CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 6a: SQL Joins & Set Operations

Amit Kumar Dhar

IIT Bhilai

Where we are

- Lecture 6: querying **one** table (`SELECT`, `WHERE`, functions).
- Real questions span **many** tables connected by foreign keys.
- Today: **joins** (the SQL realization of RA's join family) and the **set operations** on query results.
- This is the most-used SQL skill in practice — we'll do lots of examples.

Agenda

- Generated columns and `INSERT` recap
- The join mental model
- `INNER JOIN ... ON` and `USING`
- Multi-table and Self joins
- Outer joins (LEFT / RIGHT / FULL)
- Set ops: `UNION` / `INTERSECT` / `EXCEPT`

Generated / computed columns (brief)

- Some engines let a column be **derived** from others:

```
-- PostgreSQL
ALTER TABLE instructor
  ADD COLUMN monthly NUMERIC GENERATED ALWAYS AS (salary/12) STORED;
```

- Keeps derived data consistent automatically. SQLite supports `GENERATED ALWAYS AS (...)` too (virtual or stored).

INSERT (so we have data to query)

```sql
INSERT INTO department (dept_name, building, budget)
VALUES ('Comp. Sci.', 'Taylor', 1100000);

INSERT INTO instructor (id, name, dept_name, salary) VALUES
    ('10101', 'Srinivasan', 'Comp. Sci.', 95000),
    ('22222', 'Einstein',   'Physics',    95000);
```

- Column list is optional but **recommended** (order-independent, future-proof).
- Full DML (UPDATE, DELETE, UPSERT) is Lecture 10.

SQL Joins

The join mental model

- A join **combines rows from two tables** based on a condition.
- Conceptually: form the **product**, then **keep matching rows**, then **project** what you want.
- The DBMS doesn't literally build the product — the optimizer picks a smart algorithm (Week 9). But this model predicts the **result** correctly.

Two tables to join

student
`(id, name, dept_name, tot_cred)`

department
`(dept_name, building, budget)`

Question: *"Each student with the building of their department."*
The link is `student.dept_name = department.dept_name`.

CROSS JOIN (Cartesian product)

```sql
SELECT * FROM student CROSS JOIN department;   -- 15 × 7 = 105 rows
SELECT * FROM student, department;             -- same thing (comma = cross joi
```

- Every student paired with every department — mostly meaningless.
- Rarely what you want **by itself**; add a condition to make it a join.

Quick Check: CROSS JOIN

Question: If the `course` table has 20 rows and the `instructor` table has 10 rows, how many rows does `SELECT * FROM course CROSS JOIN instructor;` return?

Quick Check: CROSS JOIN

Question: If the `course` table has 20 rows and the `instructor` table has 10 rows, how many rows does `SELECT * FROM course CROSS JOIN instructor;` return?

Answer: 200 rows (20×10).

INNER JOIN … ON

```sql
SELECT s.name, d.building
FROM   student s
JOIN   department d ON s.dept_name = d.dept_name;
```

- `JOIN` = `INNER JOIN`.
- **ON** states the match condition (any predicate, usually key = FK).
- Only **matching** pairs survive.

Quick Check: INNER JOIN

Question: Write a query to find the names of students and the course IDs they have taken.

Quick Check: INNER JOIN

Question: Write a query to find the names of students and the course IDs they have taken.

Answer:

```
SELECT s.name, t.course_id
FROM student s
JOIN takes t ON s.id = t.id;
```

The old comma-join style

```sql
SELECT s.name, d.building
FROM   student s, department d
WHERE  s.dept_name = d.dept_name;
```

- Equivalent to the `INNER JOIN ... ON` form.
- **Prefer explicit `JOIN ... ON`:** the join condition can't be accidentally forgotten (forget the `WHERE` and you get a giant cross join).

JOIN … USING

- When the join columns share a name, `USING` is shorthand:

```
SELECT name, building
FROM   student JOIN department USING (dept_name);
```

- Joins on `dept_name` and keeps **one** copy of it.
- Cleaner than `ON`, but only works when the names match.

