INFS1200 — Week 9 Notes

Table of contents

Applied Class 8: DQL Nested Queries and Views **1**
- Nested queries 2
- Views 4

Reference material **6**
- Relational Division 6
- Nested Queries (Subqueries) 9
- SQL Views 15

Applied Class 8: DQL Nested Queries and Views

Practice for 2026-04-13-nested-queries-views-and-generative-ai[1]. Same BestTechLtd schema as week8-tutorial-applied-class-7-multiple-relation-queries[2]:

```
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
```

[1] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf
[2] courses/infs1200/week8-tutorial-applied-class-7-multiple-relation-queries.pdf

Nested queries

Question 1

Return the Date of Birth, first and last name of the youngest employee(s).

> **i** Working

```
SELECT E.DOB, E.FirstName, E.LastName
FROM   Employee E
WHERE  DOB >= (SELECT MAX(DOB) FROM Employee E2);
```

`>=` (not `=`) against the max, so **ties** for youngest are all returned, not just one arbitrary row.

Question 2

*Return all information about all employees that have **not** been granted a role from an Administrator with type `'ProductEngineer'`. Restriction: must use a subquery.*

> **i** Working

```
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'
);
```

Question 3

Find the number of employees who share a surname with another employee.

Working

```
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
  );
```

A correlated `EXISTS` check — for each `E1`, is there some *other* employee `E2` with the same last name?

Question 4

Return the `EmployeeId` of the Administrative Employee(s) that have granted at least as many roles to employees as Mehdi Rahman. Assume only one Mehdi Rahman exists, and include Mehdi Rahman in the result.

Working

```
SELECT     RG.AdministrationID
FROM       AdministrativeEmployee AE
JOIN       RoleGranting RG ON AE.EmployeeID = RG.AdministrationID
GROUP BY   RG.AdministrationID
HAVING     COUNT(DISTINCT RG.EmployeeID, RG.RoleID) >= (
    SELECT   COUNT(DISTINCT RG.EmployeeID, RG.RoleID)
    FROM     RoleGranting RG
    JOIN     Employee E ON RG.AdministrationID = E.EmployeeID
    WHERE    E.FirstName = 'Mehdi' AND E.LastName = 'Rahman'
    GROUP BY RG.AdministrationID
);
```

Question 5

Find the first name and last name of all employees that have access to at least all the website URIs that Employee `E0007` has access to.

Working

```
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
      )
);
```

This is a **relational division via double negation** — "there is no website E0007 can access that E cannot" — see 2026-04-13-nested-queries-views-and-generative-ai[3] for the general pattern.

Views

Question 6

*Return the EmployeeID, FirstName, LastName and RoleId of all employees who were the **first** to receive a given RoleID.*

[3] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf

Working

```
DROP VIEW IF EXISTS EarliestRoleGranting;

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

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;
```

Question 7

Return the EmployeeID of the Administrator(s) that have granted the most permissions of any other Administrator. Restriction: must use one or more views.

Working

```
DROP VIEW IF EXISTS AdminPermissionsCount;

CREATE VIEW AdminPermissionsCount AS
SELECT      AE.EmployeeID,
            COUNT(WebsiteURI) AS NumOfPermissionsGranted
FROM        AdministrativeEmployee AE
LEFT JOIN   RoleGranting RG ON RG.AdministrationID = AE.EmployeeID
LEFT JOIN   Permission P ON P.RoleID = RG.RoleID
GROUP BY    AE.EmployeeID;

SELECT *
FROM   AdminPermissionsCount
WHERE  NumOfPermissionsGranted >= (
    SELECT MAX(APC2.NumOfPermissionsGranted)
    FROM   AdminPermissionsCount APC2
);
```

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[4]; applied in week9-tutorial-applied-class-8-nested-queries-and-views[5].

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 *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}`:

[4] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf
[5] courses/infs1200/week9-tutorial-applied-class-8-nested-queries-and-views.pdf

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[6], sql-nested-queries[7]).

Via double negation

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

```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".

[6] courses/infs1200/sql-aggregation-and-grouping.pdf
[7] courses/infs1200/sql-nested-queries.pdf

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.

```
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
      )
);
```

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[8]; practised in week9-tutorial-applied-class-8-nested-queries-and-views[9].

The motivating case is the aggregate that cannot go in a `WHERE` clause (see sql-select-statement[10] 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
```

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

```sql
SELECT name, salary
FROM   Employee
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.

