s$, it occurs:

| Expression | Occurrences |
|---|---|
| `r UNION ALL s` | $m + n$ |
| `r INTERSECT ALL s` | $\min(m, n)$ |
| `r EXCEPT ALL s` | $\max(0,\ m - n)$ |

Deduplication is often the *reason* to reach for `UNION`: three independent ways of qualifying as a co-founder, unioned together, yield each person once even if they qualify twice.

```
SELECT E.EmployeeID, E.FirstName, E.LastName
FROM   Employee E
JOIN   AdministrativeEmployee AE ON AE.EmployeeID = E.EmployeeID
WHERE  AE.Type = 'LegacyEngineer'
UNION
SELECT E.EmployeeID, E.FirstName, E.LastName
FROM   Employee E
JOIN   RoleGranting RG ON RG.EmployeeID = E.EmployeeID
JOIN   Role R ON R.RoleID = RG.RoleID
WHERE  R.Name = 'FinancialPerformanceOverview';
```

### Algebraic properties

$$A \cup B = B \cup A \qquad (A \cup B) \cup C = A \cup (B \cup C)$$
$$A \cap B = B \cap A \qquad (A \cap B) \cap C = A \cap (B \cap C)$$
$$A - B \neq B - A \qquad (A - B) - C \neq A - (B - C)$$

Union and intersection are commutative and associative; difference is **neither** — so with `EXCEPT` the order of the operands, and the bracketing of a chain of them, is part of the meaning.

### Join or set operation?

| | Join | Set operation |
|---|---|---|
| Combines | Rows from many tables into new **columns** | Result sets of `SELECT`s into new **rows** |
| Column count | May differ between tables | Must match |
| Column types | Can differ | Must be compatible |
| Duplicates | Kept by default | Removed by default |

A set operation earns its place when the two halves need *different* grouping or filtering that cannot be expressed in one pass — for example "granted at least 2 roles" **and** "can edit at least 4 distinct websites", which are two different `COUNT`s over two different groupings:

```sql
SELECT   EmployeeID
FROM     RoleGranting
GROUP BY EmployeeID
HAVING   COUNT(RoleID) >= 2
INTERSECT
SELECT   EmployeeID
FROM     RoleGranting RG
JOIN     Permission P ON P.RoleID = RG.RoleID
WHERE    P.GrantType = 'Edit'
GROUP BY EmployeeID
HAVING   COUNT(DISTINCT WebsiteURI) >= 4;
```

Conversely, a simple `UNION` of two queries differing only in a `WHERE` condition is usually just an `OR` written the long way:

```sql
-- These two are equivalent
SELECT starID FROM Movie M JOIN StarsIn S ON M.movieID = S.movieID
WHERE  year = 1944 OR year = 1974;

SELECT starID FROM Movie M JOIN StarsIn S ON M.movieID = S.movieID WHERE year = 1944
UNION
SELECT starID FROM Movie M JOIN StarsIn S ON M.movieID = S.movieID WHERE year = 1974;
```

Note that `OR` and `UNION` coincide here only because the condition is about a *single* row's year. "Stars who were in a 1944 movie **and** a 1974 movie" cannot be an `AND` over one row — no single row has two years — which is exactly the case `INTERSECT` handles.

### Dialect support, and the two rewrites

The course is precise about what is missing where, and the two claims are different in scope:

| Feature | What the course says |
|---|---|
| `INTERSECT`, `EXCEPT/MINUS` | **Some SQL dialects do not support** intersection and difference — a portability caveat about dialects generally, not about MySQL specifically |
| `FULL OUTER JOIN` | **Not implemented in MySQL** (sql-joins[50]) |

`UNION` is the only one of the three set operators you can rely on everywhere. For the other two the course supplies a rewrite each, and they are different rewrites: an `INTERSECT` becomes a **join**, an `EXCEPT` becomes a **nested query**.

[50] courses/infs1200/sql-joins.pdf

`INTERSECT` becomes a self-join that requires both conditions to hold for the same key:

```sql
-- Stars in a movie from BOTH 1944 and 1974
SELECT starID FROM Movie M JOIN StarsIn S ON M.movieID = S.movieID WHERE year = 1944
INTERSECT
SELECT starID FROM Movie M JOIN StarsIn S ON M.movieID = S.movieID WHERE year = 1974;

-- Rewritten without INTERSECT, joining StarsIn to itself on starID
SELECT DISTINCT S1.starID
FROM   Movie M1
JOIN   StarsIn S1 ON M1.movieID = S1.movieID
JOIN   StarsIn S2 ON S1.starID = S2.starID
JOIN   Movie M2 ON M2.movieID = S2.movieID
WHERE  M1.year = 1944 AND M2.year = 1974;
```

`EXCEPT` queries can likewise always be implemented using nested queries — "in the first result but not the second" becomes a `NOT IN` or `NOT EXISTS` subquery (sql-nested-queries[51]):

```sql
-- Stars in a movie from 1944 but NOT 1974
SELECT starID FROM Movie M JOIN StarsIn S ON M.movieID = S.movieID WHERE year = 1944
EXCEPT
SELECT starID FROM Movie M JOIN StarsIn S ON M.movieID = S.movieID WHERE year = 1974;

-- Rewritten without EXCEPT
SELECT DISTINCT S.starID
FROM   Movie M JOIN StarsIn S ON M.movieID = S.movieID
WHERE  M.year = 1944
  AND  S.starID NOT IN (
      SELECT S2.starID
      FROM   Movie M2 JOIN StarsIn S2 ON M2.movieID = S2.movieID
      WHERE  M2.year = 1974);
```

The rewrite is not just a workaround — `NOT IN`/`NOT EXISTS` is also the building block of the double-negation form of relational-division[52], so it is the pattern worth being fluent in regardless of dialect.

## SQL Views

A **view** is a single table derived from other tables — base tables or other views — and defined by a query rather than by stored rows. Once created it is queried exactly like a table, which makes a view the natural way to name an intermediate result, hide complexity, or expose a restricted slice of the database to a class of users — the **external level** of the three-schema architecture,

[51] courses/infs1200/sql-nested-queries.pdf
[52] courses/infs1200/relational-division.pdf

made concrete (dbms-fundamentals[53]). Introduced in 2026-04-13-nested-queries-views-and-generative-ai[54]; practised in week9-tutorial-applied-class-8-nested-queries-and-views[55].

### Virtual vs materialized

- **Virtual** — the view does not physically exist on disk; it is recomputed each time it is queried, so it is always current. Think of it as a way of *specifying* a table you need to reference frequently, even though no such table exists.
- **Materialized** — the view table is physically created, and must be updated whenever the base tables are updated.

### Creating and querying a view

```
CREATE VIEW <view name> (<column name> {, <column name>}) AS <select statement>;
```

The column-name list is optional, even though the syntax above does not put it in `[...]`: supply it and the view's columns take those names, omit it and they take the names of the target list — which is why a computed column needs an `AS` alias to have a name at all.

```
-- Count, gender and average salary of employees, per department
CREATE VIEW DepEmpStatus AS
SELECT   dNumber, dName, sex, COUNT(*) AS employeeNumber, AVG(salary) AS avgSalary
FROM     Department AS D
JOIN     Employee AS E ON D.dNumber = E.dNum
GROUP BY dNum, sex;

SELECT * FROM DepEmpStatus;
```

A view has a schema like any other relation, and is written the same way in relational-mapping-notation[56] — the aliases given in the target list become its column names. It is listed in the same schema as the base tables it is derived from, directly under them:

```
Department [dNumber, dName, mgrSSN, mgrStartDate]
Employee   [ssn, name, dob, address, sex, salary, mgrSSN, dNum]

VIEW DepEmpStatus [dNumber, dName, sex, employeeNumber, avgSalary]
```

[53] courses/infs1200/dbms-fundamentals.pdf
[54] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf
[55] courses/infs1200/week9-tutorial-applied-class-8-nested-queries-and-views.pdf
[56] courses/infs1200/relational-mapping-notation.pdf

That placement is the point: from the query writer's side a view is one more relation in the schema, and nothing in a `SELECT` distinguishes it from a stored table.

### Querying a view that computes its own columns

Because the view is just a table to the queries that follow, it can be joined, filtered, grouped and aggregated over in turn — including aggregating over columns the view itself computed:

```
CREATE VIEW V AS SELECT a + b AS d, c FROM R;

SELECT   d, SUM(c)
FROM     V
GROUP BY d
HAVING   COUNT(*) <> 1;
```

Worked exercise: given `R` below, which tuples does the second query return?

| R.a | R.b | R.c | V.d = a+b | V.c |
|---|---|---|---|---|
| 1 | 1 | 3 | 2 | 3 |
| 1 | 2 | 3 | 3 | 3 |
| 2 | 1 | 4 | 3 | 4 |
| 2 | 3 | 5 | 5 | 5 |
| 2 | 4 | 1 | 6 | 1 |
| 3 | 2 | 4 | 5 | 4 |
| 3 | 3 | 6 | 6 | 6 |

Grouping `V` by `d` gives four groups — `d = 2` (one row), `d = 3` ($c = 3, 4$), `d = 5` ($c = 5, 4$) and `d = 6` ($c = 1, 6$). `HAVING COUNT(*) <> 1` throws away the singleton group `d = 2`, so the answer is `(3, 7)`, `(5, 9)` and `(6, 7)`. Two things are worth taking from it: the grouping attribute `d` **does not exist in any base table** — it is created by the view and grouped over as if it were stored; and `HAVING` filters *groups*, so `(2, 3)` is excluded even though its underlying row is perfectly ordinary.

