INFS1200 — Week 7 Notes

Table of contents

Aggregation, Grouping and Multiple Relation Queries

Module 3, part 2. Continues 2026-03-30-basic-sql-ddl-and-dml[1], using the same Company (`Department`/`Employee`/`Sale`/`Item`/`Customer`) and Movie (`Movie`/`StarsIn`/`MovieStar`) schemas.

[1] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf

Today's outline

- Aggregation and grouping (`COUNT`/`SUM`/`AVG`/`MAX`/`MIN`, `GROUP BY`, `HAVING`)
- Multiple relation queries — renaming, joins, set operations

Aggregation

Aggregates are functions that produce a **summary value** from a set of tuples:

- **`COUNT`** — counts the number of tuples returned.
- **`SUM`/`AVG`** — sum/average of a set of numeric values.
- **`MAX`/`MIN`** — maximum/minimum value from a set with a total ordering (the domain doesn't have to be numeric — e.g. dates, strings).

Aggregation functions can be used in the `SELECT` clause and the `HAVING` clause (below), with or without `DISTINCT`.

```sql
-- Total and average salary of all employees
SELECT AVG(salary), SUM(salary)
FROM   Employee;

-- Total number of employees in department 5
SELECT COUNT(*)
FROM   Employee
WHERE  dNum = 5;

-- Distinct salary count of employees in department 5
SELECT COUNT(DISTINCT salary)
FROM   Employee
WHERE  dNum = 5;
```

Grouping (GROUP BY)

An aggregate can apply to the *whole* table, but is often needed **per group** of rows — e.g. "total employees per department", "average salary per department". `GROUP BY` provides this.

```sql
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]}];
```

Rule: any attribute in the `SELECT` clause must either appear in `GROUP BY`, or be wrapped in an aggregation function.

```sql
-- Total employees (no grouping)
SELECT COUNT(*) FROM Employee;                    -- 8

-- Total employees per department
SELECT dNum, COUNT(*)
FROM   Employee
GROUP BY dNum;
--  dNum | COUNT(*)
--     1 |    1
--     4 |    3
--     5 |    4

-- Total employees earning > 40000, per department
SELECT dNum, COUNT(*)
FROM   Employee
WHERE  salary > 40000
GROUP BY dNum;

-- Average salary per department, per gender
SELECT dNum, sex, AVG(salary)
FROM   Employee
GROUP BY dNum, sex;
```

`WHERE` filters *rows* before grouping; `GROUP BY` then buckets the remaining rows.

Question 9 — INSERT and GROUP BY

Given `Students [id, fName, lName, degree]`, what does `INSERT INTO DegreeStatistics (degree, num) SELECT degree, COUNT(*) FROM Students GROUP BY degree` insert?

A. `(Arts, 2), (CompSci, 3)` B. `(Math, 1), (Arts, 2), (CompSci, 3)` C. `(Math, 1), (Arts, 8), (CompSci, 12)` D. None of the above

> **Solution**
>
> **B.** Every distinct `degree` value gets its own group and count — missing `Math` from option A is the trap (there's no `WHERE` clause excluding it). C looks like `SUM(id)` was used instead of `COUNT(*)`.

HAVING — conditions on groups

`HAVING` (following `GROUP BY`) filters *groups*, the way `WHERE` filters rows — and unlike `WHERE`, `HAVING` **can** include aggregates. Any attribute in `HAVING` must appear in `GROUP BY` or be aggregated.

```sql
-- Total employees, for departments with more than 2 employees
SELECT   dNum, COUNT(*)
FROM     Employee
GROUP BY dNum
HAVING   COUNT(*) > 2;

-- Departments with more than 2 employees earning > 20000
SELECT   dNum, COUNT(*)
FROM     Employee
WHERE    salary > 20000
GROUP BY dNum
HAVING   COUNT(*) > 2;
```

In the second example, `WHERE` and `HAVING` both operate on "employees earning > 20000" — contrast this with the correlated-subquery version of the same idea, in 2026-04-13-nested-queries-views-and-generative-ai[2], where `HAVING` checks a *different* condition across *all* employees via a subquery.

