CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 3: Relational Model (contd.) & Relational Algebra I

Amit Kumar Dhar

IIT Bhilai

Part 1: Finishing the Relational Model

NULL — the value that isn't

- **NULL** means "unknown," "not applicable," or "missing."
- It is **not** zero and **not** the empty string.
- Examples: a student enrolled but **not yet graded** → `grade IS NULL`.
- NULLs make logic **three-valued** (true / false / unknown) — a whole topic (Lecture 9). For now: know they exist and are special.

The University schema (we use it all term)

```
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(id, name, dept_name, salary)
student(id, name, dept_name, tot_cred)
section(course_id, sec_id, semester, year, building, room_no)
teaches(id, course_id, sec_id, semester, year)
takes(id, course_id, sec_id, semester, year, grade)
prereq(course_id, prereq_id)
advisor(s_id, i_id)
classroom(building, room_no, capacity)
```

University schema — the relationships

- `student.dept_name`, `instructor.dept_name`, `course.dept_name` → `department`.

University schema — the relationships

- `student.dept_name`, `instructor.dept_name`, `course.dept_name` → `department`.
- `section.course_id` → `course`; `(building, room_no)` → `classroom`.

University schema — the relationships

- `student.dept_name`, `instructor.dept_name`, `course.dept_name` → `department`.
- `section.course_id` → `course`; `(building, room_no)` → `classroom`.
- `teaches` links `instructor` ↔ `section`.

University schema — the relationships

- `student.dept_name`, `instructor.dept_name`, `course.dept_name` → `department`.
- `section.course_id` → `course`; `(building, room_no)` → `classroom`.
- `teaches` links `instructor` ↔ `section`.
- `takes` links `student` ↔ `section` (with a `grade`).

University schema — the relationships

- `student.dept_name`, `instructor.dept_name`, `course.dept_name` → `department`.
- `section.course_id` → `course`; `(building, room_no)` → `classroom`.
- `teaches` links `instructor` ↔ `section`.
- `takes` links `student` ↔ `section` (with a `grade`).
- `advisor` links `student` ↔ `instructor`.

University schema — the relationships

- `student.dept_name`, `instructor.dept_name`, `course.dept_name` → `department`.
- `section.course_id` → `course`; `(building, room_no)` → `classroom`.
- `teaches` links `instructor` ↔ `section`.
- `takes` links `student` ↔ `section` (with a `grade`).
- `advisor` links `student` ↔ `instructor`.
- `prereq` links `course` ↔ `course` (self-relationship).

Composite & multi-attribute keys

- Some keys need **several** attributes together.

Composite & multi-attribute keys

- Some keys need **several** attributes together.
- A `section` is identified by **all** of:

Composite & multi-attribute keys

- Some keys need **several** attributes together.
- A `section` is identified by **all** of:
  `(course_id, sec_id, semester, year)`.

Composite & multi-attribute keys

- Some keys need **several** attributes together.
- A `section` is identified by **all** of:
  `(course_id, sec_id, semester, year)`.
- No single attribute is unique — CS-101 runs in many semesters.

Composite & multi-attribute keys

- Some keys need **several** attributes together.
- A `section` is identified by **all** of:
  `(course_id, sec_id, semester, year)`.
- No single attribute is unique — CS-101 runs in many semesters.
- `takes` and `teaches` inherit this composite key (plus the student/instructor id).

Representing relationships as tables

- **1-to-many** (a department has many students):
  put the ”one” side’s PK as an FK on the ”many” side (`student.dept_name`).

Representing relationships as tables

- **1-to-many** (a department has many students):
  put the ”one” side’s PK as an FK on the ”many” side (`student.dept_name`).
- **Many-to-many** (students take many sections; sections have many students):
  make a **separate relation** — `takes` — holding both keys.

Representing relationships as tables

- **1-to-many** (a department has many students):
  put the ”one” side’s PK as an FK on the ”many” side (`student.dept_name`).
- **Many-to-many** (students take many sections; sections have many students):
  make a **separate relation** — `takes` — holding both keys.
- **Self-relationship** (course prerequisites):
  a table referencing the same relation twice — `prereq(course_id, prereq_id)`.

