SQL Joins

Table of contents

Renaming and qualification 1
Cartesian product 2
Two syntaxes for the same thing 2
Kinds of join 3
Inner vs outer joins 3
Self-joins 4
Joining more than two tables 5
Choosing the join condition 5

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[1] are precisely the paths a join follows. Introduced in 2026-04-06-aggregation-grouping-and-multiple-relation-queries[2]; practised in week8-tutorial-applied-class-7-multiple-relation-queries[3].

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;
```

[1] courses/infs1200/er-to-relational-mapping.pdf
[2] courses/infs1200/2026-04-06-aggregation-grouping-and-multiple-relation-queries.pdf
[3] courses/infs1200/week8-tutorial-applied-class-7-multiple-relation-queries.pdf

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

```
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.

```sql
-- 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[4]): an inner join makes the zero-cases disappear entirely rather than counting them as zero.

```sql
-- 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;
```

The `NULL`s an outer join introduces are the query's own doing, not missing data in the base tables — see sql-nulls[5].

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;
```

[4] courses/infs1200/sql-aggregation-and-grouping.pdf
[5] courses/infs1200/sql-nulls.pdf

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.

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[6]). 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[7] stack result sets as new *rows* rather than new columns, and sql-nested-queries[8] 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[9] discusses under denormalisation, and the operation decomposition[10] must keep lossless.

[6] courses/infs1200/foreign-keys.pdf
[7] courses/infs1200/sql-set-operations.pdf
[8] courses/infs1200/sql-nested-queries.pdf
[9] courses/infs1200/normal-forms.pdf
[10] courses/infs1200/decomposition.pdf