Question 10 — HAVING clause

For `SELECT team, COUNT(*) AS games_played, SUM(runsFor) AS total_runs FROM Scores GROUP BY team HAVING SUM(runsFor) > 10`*, what is the* `HAVING` *clause doing?*

A. Removes rows where `runsFor <= 10` before grouping. B. Filters rows before aggregation. C. Limits which teams are included, based on their total runs scored. D. Replaces `WHERE` for simple conditions. E. Counts opponents that allowed more than 10 runs.

> **Solution**
>
> **C.** `HAVING` filters *groups* (here, teams) by an aggregate condition computed *after* grouping — it doesn't touch individual rows before grouping (that's what `WHERE` would do).

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

Question 11 — HAVING clause application

Given `Employee [id, dNum, sex, salary]` *with rows* `(1,5,M,30000), (2,5,M,40000), (3,5,F,25000), (4,5,M,38000), (5,1,M,55000),` *for* `SELECT dNum, sex, COUNT(*) FROM Employee GROUP BY dNum, sex HAVING …`*, which* `HAVING` *clause produces a* ***different*** *result from the others?*

A. `HAVING COUNT(*) < 2` B. `HAVING AVG(salary) = 25000` C. `HAVING MAX(salary) <= MIN(salary)` D. `HAVING SUM(salary) < 60000`

> **Solution**
>
> **B.** Groups are `(5,M)` [3 rows: 30000/40000/38000], `(5,F)` [1 row: 25000], `(1,M)` [1 row: 55000]. A, C and D all select the same two singleton groups `(5,F,1)` and `(1,M,1)` (both have `COUNT(*)=1<2`; both trivially satisfy `MAX<=MIN` since there's one value; both have `SUM<60000`). B selects only `(5,F,1)`, since its average is exactly 25000 but `(1,M)`'s average is 55000 — a different result set.

Multiple relation queries

Three tools for combining relations: **renaming**, **joins**, and **set operations** (union/intersect/difference).

Renaming

Two ways to rename: qualifying attribute names (`table.attribute`), or declaring an alias (`AS`). Renaming removes ambiguity and enables self-joins.

```sql
-- Names/salaries/salaries+17% for dept 4
SELECT E.name, salary, 1.17 * salary AS 'includingSuper'
FROM   Employee E
WHERE  dNum = 4;
```

Cartesian product

`R1 × R2`: every row of `R1` combined with every row of `R2`; the result schema is the concatenation of both schemas. If `|R1| = m` and `|R2| = n`, then `|R1 × R2| = m * n`.

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

Every `MovieStar` row is paired with *every* `StarsIn` row — almost always **not** what you want (compare with a join, below).

Equi-join and joins generally

An **equi-join** combines tuples from two relations that agree on some pair of attributes (a *join condition* using only `=`). A **join** in general combines related tuples into a single result relation:

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

-- equivalently, using WHERE (Cartesian product + 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
SELECT E.name, D.dName
FROM   Department AS D, Employee AS E
WHERE  D.mgrSSN = E.ssn;

-- equivalent, using JOIN
SELECT E.name, D.dName
FROM   Department AS D
JOIN   Employee AS E ON D.mgrSSN = E.ssn;
```

`JOIN/ON` is generally preferred over the `WHERE`-based Cartesian product form because it can be more efficiently optimised by the DBMS — but for this course, they're treated as equivalent.

```
-- IDs and names of all stars who've been in a movie
SELECT DISTINCT S.starID, name
FROM   StarsIn S
JOIN   MovieStar MS ON S.starID = MS.starID;

-- IDs, names and characters of stars in the movie with movieID 1
SELECT S.starID, name, role
FROM   StarsIn S
JOIN   MovieStar MS ON S.starID = MS.starID
WHERE  S.movieID = 1;

-- Multi-join: 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';
```

Question 12 — SQL joins

Which query correctly returns employee names and the name of the department they work in?

A. `SELECT e.name, d.dName FROM Employee e, Department d WHERE e.mgrSSN = d.mgrSSN;` B. `SELECT e.name, d.dName FROM Employee e JOIN Department d ON e.ssn = d.mgrSSN;` C. `SELECT e.name, d.dName FROM Department d JOIN Employee e ON d.mgrSSN = e.mgrSSN;` D. `SELECT e.name, d.dName FROM Employee e JOIN Department d ON e.dNum = d.dNumber;`

> **Solution**
>
> **D.** A joins on shared `mgrSSN` values (matching managers to managers, not employees to *their* department). B matches an employee's `ssn` to a department's `mgrSSN` — that's "departments this employee manages", not "the department they work in". C is the same mismatch as B, reversed. D correctly joins each employee's `dNum` to the department's `dNumber` — "the department they work in".

Self-joins (recursive relationships)

