The SELECT Statement and Its Clause Order

Table of contents

The full clause skeleton 1
Logical evaluation order 2
Projection and selection 3
Projection, duplicates and bag semantics 3
Expressions and aliases 4
The WHERE clause 4
ORDER BY 5

`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[1]), nested inside another query (sql-nested-queries[2]), stacked with another result set (sql-set-operations[3]), or stored as a view (sql-views[4]). Introduced in 2026-03-30-basic-sql-ddl-and-dml[5] and extended across 2026-04-06-aggregation-grouping-and-multiple-relation-queries[6] and 2026-04-13-nested-queries-views-and-generative-ai[7].

The full clause skeleton

```
SELECT [DISTINCT] <target list>
FROM   <table list>
[WHERE search condition]
[GROUP BY <grouping attributes>]
```

[1] courses/infs1200/sql-joins.pdf
[2] courses/infs1200/sql-nested-queries.pdf
[3] courses/infs1200/sql-set-operations.pdf
[4] courses/infs1200/sql-views.pdf
[5] courses/infs1200/2026-03-30-basic-sql-ddl-and-dml.pdf
[6] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[7] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf

```
[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[8]).
- **`HAVING` may use aggregates, `WHERE` may not**, because `HAVING` runs at stage 4, after the groups exist.
- **`WHERE` filters rows, `HAVING` filters groups.** They are not interchangeable, and a query can sensibly use both — see sql-aggregation-and-grouping[9].

[8] courses/infs1200/sql-nested-queries.pdf
[9] courses/infs1200/sql-aggregation-and-grouping.pdf

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[10]. `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.

[10] 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[11].

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[12]
`+ - * /`, 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`

[11] courses/infs1200/sql-joins.pdf
[12] 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[13] 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

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

[13] 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[14]), 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.

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