Worked example: modelling `takes`

- A student can take many sections; a section has many students → many-to-many.
- Create a relation with both foreign keys **and** any relationship attributes:

```
takes(id, course_id, sec_id, semester, year, grade)
  id -> student(id)
  (course_id, sec_id, semester, year) -> section(...)
  grade  <- attribute OF the enrollment relationship
```

- `grade` belongs to the *enrollment*, not to the student or the section alone.

Turning a miniworld into tables (recipe)

1. Identify the **entities** (student, course, department…).

Turning a miniworld into tables (recipe)

1. Identify the **entities** (student, course, department…).
2. Give each a table with a **primary key**.

Turning a miniworld into tables (recipe)

1. Identify the **entities** (student, course, department…).
2. Give each a table with a **primary key**.
3. Identify **relationships**; place FKs (1-to-many) or new tables (many-to-many).

Turning a miniworld into tables (recipe)

1. Identify the **entities** (student, course, department…).
2. Give each a table with a **primary key**.
3. Identify **relationships**; place FKs (1-to-many) or new tables (many-to-many).
4. Add **domain constraints** (types, CHECKs).

Turning a miniworld into tables (recipe)

1. Identify the **entities** (student, course, department…).
2. Give each a table with a **primary key**.
3. Identify **relationships**; place FKs (1-to-many) or new tables (many-to-many).
4. Add **domain constraints** (types, CHECKs).
5. Declare **referential integrity** for every FK.

Turning a miniworld into tables (recipe)

1. Identify the **entities** (student, course, department…).
2. Give each a table with a **primary key**.
3. Identify **relationships**; place FKs (1-to-many) or new tables (many-to-many).
4. Add **domain constraints** (types, CHECKs).
5. Declare **referential integrity** for every FK.

Turning a miniworld into tables (recipe)

1. Identify the **entities** (student, course, department…).
2. Give each a table with a **primary key**.
3. Identify **relationships**; place FKs (1-to-many) or new tables (many-to-many).
4. Add **domain constraints** (types, CHECKs).
5. Declare **referential integrity** for every FK.

(Weeks 5–6 formalize this as ER modelling + normalization.)

What the relational model does NOT prescribe

- **How** rows are stored (files, pages) — physical independence.

What the relational model does NOT prescribe

- **How** rows are stored (files, pages) — physical independence.
- **Any** row ordering — you get order only via `ORDER BY`.

What the relational model does NOT prescribe

- **How** rows are stored (files, pages) — physical independence.
- **Any** row ordering — you get order only via `ORDER BY`.
- **Access paths** — indexes are a performance choice, invisible to the model.

What the relational model does NOT prescribe

- **How** rows are stored (files, pages) — physical independence.
- **Any** row ordering — you get order only via `ORDER BY`.
- **Access paths** — indexes are a performance choice, invisible to the model.
- This separation is exactly what lets the optimizer do its job (Week 10).

From model to language

- The relational model gives us the **data structure** (relations + constraints).

From model to language

- The relational model gives us the **data structure** (relations + constraints).
- We now need a way to **ask questions**.

From model to language

- The relational model gives us the **data structure** (relations + constraints).
- We now need a way to **ask questions**.
- **Relational algebra** (Lecture 3) — a formal, operator-based query language.

From model to language

- The relational model gives us the **data structure** (relations + constraints).
- We now need a way to **ask questions**.
- **Relational algebra** (Lecture 3) — a formal, operator-based query language.
- **SQL** (Week 2) — the practical language built on those ideas.

Common misconceptions

- ”Rows have an order.” — **No**, a relation is a set.

Common misconceptions

- ”Rows have an order.” — **No**, a relation is a set.
- ”A table can’t have duplicate rows.” — In the *pure model* no; in *SQL* yes, unless a key/constraint forbids it.

Common misconceptions

- ”Rows have an order.” — **No**, a relation is a set.
- ”A table can’t have duplicate rows.” — In the *pure model* no; in *SQL* yes, unless a key/constraint forbids it.
- ”NULL = 0 or ”.” — **No**, NULL is *unknown*.

Common misconceptions

- ”Rows have an order.” — **No**, a relation is a set.
- ”A table can’t have duplicate rows.” — In the *pure model* no; in *SQL* yes, unless a key/constraint forbids it.
- ”NULL = 0 or ””.” — **No**, NULL is *unknown*.
- ”Primary key must be one column.” — No, it can be **composite**.