NATURAL JOIN — convenient but risky

```
SELECT name, building
FROM   student NATURAL JOIN department;
```

- Joins on **all** identically-named columns automatically.
- **Danger:** add a column named `budget` to both tables and the join silently changes. Avoid `NATURAL JOIN` in code you'll maintain — prefer `ON`/`USING`.

Qualifying column names

- After a join, ambiguous columns must be **qualified**:

```
SELECT student.dept_name        -- which dept_name? qualify it
FROM   student JOIN department
       ON student.dept_name = department.dept_name;
```

- Table **aliases** make this concise: `s.dept_name`, `d.dept_name`.
- Always alias in multi-table queries.

Advanced Joins

Multi-table joins

”Which student took which course title?” — chain four tables:

```
SELECT s.name, c.title, t.grade
FROM   student s
JOIN   takes   t  ON t.id = s.id
JOIN   section sec ON sec.course_id = t.course_id
                  AND sec.sec_id   = t.sec_id
                  AND sec.semester = t.semester
                  AND sec.year     = t.year
JOIN   course  c  ON c.course_id = sec.course_id;
```

- Each `JOIN` adds one table via its linking condition.

Join order & readability

- The result is the same regardless of the **written** join order (for inner joins) — the optimizer decides execution order.
- For humans: order joins to **follow the foreign keys**, start from the table you're "centered" on (`student` here).
- Indent multi-column `ON` conditions for readability.

Self join — a table joined to itself

- Use when rows relate to **other rows of the same table**.
- Requires **two aliases** for the one table.

”Pairs of instructors in the same department.”

```
SELECT a.name, b.name, a.dept_name
FROM   instructor a
JOIN   instructor b ON a.dept_name = b.dept_name
                   AND a.id < b.id;      -- avoid self-pairs & duplicates
```

Self join — prerequisites example

”Course title and its prerequisite’s title.”

```sql
SELECT c.title AS course, p.title AS prerequisite
FROM   prereq pr
JOIN   course c ON c.course_id = pr.course_id
JOIN   course p ON p.course_id = pr.prereq_id;
```

- `course` appears twice, aliased `c` and `p`.

Outer Joins

The matching problem returns

- Inner join **drops** rows with no match.
- ”List all sections and their instructor” — a section with **no** instructor disappears with an inner join.
- We need to **keep unmatched rows** $\rightarrow$ **outer joins**.

LEFT OUTER JOIN

```
SELECT sec.course_id, sec.sec_id, t.id AS instructor_id
FROM   section sec
LEFT JOIN teaches t
       ON sec.course_id = t.course_id AND sec.sec_id = t.sec_id
      AND sec.semester = t.semester   AND sec.year   = t.year;
```

- **Every** section appears; sections with no instructor get `NULL` for `t.id`.
- `OUTER` keyword is optional: `LEFT JOIN` = `LEFT OUTER JOIN`.

Quick Check: LEFT JOIN

Question: We want a list of ALL courses, and the ID of the instructor teaching them (if any). Which table should be on the left?

Quick Check: LEFT JOIN

Question: We want a list of ALL courses, and the ID of the instructor teaching them (if any). Which table should be on the left?

Answer: The `course` table should be on the left, so we don't lose courses that haven't been assigned an instructor yet.

```
SELECT c.title, t.id
FROM course c LEFT JOIN teaches t
  ON c.course_id = t.course_id;
```

Finding the unmatched rows

- ”Sections with **no** assigned instructor”:

```
SELECT sec.course_id, sec.sec_id
FROM   section sec
LEFT JOIN teaches t ON sec.course_id = t.course_id
                   AND sec.sec_id   = t.sec_id
                   AND sec.semester = t.semester
                   AND sec.year     = t.year
WHERE  t.id IS NULL;      -- the "anti-join" pattern
```

- Outer join + `IS NULL` = ”rows on the left with no partner.”