```
-- 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;
```

Question 13 — Grouping with join

Given `Flight [FlightNo, Origin, Destination]`, for

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

(read as: "how many ways can you connect from `F1.Origin` to `F2.Destination` via one stopover?") which is in the result?

A. `(Brisbane, Melbourne, 2)` B. `(Perth, Sydney, 6)` C. `(Perth, Melbourne, 1)` D. All of the above E. None of the above

Solution

D — all of the above. The self-join pairs every `F1` flight with every `F2` flight departing from where `F1` lands, then groups by `(F1.Origin, F2.Destination)` and counts the pairs. Working through all nine flights (2× Brisbane→Sydney, 1× Sydney→Melbourne, 2× Melbourne→Perth, 3× Perth→Brisbane, 1× Perth→Sydney) gives:

F1.Origin	F2.Destination	COUNT(*)	Why
Brisbane	Melbourne	2	2 Brisbane→Sydney flights × 1 Sydney→Melbourne flight
Sydney	Perth	2	1 Sydney→Melbourne flight × 2 Melbourne→Perth flights
Melbourne	Brisbane	6	2 Melbourne→Perth flights × 3 Perth→Brisbane flights
Melbourne	Sydney	2	2 Melbourne→Perth flights × 1 Perth→Sydney flight
Perth	Sydney	6	3 Perth→Brisbane flights × 2 Brisbane→Sydney flights
Perth	Melbourne	1	1 Perth→Sydney flight × 1 Sydney→Melbourne flight

`(Brisbane, Melbourne, 2)`, `(Perth, Sydney, 6)` and `(Perth, Melbourne, 1)` are all genuinely in the result — so A, B and C are all correct.

Theta-join

The most general join type — the join condition can use any of `{=, , <, , >, }`, not just `=`.

Question 14 — Theta join

Given `Student [id, sName, age]`, *for* `SELECT DISTINCT S1.sName, S1.age FROM Student S1 JOIN Student S2 ON S1.age > S2.age`, *what does this return?*

A. Name/age of one of the oldest student(s). B. Name/age of all of the oldest student(s). C. Name/age of all of the youngest student(s). D. Name/age of all students older than the youngest student(s). E. None of the above.

> **i** Solution
>
> **D.** `S1.age > S2.age` only has a match for `S1` if *some* student (`S2`) is younger — i.e. `S1` is not the/a youngest student. So the result is every student who is older than at least one other student: everyone except the youngest.

Inner and outer joins

- **Inner join** (default, just `JOIN`): a tuple appears in the result only if matching tuples exist in *both* relations.
- **Outer join**: includes the inner join result, *plus* unmatched rows from one or both tables:
  - **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.**

```sql
-- Misses departments without a manager:
SELECT   D.dName, E.name
FROM     Department AS D
JOIN     Employee AS E ON D.mgrSSN = E.ssn;

-- Includes them (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;
```

Set operations

A relation is a *set* of tuples (no duplicates, no order) — set operators apply directly:

- **UNION** — tuples in either or both relations.
- **INTERSECT** — tuples in both.
- **EXCEPT** (a.k.a. `MINUS`) — tuples in the first but not the second.

Union compatibility is required: the same number of columns, with pair-wise compatible domains.

By default each of these eliminates duplicates; append `ALL` (`UNION ALL`, `INTERSECT ALL`, `EXCEPT ALL`) to keep multiset semantics. If a tuple occurs `m` times in `r` and `n` times in `s`: it occurs `m + n` times in `r UNION ALL s`, `min(m, n)` times in `r INTERSECT ALL s`, and `max(0, m - n)` times in `r EXCEPT ALL s`.

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

```
-- Stars in a movie from 1944 or 1974 - via OR:
SELECT starID FROM Movie M JOIN StarsIn S ON M.movieID = S.movieID
WHERE  year = 1944 OR year = 1974;

-- ...or equivalently via UNION:
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;
```

`INTERSECT` is part of the SQL standard but **not implemented in MySQL** — it can be rewritten with a self-join:

```
-- 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, using a self-join 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;
```

