CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 4: Relational Algebra II

Amit Kumar Dhar

August 5, 2026

IIT Bhilai

Recap of Lecture 3

- Fundamental operators: $\sigma, \pi, \rho, \cup, -, \times$.
- σ over $\times$ gives the idea of a **join**.
- Everything composes into a **query tree**.
- Today: the **join family**, **division**, **aggregation**, and the **RA** $\leftrightarrow$ **SQL** translation that carries us into SQL next.

Agenda

- Why joins (product + selection is clumsy)
- **Theta join** and **equijoin**
- **Natural join** $\bowtie$
- **Outer joins** (left, right, full)
- **Division** $\div$
- **Aggregation** and grouping (extended RA)
- The full **RA** $\leftrightarrow$ **SQL** table

The Join Family

The problem with plain product

- `instructor` $\times$ `department` = 12 $\times$ 7 = 84 rows, most meaningless.
- We almost always want the **matching** pairs only.
- Writing $\sigma_{\ldots=\ldots}$ every time is verbose and error-prone.
- **Join operators** package ”product + the matching condition” cleanly.

Theta join $\bowtie_\theta$

- $R \bowtie_\theta S = \sigma_\theta(R \times S)$ — product, keep tuples satisfying θ.
- θ is any predicate ($<, \leq, =, \neq, >, \geq$, combined with $\wedge/\vee$).

$$\textit{instructor} \bowtie_{\textit{instructor.dept\_name}=\textit{department.dept\_name}} \textit{department}$$

- Pairs each instructor with the department row it belongs to.

Equijoin

- A **theta join whose condition is only equalities** is an **equijoin**.
- The most common case in practice (joining on key = foreign key).

$$student \bowtie_{student.dept\_name = department.dept\_name} department$$

- The join column appears **twice** (once from each side) — natural join fixes that.

Natural join $\bowtie$

- $R \bowtie S$ automatically:
  1. finds all attributes with the **same name** in both,
  2. requires them to be **equal**, and
  3. keeps just **one copy** of each shared attribute.

$$instructor \bowtie department$$

- joins on `dept_name` (shared), producing one `dept_name` column.

Natural join — worked example

`instructor(id, name, dept_name, salary)` ⋈
`department(dept_name, building, budget)`:

id	name	dept_name	salary	building	budget
10101	Srinivasan	Comp. Sci.	95000	Taylor	1100000
22222	Einstein	Physics	95000	Watson	700000

- Matched on `dept_name`; kept **one** `dept_name` column.

Natural join — the danger

- Natural join matches on **all** identically-named columns.
- If two tables share `building` *and* `dept_name`, both are used — maybe not what you meant.
- **Silent bug risk:** renaming a column can change which join happens.
- Best practice: prefer **explicit** conditions (equijoin) in real SQL.

Chaining joins

- Joins compose: link three or more relations.

$$student \bowtie takes \bowtie section \bowtie course$$

- ”Which student took which course (with title)?”
- Natural join follows the shared keys (`id`, then the section key, then `course_id`).
- Join is **associative & commutative** (for inner/natural) $\rightarrow$ the optimizer chooses the order (Week 10).

Outer Joins

The matching problem — some rows disappear

- Inner/natural joins keep only tuples that **match on both sides**.
- A section with **no instructor** (CS-319 sec 2) vanishes from `section` ⋈ `teaches`.
- Sometimes we want to keep unmatched rows and pad with **NULL**.
- Enter **outer joins**.

Left outer join

- Left outer join keeps **every** tuple of the left relation; unmatched right side → NULLs.

section **LEFT OUTER JOIN** *teaches*

course_id	sec_id	...	instructor id
CS-319	2	...	**NULL**

- "All sections, with the instructor **if any**."

Right and full outer joins

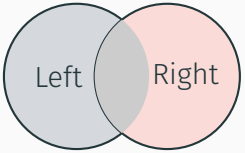
- **Right outer join** — keep every tuple of the **right** relation.
  - Equivalent to flipped Left outer join.
- **Full outer join** — keep unmatched tuples from **both** sides, padding with NULLs.
- Use full outer join to find ”rows on either side with no partner.”

Outer joins summary

Join	Keeps unmatched from
Inner	neither
Left	left only
Right	right only
Full	both

Padding for missing values is always **NULL** — outer joins are a major source of NULLs (and of Lecture 9's three-valued logic).