Exercises to try (Relational Model)

1. List all candidate keys of `takes`. Which would you pick as PK and why?
2. Why can't `grade` live in the `student` table?
3. Which FKs in `section` could reasonably be NULL? Which must not?
4. Give a superkey of `instructor` that is **not** a candidate key.

Part 2: Relational Algebra

Recap & Motivation

- We have a **data model**: relations + keys + constraints.

Recap & Motivation

- We have a **data model**: relations + keys + constraints.
- Now we need a way to **ask questions** about the data.

Recap & Motivation

- We have a **data model**: relations + keys + constraints.
- Now we need a way to **ask questions** about the data.
- **Relational algebra (RA)** is a formal query language: operators that take relations in and produce a relation out.

Recap & Motivation

- We have a **data model**: relations + keys + constraints.
- Now we need a way to **ask questions** about the data.
- **Relational algebra (RA)** is a formal query language: operators that take relations in and produce a relation out.
- It's the **theory under SQL** and the **language of the optimizer**.

The core idea

An RA operator takes one or two relations and returns a new relation.

- Because the output is a relation, you can feed it into another operator. (Closure)

The core idea

An RA operator takes one or two relations and returns a new relation.

- Because the output is a relation, you can feed it into another operator. (**Closure**)
- A **query** is an **expression** built from operators.

The core idea

An RA operator takes one or two relations and returns a new relation.

- Because the output is a relation, you can feed it into another operator. (**Closure**)
- A **query** is an **expression** built from operators.
- Six fundamental operators: $\sigma, \pi, \rho, \cup, -, \times$

Unary Operators

Selection σ (sigma) — pick rows

- $\sigma_{\text{predicate}}(R)$ returns the tuples of R satisfying the predicate.

Selection σ (sigma) — pick rows

- $\sigma_{\text{predicate}}(R)$ returns the tuples of R satisfying the predicate.
- It **filters rows**; the schema is unchanged.

Selection σ (sigma) — pick rows

- $\sigma_{\text{predicate}}(R)$ returns the tuples of R satisfying the predicate.
- It **filters rows**; the schema is unchanged.

Selection σ (sigma) — pick rows

- $\sigma_{\text{predicate}}(R)$ returns the tuples of R satisfying the predicate.
- It **filters rows**; the schema is unchanged.

$$\sigma_{dept\_name='\text{Comp. Sci.}'}(instructor)$$

id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	95000
45565	Katz	Comp. Sci.	75000

Projection π (pi) — pick columns

- $\pi_{A_1,\ldots,A_n}(R)$ keeps only the listed attributes.

Projection π (pi) — pick columns

- $\pi_{A_1,\ldots,A_n}(R)$ keeps only the listed attributes.
- It **selects columns** and **drops the rest**.

Projection π (pi) — pick columns

- $\pi_{A_1,\ldots,A_n}(R)$ keeps only the listed attributes.
- It **selects columns** and **drops the rest**.
- RA relations are **sets** $\rightarrow$ projection **eliminates duplicate rows**.

Projection π (pi) — pick columns

- $\pi_{A_1,\ldots,A_n}(R)$ keeps only the listed attributes.
- It **selects columns** and **drops the rest**.
- RA relations are **sets** $\rightarrow$ projection **eliminates duplicate rows**.

Projection π (pi) — pick columns

- $\pi_{A_1,\ldots,A_n}(R)$ keeps only the listed attributes.
- It **selects columns** and **drops the rest**.
- RA relations are **sets** $\rightarrow$ projection **eliminates duplicate rows**.

$$\pi_{name,\, salary}(instructor)$$

name	salary
Srinivasan	95000
Einstein	95000

Composing σ and π

"Names of instructors in Comp. Sci."

$$\pi_{name}\left(\sigma_{dept\_name=\text{'Comp. Sci.'}}(instructor)\right)$$

- Read **inside-out**: filter first, then project.

Composing σ and π

"Names of instructors in Comp. Sci."

$$\pi_{name}\left(\sigma_{dept\_name='\text{Comp. Sci.}'}(instructor)\right)$$

- Read **inside-out**: filter first, then project.
- **Closure** in action!

Composing σ and π

"Names of instructors in Comp. Sci."

$$\pi_{name}\left(\sigma_{dept\_name=\text{'Comp. Sci.'}}(instructor)\right)$$

- Read **inside-out**: filter first, then project.
- **Closure** in action!
- Does order matter? $\pi_{name}(\sigma_{\dots})$ is valid, but $\sigma_{\dots}(\pi_{name})$ might be invalid if π drops the attribute needed by σ.