### Why views exist

| Benefit | What it buys you |
|---|---|
| **Simplification** | Hides the complexity of the underlying tables behind one name |

| Benefit | What it buys you |
|---|---|
| **Security** | Hides sensitive columns from users who should not see them |
| **Computed columns** | Derived values are computed on the fly, not stored |
| **Logical data independence** | Users and programs querying the view are insulated from changes to the underlying logical schema |

The security case is concrete: a user granted access to `DepEmpStatus` but *not* to `Department`/`Employee` can read departmental statistics without ever seeing an individual employee's address or salary. See database-access-control[57] for where this sits among the course's authorisation mechanisms — granting a privilege on a view rather than on the base table restricts *which columns* and *which rows* an account can read at the same time, which a table-level grant cannot express.

### Views instead of repeated subqueries

The worked example is *"find the name of the department that spends the largest sum of money on salaries"*. A query that needs the same derived table twice ends up computing it twice inside itself, which is **hideously ugly**:

```
-- Hideously ugly: the per-department total is re-derived in two places
SELECT   d.dName
FROM     Department d,
         (SELECT dNum, SUM(salary) AS departmentWage
          FROM   Employee GROUP BY dNum) AS Temp
WHERE    d.dNumber = Temp.dNum
  AND    Temp.departmentWage = (
      SELECT MAX(departmentWage)
      FROM   (SELECT SUM(salary) AS departmentWage
              FROM   Employee GROUP BY dNum) AS Temp2
  );

-- Cleaner: define the intermediate result once
CREATE VIEW Temp AS
SELECT   dNum, SUM(salary) AS departmentWage
FROM     Employee
GROUP BY dNum;

SELECT *
```

[57] courses/infs1200/database-access-control.pdf

```
FROM   Temp
WHERE  departmentWage IN (SELECT MAX(departmentWage) FROM Temp);
```

Both halves of the ugly version compute the *same* grouped aggregate; the view names it once and both the outer query and the `MAX` subquery then read that one name. (The second query selects from `Temp` only, so it returns the winning `dNum`/`departmentWage` pair — recovering `dName` needs one further join back to `Department`, which is the step drawn at the bottom of Figure 1.)