Full: All
Left: Left + Center
Inner: Center Only



Division

Division ÷ — ”for all”

- $R \div S$ answers **”find x related to *every* y in S.”**
- If $R(a, b)$ and $S(b)$, then $R \div S$ = the set of a values that appear paired with **all** b in S.
- Classic query: *”students who took **every** Comp. Sci. course.”*

Division — picture

`enrolled(sid, cid)` ÷ `cs_courses(cid)`:

- Keep a student `sid` **only if** for *every* course in `cs_courses`, the row `(sid, cid)` exists in `enrolled`.
- ”Supplies every part,” ”covers all regions,” ”passed all exams” — all division.
- SQL has no ÷ keyword; we emulate it with **`NOT EXISTS ... NOT EXISTS`** (double negation) — Week 3.

Division from fundamentals

- Division is derivable:

$$R \div S = \pi_a(R) - \pi_a\big((\pi_a(R) \times S) - R\big)$$

- Read it as: all candidate a, minus those missing **some** required pair.
- You don't need to memorize this — but it shows RA's expressive completeness.

Aggregation

Aggregation (extended RA)

- Pure RA has **no** SUM/COUNT/AVG — it's set-oriented.
- **Extended RA** adds an aggregation operator, written $\mathcal{G}$ (script G):

$$_{G_1,\ldots,G_k}\mathcal{G}_{F_1(A_1),\ldots,F_m(A_m)}(R)$$

- $G_1 \ldots G_k$ = grouping attributes; $F_i(A_i)$ = aggregate functions (`sum`, `count`, `avg`, `min`, `max`).

Aggregation without grouping

- ”Average instructor salary”:

$$\mathcal{G}_{avg(salary)}(instructor)$$

- No grouping attributes $\rightarrow$ the **whole relation** is one group $\rightarrow$ a single row.

avg_salary
78700

Aggregation with grouping

- "Average salary **per department**":

$$_{dept\_name}\mathcal{G}_{avg(salary)}(instructor)$$

dept_name	avg_salary
Comp. Sci.	88666.7
Physics	92333.3
Finance	85000

- One output row **per group**. This is exactly SQL's **GROUP BY** (Week 3).

Semijoin and antijoin (useful patterns)

- **Semijoin ($\ltimes$):** rows of R that have **at least one** match in S, keeping only R's columns.
  - "instructors who teach *something*" — don't duplicate per section.
  - SQL: `WHERE EXISTS (SELECT 1 FROM teaches t WHERE t.id = i.id)`.
- **Antijoin ($\triangleright$):** rows of R with **no** match in S.
  - "instructors who teach *nothing*."
  - SQL: `WHERE NOT EXISTS (...)` or `LEFT JOIN ... WHERE right IS NULL`.

Semijoin vs inner join — why it matters

```sql
-- inner join: an instructor teaching 3 sections appears 3 times
SELECT i.name FROM instructor i JOIN teaches t ON t.id = i.id;

-- semijoin: each qualifying instructor appears once
SELECT i.name FROM instructor i
WHERE EXISTS (SELECT 1 FROM teaches t WHERE t.id = i.id);
```

- Semijoin answers "**does a match exist?**" without row multiplication.
- We formalize `EXISTS`/`NOT EXISTS` in Week 3.

Aggregation: the five standard functions

Function	Meaning	NULL handling
`count(*)`	number of rows	counts everything
`count(A)`	non-null values of A	**ignores NULLs**
`sum(A)`	total	ignores NULLs
`avg(A)`	mean	ignores NULLs
`min/max(A)`	extremes	ignores NULLs

- `avg` divides by the count of **non-null** values — a frequent surprise.

Aggregation edge cases

- `count(*)` vs `count(grade)`: the latter **excludes** ungraded (NULL) rows.
- `sum` of an empty group is **NULL**, not 0 (wrap with `COALESCE` for 0).
- `avg` of all-NULL is NULL.
- `count(DISTINCT A)` counts distinct non-null values.

These map directly to the SQL we'll use heavily in Week 3.

Division worked example

`takes(id, course_id)` ÷ `cs_courses(course_id)` = ”students who took **every** CS course.”

- Candidate students: $\pi_{id}(takes)$.
- Subtract students **missing at least one** required course.
- Result: only students whose enrollments **cover all** CS courses.
- In SQL: the **double `NOT EXISTS`** idiom (Lecture 8).

RA $\leftrightarrow$ SQL Translation

RA $\leftrightarrow$ SQL: the unary operators

Relational algebra	SQL
$\sigma_p(R)$	`SELECT * FROM R WHERE p`
$\pi_{A,B}(R)$	`SELECT DISTINCT A, B FROM R`
generalized π (computed col)	`SELECT A, expr AS c FROM R`
$\rho_x(R)$	`FROM R AS x`

RA $\leftrightarrow$ SQL: set operators

Relational algebra	SQL
$R \cup S$	`R UNION S`
$R \cap S$	`R INTERSECT S`
$R - S$	`R EXCEPT S`

- SQL adds `... ALL` variants that **keep duplicates** (`UNION ALL`, etc.).
- Plain RA is always duplicate-free.

