# Relational Division

## Table of contents


The operation . . . . . 1
In SQL . . . . . 2
Recognising it in the wild . . . . . 4


**Division** is the operation that answers *"for all" / "for every"* questions — find the X that are related to **every** Y, such as the movie stars who appeared in *all* the movies produced in a given year, or the employees who can access *all* the websites some other employee can. It is the one common query shape that neither a join nor a simple aggregate expresses directly, which is why it gets its own name and its own two standard SQL patterns. Introduced in 2026-04-13-nested-queries-views-and-generative-ai[1]; applied in week9-tutorial-applied-class-8-nested-queries-and-views[2].

### The operation

Dividing $R1/R2$ requires the operands to be **division compatible**: R1's last $n$ columns must be identically named to R2's $n$ columns, where $n$ is R2's degree. The result has R1's columns *except* R2's, and contains a value $t$ only if $t$ appears in R1 in combination with **every** tuple of R2.

```
Student Degree   Subject          Subject   Student Degree
Anna    BIT      CS114     ÷      CS114  =  Anna    BIT
Anna    BIT      CS115            CS115
Anna    BIT      CS180
Fred    BSc      CS114
Fred    BSc      CS180
```

[1] courses/infs1200/2026-04-13-nested-queries-views-and-generative-ai.pdf
[2] courses/infs1200/week9-tutorial-applied-class-8-nested-queries-and-views.pdf

Fred is excluded: he hasn't taken `CS115`, so he doesn't divide evenly by `{CS114, CS115}`. Two things about the *shape* of the result are easy to get wrong:

- The result is **narrower** than R1 — it keeps only the columns not contributed by R2. Dividing a 3-column relation by a 2-column one gives a 1-column answer, not whole 3-column tuples.
- Extra rows in R1 are harmless; only the *required* combinations matter. Anna still qualifies despite also having taken `CS180`, which isn't in the divisor at all.

Dividing the same relation by successively larger divisors shows how the demand tightens. With `A [sno, pno]` holding `s1`→`{p1,p2,p3,p4}`, `s2`→`{p1,p2}`, `s3`→`{p2}`, `s4`→`{p2,p4}`:

| Divisor | Requires | `A / B` |
|---|---|---|
| `B1 = {p2}` | `p2` | `s1, s2, s3, s4` |
| `B2 = {p2, p4}` | `p2` **and** `p4` | `s1, s4` |
| `B3 = {p1, p2, p4}` | all three | `s1` |

Every divisor row is an extra hurdle, so the result can only shrink as the divisor grows — a useful sanity check on an exam answer. And an empty divisor would be satisfied by everybody, which is the degenerate end of the same pattern.

## In SQL

The first move on any division question is to name the dividend and the divisor before writing a line of SQL. "Find the movie star(s) who acted in *at least all* the movies produced in the year 1934" resolves to

```
StarsIn  ÷  (the movieIDs of Movies from 1934)  =  the qualifying stars
```

— the dividend is the relationship table, the divisor is the filtered set of things that must all be covered, and the result is the dividend's remaining column.

SQL has no `/` operator for relations, so division is expressed either by **counting** or by **double negation**. Both are worth having ready: the counting form is shorter, while the double-negation form is a direct transcription of the logical reading of the question.

The two are the same picture read from opposite sides. Draw the movies produced in 1934 as one circle and the movies X acted in as another; X qualifies exactly when the first circle sits wholly inside the second.

| Form | What it checks |
|---|---|
| **Counting** | The **overlap** is as large as the whole first circle — X acted in as many 1934 movies as there are 1934 movies |
| **Double negation** | The part of the first circle **outside** the second is **empty** — there is no 1934 movie X did not act in |

### Via counting

For a candidate X, count how many of the required Ys it is related to, and compare that against the total number of Ys. Equal counts mean X is related to all of them.

```sql
-- Star(s) who acted in at least all the movies produced in 1934
SELECT   X.starID, X.name
FROM     MovieStar X
JOIN     StarsIn S ON X.starID = S.starID
JOIN     Movie M ON S.movieID = M.movieID
WHERE    M.year = 1934
GROUP BY X.starID
HAVING   COUNT(*) = (SELECT COUNT(*) FROM Movie M2 WHERE M2.year = 1934);
```

The `WHERE M.year = 1934` restricts the count to the relevant movies, and the scalar subquery in `HAVING` supplies the target (sql-aggregation-and-grouping[3], sql-nested-queries[4]).

### Via double negation

Reformulate "X is related to every Y" as "there does **not exist** a Y that X is not related to":

```sql
-- "Find X such that there is no 1934 movie which X did not act in"
SELECT starID, name
FROM   MovieStar X
WHERE  NOT EXISTS (
    SELECT *
    FROM   StarsIn S
    JOIN   Movie M ON S.movieID = M.movieID
    WHERE  M.year = 1934
      AND  M.movieID NOT IN (
          SELECT movieID FROM StarsIn M2 WHERE M2.starID = X.starID
```

[3] courses/infs1200/sql-aggregation-and-grouping.pdf
[4] courses/infs1200/sql-nested-queries.pdf

```
      )
);
```

Read it inside out: the innermost subquery is "all the movies X acted in"; the middle one is "1934 movies X did **not** act in"; the outer `NOT EXISTS` says "there is no such movie" — i.e. X acted in every one. The correlation on `X.starID` is what makes the inner queries mean "this star" rather than "any star".

### Recognising it in the wild

The giveaway phrasings are *all*, *every*, and *at least all* — the last of which is division rather than equality: X may reach more than the divisor requires and still qualify. The same skeleton, with a subquery supplying the divisor, answers "employees with access to at least all the websites `E0007` has access to": there is no website `E0007` can reach that this employee cannot.

```
SELECT E.FirstName, E.LastName
FROM   Employee E
WHERE  NOT EXISTS (
    SELECT P.WebsiteURI
    FROM   Employee E2
    JOIN   RoleGranting RG1 ON RG1.EmployeeID = E2.EmployeeID
    JOIN   Permission P ON P.RoleID = RG1.RoleID
    WHERE  E2.EmployeeID = 'E0007'
      AND  P.WebsiteURI NOT IN (
          SELECT P2.WebsiteURI
          FROM   RoleGranting RG2
          JOIN   Permission P2 ON P2.RoleID = RG2.RoleID
          WHERE  RG2.EmployeeID = E.EmployeeID
      )
);
```