INFS1200 — Week 7 Notes

Table of contents

Aggregation, Grouping and Multiple Relation Queries **2**
- Today's outline . . . 2
- Aggregation . . . 2
- Grouping (GROUP BY) . . . 3
- Question 9 — INSERT and GROUP BY . . . 3
- Question 10 — HAVING clause . . . 4
- Question 11 — HAVING clause application . . . 5
- Multiple relation queries . . . 5
- Question 12 — SQL joins . . . 7
- Question 13 — Grouping with join . . . 8
- Question 14 — Theta join . . . 9
- Set operations . . . 10
- Question 15 — UNION query . . . 12
- Question 16 — UNION query (two tables) . . . 12
- Summary . . . 12

Applied Class 6: DQL Aggregation and Grouping **13**

Case Study 5: EasyDrive Insurance (Queries) **16**
- Schema . . . 16

Reference material **19**
- Aggregation and Grouping . . . 19
- SQL Joins . . . 23
- The SELECT Statement and Its Clause Order . . . 27
- SQL Set Operations . . . 32

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.

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;
```

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

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.

```
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.

```
-- 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.

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

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).

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.

```
-- 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`.

```
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;
```

```sql
-- 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;`

i 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)

```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;
```

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.**

```
-- 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:

```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, 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?*

> **i** 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?*

> **i** 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.

[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

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
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**
>
> ```sql
> 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.

[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

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`.

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?

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.

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

```
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` 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
```

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

Task 1a

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

Working

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

Task 2

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

Working

```
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.

Working

```
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;
```

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 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`.

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

Reference material

Aggregation and Grouping

An **aggregate function** produces a single summary value from a *set* of tuples, and `GROUP BY` is what lets that summary be computed per bucket of rows rather than over the whole table. Together they turn a query from "which rows?" into "how many / how much / what is the extreme?". Introduced in 2026-04-06-aggregation-grouping-and-multiple-relation-queries[11]; practised in week7-tutorial-applied-class-6-aggregation-and-grouping[12] and week7-tutorial-case-study-5-easydrive-insurance[13].

The aggregate functions

Function	Returns
`COUNT`	The number of tuples
`SUM`	The sum of a set of numeric values
`AVG`	The average of a set of numeric values
`MAX` / `MIN`	The largest/smallest value of any domain with a **total ordering** — not just numbers

`MAX`/`MIN` working on any ordered domain is the source of a recurring trap: dates order chronologically, so the **youngest** employee is the one with the **largest** (most recent) date of birth.

```
SELECT MAX(DOB) FROM Employee;   -- youngest, not oldest
```

Any aggregate can be applied with or without `DISTINCT`, which matters most for `COUNT`: `COUNT(*)` counts tuples, while `COUNT(DISTINCT attr)` counts how many *different* values of `attr` those tuples hold.

```
SELECT COUNT(*)               FROM Employee WHERE dNum = 5;
SELECT COUNT(DISTINCT salary) FROM Employee WHERE dNum = 5;

-- How many distinct roles reach each website - not how many
-- Permission rows mention it
SELECT   WebsiteURI, COUNT(DISTINCT RoleID)
FROM     Permission
GROUP BY WebsiteURI;
```

[11] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[12] courses/infs1200/week7-tutorial-applied-class-6-aggregation-and-grouping.pdf
[13] courses/infs1200/week7-tutorial-case-study-5-easydrive-insurance.pdf

Aggregation functions may be used in **the `SELECT` clause and the `HAVING` clause** — and **never in `WHERE`**, which runs before any grouping exists (see sql-select-statement[14]). To filter rows against an aggregate, compute it in a subquery: sql-nested-queries[15]. To *sort* by one, name the target-list position the aggregate occupies rather than repeating the aggregate in `ORDER BY`.

Aggregating without GROUP BY

With no `GROUP BY`, the entire result set is treated as **one group**: the aggregate spans the whole table and the query returns a single row.

```
SELECT AVG(salary), SUM(salary) FROM Employee;

-- Several aggregates and an expression over them, in one row
SELECT MAX(Level), MIN(Level),
       (MAX(Level) - MIN(Level)) AS DifferenceInAdministrativeEmployeeLevels
FROM   AdministrativeEmployee;
```

GROUP BY

`GROUP BY` and `HAVING` are stages 3 and 4 of the clause pipeline in sql-select-statement[16], which is where their ordering rules come from. `WHERE` filters *rows* first; `GROUP BY` then buckets the survivors, one group per distinct combination of the grouping attributes, and one output row per group.

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

-- Only employees earning > 40000 are counted
SELECT   dNum, COUNT(*)
FROM     Employee
WHERE    salary > 40000
GROUP BY dNum;
```

[14] courses/infs1200/sql-select-statement.pdf
[15] courses/infs1200/sql-nested-queries.pdf
[16] courses/infs1200/sql-select-statement.pdf

```sql
-- Grouping by two attributes: one group per (department, gender) pair
SELECT   dNum, sex, AVG(salary)
FROM     Employee
GROUP BY dNum, sex;
```

The rule: when `GROUP BY` is used, any attribute appearing in the `SELECT` clause must either also appear in the `GROUP BY` clause or be inside an aggregation function. There is no other option, because a group has exactly one value for each grouping attribute but many values for everything else — SQL would not know which one to print.

A useful degenerate case: grouping by a primary key or other unique field puts exactly one tuple in every group, so the query returns as many rows as the table has, and each aggregate just echoes that row's own value.

HAVING — conditions on groups

`HAVING` filters *groups* the way `WHERE` filters rows, and unlike `WHERE` it **can** contain aggregates. The same attribute rule applies in its own words: any attribute appearing in the `HAVING` clause must either appear in the `GROUP BY` clause or be inside an aggregation function.

```sql
-- 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 query `WHERE` and `HAVING` both talk about the same filtered population ("employees earning over 20000"). Making `HAVING` test a *different* population — e.g. the department's total headcount, not just its well-paid staff — needs a correlated subquery, see sql-nested-queries[17].

Idioms worth recognising:

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

```sql
-- "at least two X per group"
SELECT   RoleId FROM Permission
WHERE    GrantType = 'Edit'
GROUP BY RoleId
HAVING   COUNT(WebsiteURI) >= 2;

-- "values shared by more than one row"
SELECT   DISTINCT LastName FROM Employee
WHERE    LastName IS NOT NULL
GROUP BY LastName
HAVING   COUNT(FirstName) > 1;
```

The `IS NOT NULL` guard there isn't decoration. "Unknown" is not a shared surname, and the course never settles what a null does once it reaches a `GROUP BY` bucket or an aggregate — see sql-nulls[18] for exactly what Module 3 does and doesn't specify. Excluding nulls explicitly is the habit the material teaches, precisely so the answer doesn't depend on a rule you were never given.

Comparing a group's count against a count computed elsewhere is the basis of the counting form of relational-division[19] — "this X is related to as many Ys as there are Ys, so it is related to all of them".

Note that different `HAVING` conditions can coincide on a given data set without being equivalent: over singleton groups, `COUNT(*) < 2`, `MAX(salary) <= MIN(salary)` and `SUM(salary) < 60000` may all select exactly the same groups, while `AVG(salary) = 25000` picks out a different set. Reason about what each condition *means* per group, not about which rows it happens to return once.

Choosing the grouping attributes carefully

The `GROUP BY` list defines what "one thing" means for the whole query, so getting it wrong produces a plausible-looking but wrong count rather than an error. Counting customers per street by grouping `Address` on `(Postcode, StreetName)` overcounts if several customers share one `AddressID` — the question was about *people per street*, but the grouping was over *address rows*. Ask what a single output row is supposed to represent, and check that the grouping produces exactly one row per such thing; often the honest answer requires joining another table in first (sql-joins[20]).

[18] courses/infs1200/sql-nulls.pdf
[19] courses/infs1200/relational-division.pdf
[20] courses/infs1200/sql-joins.pdf

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[21] are precisely the paths a join follows. Introduced in 2026-04-06-aggregation-grouping-and-multiple-relation-queries[22]; practised in week8-tutorial-applied-class-7-multiple-relation-queries[23].

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

[21] courses/infs1200/er-to-relational-mapping.pdf
[22] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[23] 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[24]): 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;
```

[24] 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[25].

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.

[25] 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[26]). 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[27] stack result sets as new *rows* rather than new columns, and sql-nested-queries[28] 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[29] discusses under denormalisation, and the operation decomposition[30] must keep lossless.

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].

The full clause skeleton

[26] courses/infs1200/foreign-keys.pdf
[27] courses/infs1200/sql-set-operations.pdf
[28] courses/infs1200/sql-nested-queries.pdf
[29] courses/infs1200/normal-forms.pdf
[30] courses/infs1200/decomposition.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

```
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]).
- **`HAVING` may use aggregates, `WHERE` may not**, because `HAVING` runs at stage 4, after the groups exist.

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

- **`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;
```

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.

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

Expressions and aliases

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

```sql
-- 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]
`+ - * /`, 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`

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

```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.

ORDER BY

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

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

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].

The three operators

[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

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.

```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';
```

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:

```
-- 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**.

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

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

```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.

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