Rename ρ (rho)

- $\rho_x(R)$ renames relation R to x.

Rename ρ (rho)

- $\rho_x(R)$ renames relation R to x.
- $\rho_{x(A_1,\ldots,A_n)}(R)$ also renames the attributes.

Rename ρ (rho)

- $\rho_x(R)$ renames relation R to x.
- $\rho_{x(A_1,\ldots,A_n)}(R)$ also renames the attributes.
- Why rename?

Rename ρ (rho)

- $\rho_x(R)$ renames relation R to x.
- $\rho_{x(A_1,\ldots,A_n)}(R)$ also renames the attributes.
- Why rename?
  - temporary name in an expression,

Rename ρ (rho)

- $\rho_x(R)$ renames relation R to x.
- $\rho_{x(A_1,\ldots,A_n)}(R)$ also renames the attributes.
- Why rename?
  - temporary name in an expression,
  - to **compare a relation with itself**,

Rename ρ (rho)

- $\rho_x(R)$ renames relation R to x.
- $\rho_{x(A_1,\ldots,A_n)}(R)$ also renames the attributes.
- Why rename?
  - temporary name in an expression,
  - to **compare a relation with itself**,
  - resolve name clashes after a product.

Rename ρ (rho)

- $\rho_x(R)$ renames relation R to x.
- $\rho_{x(A_1,\ldots,A_n)}(R)$ also renames the attributes.
- Why rename?
  - temporary name in an expression,
  - to **compare a relation with itself**,
  - resolve name clashes after a product.

Rename ρ (rho)

- $\rho_x(R)$ renames relation R to x.
- $\rho_{x(A_1,\ldots,A_n)}(R)$ also renames the attributes.
- Why rename?
  - temporary name in an expression,
  - to **compare a relation with itself**,
  - resolve name clashes after a product.

$$\rho_T(\mathit{instructor}), \qquad \rho_{S(\mathit{sid},\mathit{sname})}(\mathit{student})$$

Set Operators

Set operators & Union-compatibility

- $\cup, \cap, -$ treat relations as sets of tuples.

Set operators & Union-compatibility

- $\cup, \cap, -$ treat relations as sets of tuples.
- They require the two relations to be **union-compatible**:

Set operators & Union-compatibility

- $\cup, \cap, -$ treat relations as sets of tuples.
- They require the two relations to be **union-compatible**:
  - **same number of attributes**, and

Set operators & Union-compatibility

- $\cup, \cap, -$ treat relations as sets of tuples.
- They require the two relations to be **union-compatible**:
  - **same number of attributes**, and
  - **corresponding attributes over the same domains**.

Set operators examples

Union $\cup$ Tuples in R, or S, or both (duplicates removed).

Set operators examples

Union $\cup$ Tuples in R, or S, or both (duplicates removed).

$$\pi_{name}(instructor) \cup \pi_{name}(student)$$

Set operators examples

Union $\cup$ Tuples in R, or S, or both (duplicates removed).

$$\pi_{name}(instructor) \cup \pi_{name}(student)$$

Difference $-$ Tuples in R **but not** in S. (”not exists”)

Set operators examples

Union $\cup$ Tuples in R, or S, or both (duplicates removed).

$$\pi_{name}(instructor) \cup \pi_{name}(student)$$

Difference $-$ Tuples in R **but not** in S. (”not exists”)

$$\pi_{dept\_name}(instructor) - \pi_{dept\_name}(student)$$

Set operators examples

Union $\cup$ Tuples in R, or S, or both (duplicates removed).

$$\pi_{name}(instructor) \cup \pi_{name}(student)$$

Difference $-$ Tuples in R **but not** in S. (”not exists”)

$$\pi_{dept\_name}(instructor) - \pi_{dept\_name}(student)$$

Intersection $\cap$ Tuples in **both** R and S. (Derived: $R - (R - S)$)

Set operators examples

Union $\cup$ Tuples in R, or S, or both (duplicates removed).

$$\pi_{name}(instructor) \cup \pi_{name}(student)$$

Difference $-$ Tuples in R **but not** in S. (”not exists”)

$$\pi_{dept\_name}(instructor) - \pi_{dept\_name}(student)$$

Intersection $\cap$ Tuples in **both** R and S. (Derived: $R - (R - S)$)

$$\pi_{name}(instructor) \cap \pi_{name}(student)$$

Cartesian Product

Cartesian product $\times$

- $R \times S$ pairs **every** tuple of R with **every** tuple of S.