RA $\leftrightarrow$ SQL: joins

Relational algebra	SQL
$R \times S$	`FROM R, S` or `R CROSS JOIN S`
$R \bowtie_\theta S$	`R JOIN S ON` θ
$R \bowtie S$ (natural)	`R NATURAL JOIN S`
R LEFT OUTER JOIN S	`R LEFT JOIN S ON` θ
R RIGHT OUTER JOIN S	`R RIGHT JOIN S ON` θ
R FULL OUTER JOIN S	`R FULL JOIN S ON` θ

RA $\leftrightarrow$ SQL: aggregation

Relational algebra	SQL
$\mathcal{G}_{avg(salary)}(R)$	`SELECT avg(salary) FROM R`
${}_{d}\mathcal{G}_{avg(salary)}(R)$	`SELECT d, avg(salary) FROM R GROUP BY d`
division $\div$	`... WHERE NOT EXISTS (... NOT EXISTS ...)`

By Lecture 6 you'll translate freely in both directions.

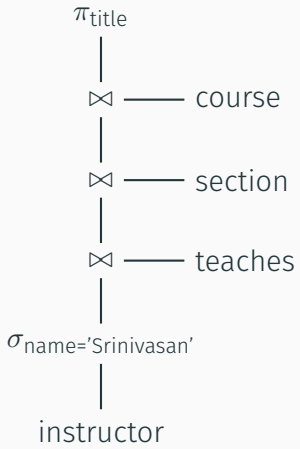
Full worked example (RA)

”Titles of courses taught by an instructor named ’Srinivasan’.”

$$\pi_{title}\Big(\sigma_{name='\text{Srinivasan}'}(instructor) \bowtie teaches \bowtie section \bowtie course\Big)$$

- Filter to Srinivasan, join through `teaches` $\rightarrow$ `section` $\rightarrow$ `course`, project titles.

The same query as a tree



The optimizer may **reorder** these joins for efficiency.

Theta join with an inequality

- Not all joins are equalities. "Instructor pairs where one out-earns another in the same department":

$$instructor \bowtie_{a.dept=b.dept \wedge a.salary>b.salary} instructor$$

- Requires renaming the relation (two copies *a*, *b*) — a **self theta-join**.
- SQL: `FROM instructor a JOIN instructor b ON a.dept_name=b.dept_name AND a.salary>b.salary`.

Outer join + aggregation = ”counts including zeros”

- A very common report pattern: *”number of sections each instructor teaches, including those who teach none.”*

$$_{i.id}\mathcal{G}_{count(t.sec\_id)}\left(instructor \text{ LEFT OUTER JOIN } teaches\right)$$

- The **left outer join** keeps zero-teaching instructors; `count` of the NULL-padded side yields **0**. We’ll write this in SQL next week.

Join elimination (why the optimizer cares)

- If a query joins `student` $\bowtie$ `department` but selects only `student` columns, and every student's `dept_name` is a valid FK, the join to `department` **can be removed** without changing the answer.
- The optimizer uses RA equivalences + constraints to do this automatically.
- Lesson: the **declarative** RA/SQL lets the system optimize behind your back.

When RA runs out (and SQL keeps going)

- RA (even extended) can't easily express:
  - "**rank** instructors by salary within each department,"
  - "running totals," "top-N per group."
- These need **window functions** — an SQL feature beyond classical RA (Lecture 9). RA is the foundation, not the ceiling.

Common mistakes

- Using **natural join** when tables share an *unexpected* column name.
- Expecting inner join to keep unmatched rows — use an **outer join**.
- Forgetting that RA aggregation collapses groups (one row per group).
- Reaching for division without recognizing the "**for all**" pattern.

Summary — key takeaways

- **Join = product + condition;** the family runs inner → left → right → full.
- **Natural join** matches shared names and de-duplicates them (convenient but risky).
- **Division** captures "for **all**" queries.
- **Extended RA** adds **aggregation/grouping**.
- Every RA operator maps to an SQL construct — we now switch to writing SQL.

Exercises

1. Write RA for ”instructors who teach **no** section” (hint: difference or outer join + NULL).
2. Express ”departments where **every** course has > 3 credits” with division.
3. Give the SQL for ${}_{dept}\mathcal{G}_{max(salary)}(instructor)$.
4. Why might $A \bowtie B \bowtie C$ be faster in one join order than another?

Before next lecture

- Optional reading: Silberschatz §3 intro (SQL) & §2.6 (extended RA).
- Install PostgreSQL is **not** needed yet; SQLite is enough through Week 3.

Next time — Lecture 5: SQL DDL and **single-table queries** — actually creating tables and running `SELECT` on real data.

Questions?