```sql
-- Stars in a movie from 1944 but NOT 1974 (EXCEPT):
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;
```

`EXCEPT` queries can also always be rewritten as nested queries (next lecture).

Properties of set operators

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

Union and intersection are commutative and associative; difference is **neither**.

JOIN vs set operations

	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

Question 15 — UNION query

Given `Table [A, B, C]` with rows `(1,X,11)`, `(2,Y,12)`, `(3,Y,13)`, what does `SELECT * FROM Table UNION SELECT a, b FROM Table WHERE b = 'X' OR c = '13'` produce?

> **Solution**
>
> **D — none of the above; the query raises an error.** `SELECT *` returns 3 columns (`A, B, C`) but `SELECT a, b` returns only 2 — the two halves of the `UNION` are **not union-compatible** (mismatched column counts), so none of the listed result tables can occur.

Question 16 — UNION query (two tables)

Given `Table1 [A,B,C] = (1,X,11), (2,Y,12), (3,Y,13)` *and* `Table2 [D,E,C] = (3,X,11), (4,Y,12), (3,Y,13)`*, what does* `SELECT * FROM Table1 UNION SELECT d, e, c FROM Table2 WHERE E = 'X' OR C = 13` *produce?*

> **Solution**
>
> **C.** `Table2` rows matching `E='X' OR C=13` are `(3,X,11)` and `(3,Y,13)`; renamed to `(A,B,C)` these are `(3,X,11)` and `(3,Y,13)`. Unioned with all of `Table1` — `(1,X,11)`, `(2,Y,12)`, `(3,Y,13)` — and removing the duplicate `(3,Y,13)`, the result is `(1,X,11)`, `(2,Y,12)`, `(3,Y,13)`, `(3,X,11)`: four rows, all of `Table1` plus the one genuinely new row `(3,X,11)`. (A is missing a row from `Table1`; B fails to remove the duplicate `(3,Y,13)`.)

Summary

You can now aggregate and group data, and query across multiple relations using renaming, joins (equi/theta/inner/outer), and set operations. Next lecture: nested queries, views, and Generative AI & SQL. See week7-tutorial-applied-class-6-aggregation-and-grouping[3], week7-tutorial-case-study-5-easydrive-insurance[4] and week8-tutorial-applied-class-7-multiple-relation-queries[5] for practice.

Applied Class 6: DQL Aggregation and Grouping

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

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

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

Question 1

Return the youngest Employee's Date of Birth.

> **Working**

```
SELECT MAX(DOB)
FROM   Employee;
```

The *youngest* employee has the *largest* (most recent) `DOB` — a common trap is reaching for `MIN` by reflex.

Question 2

For each website URI, find the number of roles that can access it. Two columns: the website URI, and the number of roles.

> **Working**

```
SELECT   WebsiteURI, COUNT(DISTINCT RoleID)
FROM     Permission
GROUP BY WebsiteURI;
```

Question 3

Return the `EmployeeId` of all employees who have been granted more than one role.

> **Working**

```
SELECT   EmployeeID
FROM     RoleGranting
GROUP BY EmployeeID
HAVING   COUNT(RoleID) > 1;
```

Question 4

Return the highest `AdministrativeEmployee` *level, the lowest level, and the difference between both, aliased* `DifferenceInAdministrativeEmployeeLevels`*.*

i Working

```
SELECT MAX(Level), MIN(Level),
       (MAX(Level) - MIN(Level)) AS DifferenceInAdministrativeEmployeeLevels
FROM   AdministrativeEmployee;
```

Question 5 — Discuss

(1) What do you observe using an aggregation function without `GROUP BY`*? Why? (2) What do you observe aggregating (*`GROUP BY`*) over a primary key or unique field? Why?*

i Working

1. **Observation**: the aggregate operates over the *entire* table and returns a single row/value. **Reason**: with no `GROUP BY`, the whole result set is treated as one big group.
2. **Observation**: the number of rows returned equals the number of rows in the table, and most aggregate functions return values identical to the ungrouped row. **Reason**: a primary key/unique field is unique and non-null per row, so every "group" contains exactly one tuple.

Question 6

Return the average salary of Employees born after 1 January 1990.

i Working