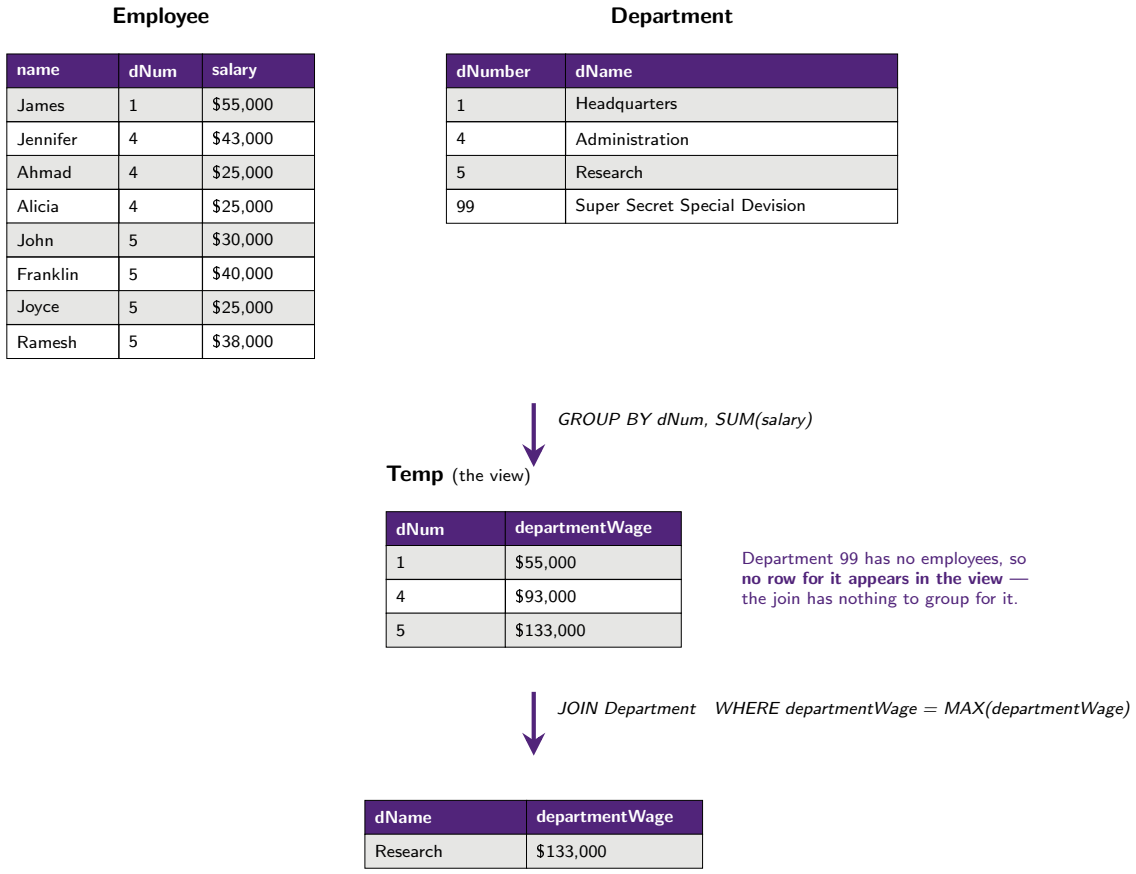


Figure 1: The worked view example, run on sample data: the per-department salary total is computed once into the view, and only then filtered down to the maximum. Department 99, a department with no employees, produces **no row in the view at all**, because the grouped join has nothing to group for it; a question phrased *which department spends the most* never notices, but one phrased *list every department's total* would.

This is the standard way to answer "the group with the largest total" questions: a view holding

one row per group with its aggregate, then a second query selecting the rows matching the extreme (sql-nested-queries[58] covers the `>= MAX(...)` / `>= ALL` idioms that keep ties). Because a view definition is persistent, a script that recreates one should drop it first:

```
DROP VIEW IF EXISTS EarliestRoleGranting;

CREATE VIEW EarliestRoleGranting AS
SELECT   RoleID, MIN(TimeStamp) AS EarliestTimeStamp
FROM     RoleGranting
GROUP BY RoleID;

-- Who received each role first: join the base table back to the view
SELECT E.EmployeeID, E.FirstName, E.LastName, RG.RoleID
FROM   RoleGranting RG
JOIN   EarliestRoleGranting ERG
       ON ERG.RoleID = RG.RoleID AND ERG.EarliestTimeStamp = RG.TimeStamp
JOIN   Employee E ON E.EmployeeID = RG.EmployeeID;
```

That join-back pattern — aggregate in a view, then join the view to the base table on the aggregated value — is how you retrieve *the rows that achieve* an aggregate, which a `GROUP BY` alone cannot give you (sql-aggregation-and-grouping[59]).

### Updating through a view

A view has no rows of its own, so **view updates must occur at the base tables** — and working out which base rows an update to a view row should touch is **ambiguous** and **difficult**. Consider `DepEmpStatus` above: its `avgSalary` column is an average over many employees, so an instruction to change it names no particular salary to change, and its `employeeNumber` column cannot be changed at all without inventing or destroying employees.

DBMSs therefore restrict view updates to **some simple views on single tables**, which are the ones called **updatable views**. The characteristic shape is a view that selects a subset of the rows and columns of one base table, with no join, no grouping and no aggregate, so each view row corresponds to exactly one base row.

That restriction is what makes the security case in the table above practical rather than read-only: the manager of each department can be given a view of the employees of *their* department, and because it is a plain row-and-column slice of one table, they can access **and edit** through it.

[58] courses/infs1200/sql-nested-queries.pdf
[59] courses/infs1200/sql-aggregation-and-grouping.pdf

### Dropping views

```
DROP VIEW [IF EXISTS] view_name [, view_name] ... [RESTRICT | CASCADE];

DROP VIEW DepEmpStatus;
```

Dropping a view does **not** affect any tuples of the underlying relation — only the definition goes away. That asymmetry is the whole point of a virtual view: nothing of the data lives in it.

Views do, however, constrain what can be done to the tables under them: `DROP TABLE … RESTRICT` drops the table *unless* there is a view on it, while `DROP TABLE … CASCADE` drops the table *and* recursively drops any view referencing it (see sql-ddl[60]).

[60] courses/infs1200/sql-ddl.pdf