```sql
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.

```sql
-- 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[11].

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,
```

[11] 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[12]) 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);
```

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

Joins or subqueries?

Many nested queries can be rewritten as a plain join (sql-joins[13]) — 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[14]).

"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[15].

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, made concrete (dbms-fundamentals[16]). Introduced in 2026-04-13-nested-queries-views-and-generative-ai[17]; practised in week9-tutorial-applied-class-8-nested-queries-and-views[18].

[13] courses/infs1200/sql-joins.pdf
[14] courses/infs1200/sql-set-operations.pdf
[15] courses/infs1200/relational-division.pdf
[16] courses/infs1200/dbms-fundamentals.pdf
[17] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf
[18] courses/infs1200/week9-tutorial-applied-class-8-nested-queries-and-views.pdf

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[19] — 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]
```

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.

[19] courses/infs1200/relational-mapping-notation.pdf

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
Security	Hides sensitive columns from users who should not see them
Computed columns	Derived values are computed on the fly, not stored

Benefit	What it buys you
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[20] 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 *
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

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

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.)

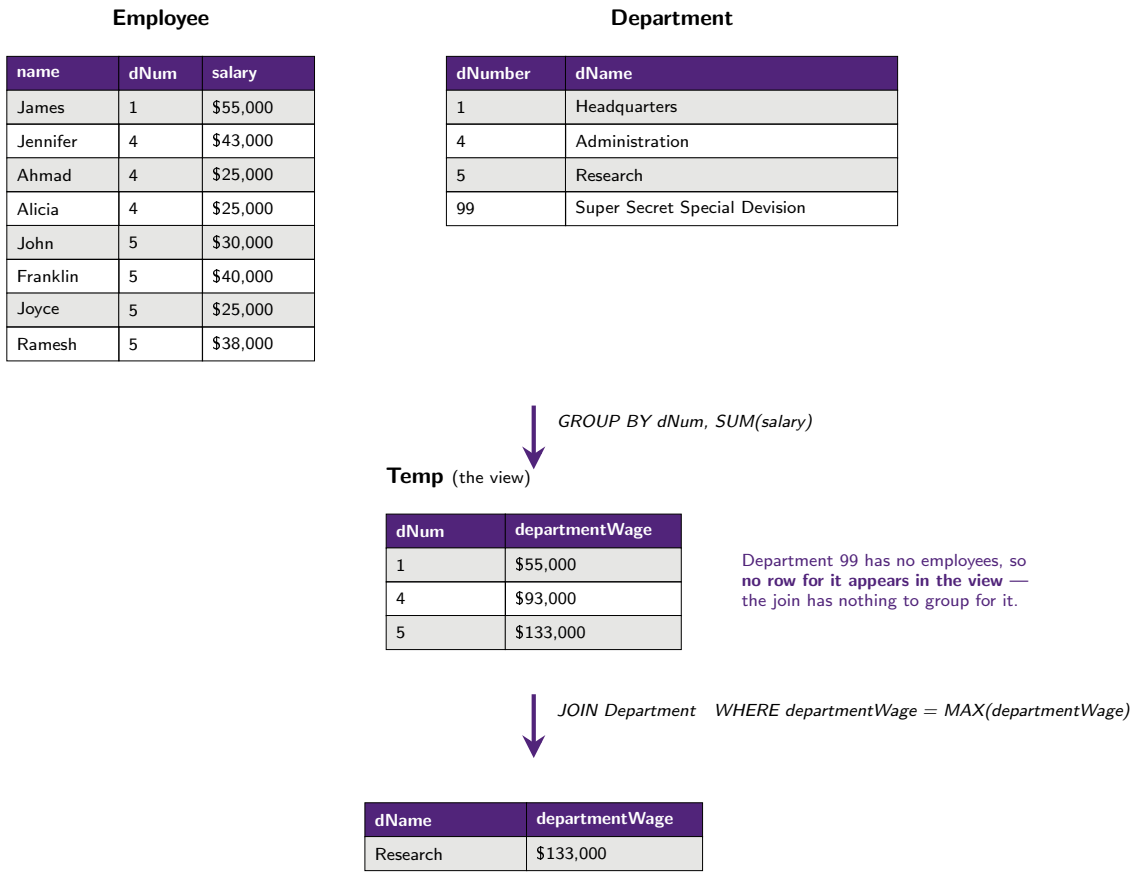

Figure 1: The worked view example, run on sample data: the per-department salary total is computed once into the view, and only then filtered down to the maximum. Department 99, a department with no employees, produces **no row in the view at all**, because the grouped join has nothing to group for it; 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[21] covers the `>= MAX(...)` / `>= ALL` idioms that keep ties). Because a view definition is persistent, a script that recreates one should drop it first:

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

```sql
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[22]).

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.

Dropping views

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

DROP VIEW DepEmpStatus;
```

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

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[23]).

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