```
SELECT AVG(Salary)
FROM   Employee
WHERE  DOB > '1990-01-01';
```

Question 7

Return the `roleID` of all roles which have at least two associated website URIs with a `grantType` of `'Edit'`.

Working

```sql
SELECT   RoleId
FROM     Permission
WHERE    GrantType = 'Edit'
GROUP BY RoleId
HAVING   COUNT(WebsiteURI) >= 2;
```

Question 8 (Challenge)

Return all last names that are shared by multiple employees.

Working

```sql
SELECT   DISTINCT LastName
FROM     Employee
WHERE    LastName IS NOT NULL
GROUP BY LastName
HAVING   COUNT(FirstName) > 1;
```

Case Study 5: EasyDrive Insurance (Queries)

Practice for 2026-04-06-aggregation-grouping-and-multiple-relation-queries[8]. An analyst at EasyDrive Insurance needs several queries answered against their production data (loaded from a synthetic-data dump), but struggles to write them — this case study writes them on the analyst's behalf.

Schema

Note this schema has evolved slightly since week6-tutorial-case-study-4-easydrive-insurance[9] — `VehicleType` has been split into `VehicleCodeMapping`, `VehicleValue`

[8] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[9] courses/infs1200/week6-tutorial-case-study-4-easydrive-insurance.pdf

and `VehicleExcessRange` (as transcribed directly from the source materials — the two case studies' schemas are not perfectly identical).

```
Customer           [CustomerID, Name, DateOfBirth, Email, Occupation, AddressID]
Address            [AddressID, StreetName, Number, Suburb, Postcode, State, Country]
Vehicle            [VehicleID, VehicleCode, VehiclePurpose, EstYearlyKm]
VehicleCodeMapping [VehicleCode, Make, Model, Year]
VehicleValue       [VehicleCode, MarketValue]
VehicleExcessRange [VehicleCode, MinimumExcess, MaximumExcess]
Policy             [PolicyID, CustomerID, VehicleID, PolicyStartYear, PolicyPurchaseDate, Ex

Customer.AddressID references Address.AddressID
Policy.VehicleID references Vehicle.VehicleID
Policy.CustomerID references Customer.CustomerID
Vehicle.VehicleCode references VehicleCodeMapping.VehicleCode
VehicleExcessRange.VehicleCode references VehicleCodeMapping.VehicleCode
VehicleValue.VehicleCode references VehicleCodeMapping.VehicleCode
```

Task 1a

Return the number of vehicles used for every `VehiclePurpose`, ordered greatest to least.

i Working

```sql
SELECT   VehiclePurpose, COUNT(*)
FROM     Vehicle
GROUP BY VehiclePurpose
ORDER BY COUNT(*) DESC;
```

Task 2

Which `CustomerID`(s) have at least 2 Policies?

i Working

```sql
SELECT   CustomerID
FROM     Policy
GROUP BY CustomerID
HAVING   COUNT(*) >= 2;
```

Task 3

The marketing team wants to find the most common suburb of our customers (return the suburb name and street count, two columns). If there are ties, return all of them, in ascending alphabetical order of suburb name. Assume different suburbs never share a name.

i Working

```sql
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;
```

`>= ALL (...)` here means "greater than or equal to every group's count" — i.e. the maximum. See 2026-04-13-nested-queries-views-and-generative-ai[10] for more on `ALL`.

Task 4a

A possible fraud was detected — identify all street names containing `"et"` with at least two customers living on that street.

```sql
SELECT   StreetName
FROM     Address
WHERE    StreetName LIKE '%et%'
GROUP BY Postcode, StreetName
HAVING   COUNT(*) >= 2;
```

i Working

This query is **incorrect** as written: grouping by `(Postcode, StreetName)` doesn't account for whether two customers actually live at the *same* address (e.g. a shared house sharing one `AddressID`) versus merely the same street. Correctly answering this requires joining `Address` to `Customer` and reasoning about which customers share an address — which needs a multi-relation query, covered next week. This is a useful edge case to notice: grouping by the wrong combination of columns can silently produce a plausible-looking

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

but wrong count.

Task 4b

Duplicate counting can happen for many different reasons — check whether the query above counts each street name correctly.

Working

Same underlying issue as 4a — grouping by `StreetName` alone (dropping `Postcode` from the `GROUP BY`) doesn't fix the shared-address double-counting problem either; both versions can overcount when multiple customers share one `AddressID`.