RIGHT and FULL outer joins

```sql
-- keep every instructor, even those teaching nothing:
SELECT i.name, t.course_id
FROM   teaches t
RIGHT JOIN instructor i ON i.id = t.id;

-- keep unmatched rows from BOTH sides:
SELECT * FROM a FULL JOIN b ON a.k = b.k;
```

- `RIGHT JOIN` = flipped `LEFT JOIN`.
- SQLite added `RIGHT`/`FULL JOIN` only in **3.39 (2022)**; older builds lack them — rewrite as a `LEFT JOIN` if needed.

Join type summary

SQL	Keeps unmatched	RA
`CROSS JOIN`	(all pairs)	$\times$
`INNER JOIN`	neither	$\bowtie$
`LEFT JOIN`	left rows	$\uplus_L$
`RIGHT JOIN`	right rows	$\uplus_R$
`FULL JOIN`	both	$\uplus_F$

Set Operations

Set operations on query results

- Combine the **outputs** of two `SELECT`s (which must be **union-compatible**: same number of columns, compatible types).

SQL	Meaning	RA
`UNION`	in either (distinct)	$\cup$
`INTERSECT`	in both	$\cap$
`EXCEPT`	in first, not second	$-$

UNION example

”All names that are a student or an instructor.”

```
SELECT name FROM student
UNION
SELECT name FROM instructor;
```

- Duplicates removed automatically.
- Column names come from the **first** SELECT.

Quick Check: UNION

Question: How do you find all unique department names that appear in either the `course` table or the `instructor` table?

Quick Check: UNION

Question: How do you find all unique department names that appear in either the `course` table or the `instructor` table?

Answer:

```
SELECT dept_name FROM course
UNION
SELECT dept_name FROM instructor;
```

UNION ALL — keep duplicates (and faster)

```
SELECT name FROM student
UNION ALL
SELECT name FROM instructor;
```

- Keeps every row, including duplicates.
- **Cheaper** — no duplicate-elimination step. Use `UNION ALL` when you know there are no duplicates, or you want the counts.

INTERSECT and EXCEPT

```sql
-- names shared by students AND instructors:
SELECT name FROM student
INTERSECT
SELECT name FROM instructor;

-- departments with instructors but NO students:
SELECT dept_name FROM instructor
EXCEPT
SELECT dept_name FROM student;
```

- `INTERSECT ALL` / `EXCEPT ALL` keep duplicate multiplicities (Postgres; SQLite supports `INTERSECT`/`EXCEPT`, not the `ALL` variants).

Quick Check: EXCEPT

Question: How would you find courses that have NEVER been taught?

Quick Check: EXCEPT

Question: How would you find courses that have NEVER been taught?

Answer:

```
SELECT course_id FROM course
EXCEPT
SELECT course_id FROM teaches;
```

Quick Check: EXCEPT

Question: How would you find courses that have NEVER been taught?

Quick Check: EXCEPT

Question: How would you find courses that have NEVER been taught?

Answer:

```
SELECT course_id FROM course
EXCEPT
SELECT course_id FROM teaches;
```

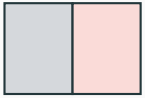
Set ops vs joins — don't confuse them

Joins

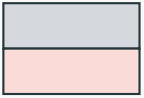
- combine **columns** (widen rows) by matching a condition.
- "Students with their department building" → `JOIN`.

Set ops

- combine **rows** of two results with the **same columns**.
- "Students who are also instructors" → `INTERSECT`.



Join



Set Ops

Pitfalls & Practice

Pitfall: accidental cross join

```sql
SELECT s.name, d.building
FROM   student s, department d;      -- forgot the WHERE!
```

- 105 rows of nonsense. Symptom: **row count explodes** (roughly $|R| \times |S|$).
- Prevention: use `JOIN ... ON` so the condition is structurally required.

Pitfall: duplicate rows from joins

- Joining to a table with **multiple matches** multiplies rows.
- A student with 3 `takes` rows appears **3 times** in `student JOIN takes`.
- If you only want the student, add `DISTINCT` or aggregate (Week 3).
- Always ask: *"is this join one-to-one, one-to-many, or many-to-many?"*

Pitfall: WHERE vs ON in outer joins

```
-- WRONG: this turns the LEFT JOIN back into an inner join
FROM section sec LEFT JOIN teaches t ON sec.course_id = t.course_id
WHERE t.semester = 'Spring';

-- RIGHT: put the filter in the ON clause
FROM section sec LEFT JOIN teaches t
     ON sec.course_id = t.course_id AND t.semester = 'Spring';
```

- A `WHERE` on the right table's column removes the NULL-padded rows.

Non-equijoins

- Join conditions aren't limited to =.

"Instructors paired with students earning… er, with a *higher* id" — or more usefully, band lookups:

```
SELECT i.name, g.band
FROM   instructor i
JOIN   salary_band g ON i.salary BETWEEN g.low AND g.high;
```

- Ranges, <, <> are all valid join predicates (a **theta join**).

Join + aggregation preview

"Number of students per department" (a taste of Week 3):

```
SELECT d.dept_name, COUNT(s.id) AS n_students
FROM   department d
LEFT JOIN student s ON s.dept_name = d.dept_name
GROUP  BY d.dept_name;
```

- `LEFT JOIN` + `COUNT(s.id)` gives `0` for empty departments (`COUNT` ignores the NULL from the outer join). Note: `COUNT(*)` would give 1.

USING vs ON — subtle output difference

```
SELECT * FROM student JOIN department USING (dept_name);   -- one dept_name col
SELECT * FROM student JOIN department
       ON student.dept_name = department.dept_name;        -- TWO dept_name col
```

- `USING` and `NATURAL JOIN` coalesce the shared column into one.
- `ON` keeps both copies — remember when you `SELECT *`.

Three-way set operation

- Set ops chain; parentheses control precedence:

```
SELECT name FROM student
UNION
SELECT name FROM instructor
EXCEPT
SELECT 'Einstein';
```

- Evaluated left-to-right unless parenthesized. `INTERSECT` binds tighter than `UNION`/`EXCEPT` in the SQL standard — parenthesize to be safe.

Emulating FULL JOIN on old SQLite

- No `FULL JOIN`? Combine two `LEFT JOIN`s with `UNION`:

```
SELECT a.k, a.x, b.y FROM a LEFT JOIN b ON a.k = b.k
UNION
SELECT b.k, a.x, b.y FROM b LEFT JOIN a ON a.k = b.k;
```

- The `UNION` removes the duplicated matched rows. Handy portability trick.

A fuller worked query

”Every course (title) and the name of any instructor who taught it — including courses taught by no one.”

```
SELECT c.title, i.name
FROM   course c
LEFT JOIN section sec ON sec.course_id = c.course_id
LEFT JOIN teaches t   ON t.course_id = sec.course_id AND t.sec_id = sec.sec_id
                     AND t.semester = sec.semester   AND t.year   = sec.year
LEFT JOIN instructor i ON i.id = t.id
ORDER BY c.title;
```

Common mistakes recap

- Comma-join without a `WHERE` $\rightarrow$ cross join.
- `NATURAL JOIN` matching an unintended column.
- Filtering the right table of a `LEFT JOIN` in `WHERE`.
- Using a join when you meant a set operation (or vice versa).
- Forgetting `DISTINCT`/aggregation after a one-to-many join.

Summary — key takeaways

- **Inner join** keeps matches; **outer joins** keep unmatched rows with NULLs.
- Prefer explicit **`JOIN ... ON`** with **aliases**; avoid `NATURAL JOIN`.
- **Self joins** relate rows within one table (two aliases).
- **Set ops** (`UNION`/`INTERSECT`/`EXCEPT`) combine same-shaped results; `ALL` keeps duplicates.
- Watch for **row explosions** and the **outer-join `WHERE` trap**.

Exercises (on `university.sql`)

1. Names of instructors and the titles of courses they teach.
2. All departments and their total number of students (0 if none) — *outer join*.
3. Pairs of students in the same department (no self-pairs, no duplicates).
4. Courses that have a prerequisite, showing both titles.
5. Names that belong to a student **but not** any instructor.

Questions?