Cartesian product $\times$

- $R \times S$ pairs **every** tuple of R with **every** tuple of S.
- Result schema = all attributes of R **and** S.

Cartesian product $\times$

- $R \times S$ pairs **every** tuple of R with **every** tuple of S.
- Result schema = all attributes of R **and** S.
- Result size = $|R| \times |S|$ tuples.

Cartesian product $\times$

- $R \times S$ pairs **every** tuple of R with **every** tuple of S.
- Result schema = all attributes of R **and** S.
- Result size = $|R| \times |S|$ tuples.
- Most pairs are **meaningless** on their own...

Cartesian product $\times$

- $R \times S$ pairs **every** tuple of R with **every** tuple of S.
- Result schema = all attributes of R **and** S.
- Result size = $|R| \times |S|$ tuples.
- Most pairs are **meaningless** on their own…
- …which is why we immediately **filter** with σ.

Product + selection = Join

”Each instructor with their department’s building”:

$$\sigma_{instructor.dept\_name=department.dept\_name}(instructor \times department)$$

- Pair everything, then **keep only matching** pairs.

Product + selection = Join

”Each instructor with their department’s building”:

$$\sigma_{instructor.dept\_name=department.dept\_name}(instructor \times department)$$

- Pair everything, then **keep only matching** pairs.
- This filtered product is precisely a **join** (Lecture 4).

Query Trees & Composition

Query trees

Any RA expression is a **tree**:

- **leaves** = base relations,
- **internal nodes** = operators,
- **root** = final result.

The optimizer manipulates **exactly this tree**.

Worked query

”For each section, the building and room capacity.”

$$\sigma_{\substack{\mathit{section.building}=\mathit{classroom.building} \\ \wedge\ \mathit{section.room\_no}=\mathit{classroom.room\_no}}} (\mathit{section} \times \mathit{classroom})$$

then project `course_id, sec_id, capacity`.

RA $\rightarrow$ SQL sneak peek

RA	SQL
$\sigma_p(R)$	`SELECT * FROM R WHERE p`
$\pi_A(R)$	`SELECT DISTINCT A FROM R`
$R \cup S$	`... UNION ...`
$R \cap S$	`... INTERSECT ...`
$R - S$	`... EXCEPT ...`
$R \times S$	`SELECT * FROM R, S`

RA $\rightarrow$ SQL sneak peek

RA	SQL
$\sigma_p(R)$	`SELECT * FROM R WHERE p`
$\pi_A(R)$	`SELECT DISTINCT A FROM R`
$R \cup S$	`... UNION ...`
$R \cap S$	`... INTERSECT ...`
$R - S$	`... EXCEPT ...`
$R \times S$	`SELECT * FROM R, S`

SQL **SELECT** = RA π (projection). SQL **WHERE** = RA σ (selection).

Summary — key takeaways

- RA operators consume relations and produce relations $\rightarrow$ they **compose**.

Summary — key takeaways

- RA operators consume relations and produce relations $\rightarrow$ they **compose**.
- **Unary:** σ (rows), π (columns), ρ (renaming).

Summary — key takeaways

- RA operators consume relations and produce relations $\rightarrow$ they **compose**.
- **Unary:** σ (rows), π (columns), ρ (renaming).
- **Set:** $\cup, \cap, -$ (need union-compatibility).

Summary — key takeaways

- RA operators consume relations and produce relations $\rightarrow$ they **compose**.
- **Unary:** σ (rows), π (columns), ρ (renaming).
- **Set:** $\cup, \cap, -$ (need union-compatibility).
- $\times$ pairs all tuples; σ over $\times$ gives us **joins**.

Summary — key takeaways

- RA operators consume relations and produce relations $\rightarrow$ they **compose**.
- **Unary:** σ (rows), π (columns), ρ (renaming).
- **Set:** $\cup, \cap, -$ (need union-compatibility).
- $\times$ pairs all tuples; σ over $\times$ gives us **joins**.
- Every query is a **tree** — the same structure the optimizer rewrites.

Questions?

Next time — Lecture 4: The Join Family