INFS1200 — Week 8 Notes

Table of contents

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

[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

- 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.

```sql
-- 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 ...)`
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").

```sql
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.

```sql
-- 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`:

```sql
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:

```sql
-- 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

```
-- 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

```
-- 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.

[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

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';
```

Question 2

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

[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

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.

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.

[8] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf