Aggregation and Grouping

Table of contents

The aggregate functions 1
Aggregating without GROUP BY 2
GROUP BY 2
HAVING — conditions on groups 3
Choosing the grouping attributes carefully 4

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[1]; practised in week7-tutorial-applied-class-6-aggregation-and-grouping[2] and week7-tutorial-case-study-5-easydrive-insurance[3].

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.

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

```sql
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.

```sql
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;
```

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[4]). To filter rows against an aggregate, compute it in a subquery: sql-nested-queries[5]. 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.

```sql
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[6], 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.

[4] courses/infs1200/sql-select-statement.pdf
[5] courses/infs1200/sql-nested-queries.pdf
[6] courses/infs1200/sql-select-statement.pdf

```sql
-- 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;

-- 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[7].

Idioms worth recognising:

```
-- "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[8] 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[9] — "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

[7] courses/infs1200/sql-nested-queries.pdf
[8] courses/infs1200/sql-nulls.pdf
[9] courses/infs1200/relational-division.pdf

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[10]).

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