BT5110: Tutorial 1 — Relational Model

Pratik Karmakar

School of Computing,
National University of Singapore

AY25/26 S1

What is the relational model? (in plain words)

- Think **spreadsheets**: each **table** is a sheet, **rows** are items, **columns** are properties.
- Formally, a table is a **relation**; a row is a **tuple**; a column is an **attribute**.
- Each table has a rule for what counts as a valid row: that rulebook is the **schema**.
- Tables can be **linked**—a value in one table points to a matching value in another.
- Why it's powerful: simple structure, **strong guarantees** (constraints, transactions), and **declarative** querying (SQL).

Everyday analogy

Students table

- One row per student
- Columns: `email`, `name`, `faculty`, `department`, `year`
- **Primary key** (`email`) uniquely identifies a student

Books table

- One row per book title
- Columns: `isbn13` (PK), `isbn10`, `title`, `author`, `publisher`, `year`

Copies & Loans

- `copy(owner, book, copy_no, ...)`
- `loan(owner, book, copy_no, borrowed, returned)`
- **Foreign keys** connect:
  - `copy.owner` → `student.email`
  - `copy.book` → `book.isbn13`
  - `loan.(owner, book, copy_no)` → `copy`

Keys, relationships, and constraints

- **Primary key (PK)**: a column (or set) that uniquely identifies each row.
- **Foreign key (FK)**: a column whose values must match a PK in another table.
- **Domain & CHECK constraints**: control valid values (e.g., dates, non-negative pages).
- **Entity integrity**: PK cannot be `NULL`.
- **Referential integrity**: FK must reference an existing PK (or be `NULL` if allowed).
- These rules keep data **consistent** across tables.

How we ask questions (SQL mirrors simple ops)

Core ideas

- **Filter** rows (*selection*, σ): `WHERE`
- **Pick** columns (*projection*, π): `SELECT` list
- **Match** tables (*join*, $\bowtie$): `JOIN ... ON`
- **Summarize** (*group/aggregate*): `GROUP BY`, `COUNT`, `SUM`, ...

Example questions

- "Which copies are currently on loan?"
- "Which students own at least two books?"
- "How many Chemistry students borrowed in 2024?"

Takeaway: you describe *what* you want; the database figures out *how* to get it efficiently (query optimization).

Why Relational Algebra?

- **Foundation of SQL**: Relational algebra provides the formal, mathematical basis for relational databases.
- Defines **operations on relations** that always produce relations.
- Ensures queries are **precise**, **composable**, and **optimizable**.
- SQL = (mostly) declarative syntax built on these algebraic ideas.

Core Operators

- **Selection (σ)**: Pick rows that satisfy a condition. $\sigma_{\text{year}=2024}(\text{Student})$
- **Projection (π)**: Pick certain columns. $\pi_{\text{name, email}}(\text{Student})$
- **Renaming (ρ)**: Rename relation/attributes. $\rho_S(\text{Student})$
- **Union ($\cup$), Difference ($-$), Intersection ($\cap$)**: Set-like ops (schemas must match).
- **Cartesian Product ($\times$)**: Pair all tuples across two relations.
- **Join ($\bowtie$)**: Combine rows across relations on matching attributes.

Examples and SQL Mapping

Relational Algebra

- $\pi_{\text{name}}(\sigma_{\text{faculty}=\text{'SoC'}}(\text{Student}))$
- $\text{Student} \bowtie_{\text{student.email}=\text{copy.owner}} \text{Copy}$
- $\pi_{\text{isbn13}}(\text{Book}) - \pi_{\text{book}}(\text{Copy})$

SQL Equivalent

```sql
SELECT name
FROM student
WHERE faculty = 'SoC';
```

```sql
SELECT *
FROM student s
JOIN copy c
  ON s.email = c.owner;
```

```sql
SELECT isbn13
FROM book
EXCEPT
SELECT book
FROM copy;
```

Takeaway

- Relational algebra = the **mathematical core** of querying.
- SQL extends it with ordering, grouping, aggregation, etc.
- Understanding algebra clarifies how SQL queries work "under the hood."

Case Description

Students at the National University of Ngendipura (NUN) buy, lend, and borrow books. Your company, Apasaja Private Limited, is commissioned by NUN Students Association (NUNStA) to implement an online book exchange system that records information about students, books they own, and the books they lend/borrow.

The database stores:

- **Students**: name, faculty, department, join date (year). Identifier: `email`.
- **Books**: title, authors, publisher, year/edition, `ISBN-10`, `ISBN-13`.
- **Loans**: per-copy borrow and return dates.

Auditing policy keeps book/copy/owner info while owners are students or copies have loan records; graduated students are kept if loans exist on books they owned.

Q1. Data Definition Language (DDL)

(a) Download from `Canvas ▷ Files ▷ Cases ▷ Book Exchange`:
`NUNStASchema.sql`, `NUNStAClean.sql`, `NUNStAStudent.sql`, `NUNStABook.sql`, `NUNStACopy.sql`, `NUNStALoan.sql`.

(b) What they do:

- **Clean Up**: `NUNStAClean.sql` drops tables using `IF EXISTS`.
- **Schema**: `NUNStASchema.sql` creates tables with domains/constraints using `IF NOT EXISTS`.
- **Data**: `NUNStAStudent.sql`, `NUNStABook.sql`, `NUNStACopy.sql`, `NUNStALoan.sql` populate tables (order matters).

Q1. Create & Populate — Correct Order

(c) In **pgAdmin 4**, run:

1. Create: `student`, `book`
2. Create: `copy`, `loan` (FKs depend on 1)
3. Populate: `student`, `book` (any order)
4. Populate: `copy` $\rightarrow$ `loan`

Cleanup is reverse: `loan` $\rightarrow$ `copy` $\rightarrow$ `student`, `book`.

Q2. Insert / Delete / Update — Book Inserts

(a) Insert a new book:

```
INSERT INTO book VALUES (
  'An Introduction to Database Systems',
  'paperback',
  640,
  'English',
  'C. J. Date',
  'Pearson',
  '2003-01-01',
  '0321197844',
  '978-0321197849'
);
-- Verify:
SELECT * FROM book;
```

Q2. Book Variants & Keys

(b) Same book, different `ISBN13` (unique `isbn10` violated):

```sql
INSERT INTO book VALUES (
  'An Introduction to Database Systems',
  'paperback',
  640,
  'English',
  'C. J. Date',
  'Pearson',
  '2003-01-01',
  '0321197844',
  '978-0201385908'
);
```

Q2. Book Variants & Keys

(c) Same book, different `ISBN10` (PK `isbn13` violated):

```sql
INSERT INTO book VALUES (
  'An Introduction to Database Systems',
  'paperback',
  640,
  'English',
  'C. J. Date',
  'Pearson',
  '2003-01-01',
  '0201385902',
  '978-0321197849'
);
```

Q2. Insert Students

(d) Explicit values; `year` may be `NULL`:

```sql
INSERT INTO student VALUES (
  'TIKKI TAVI',
  'tikki@gmail.com',
  '2024-08-15',
  'School of Computing',
  'CS',
  NULL
);
```

Q2. Insert Students

(e) Column list (omitting PK forces NULL and fails):

```
INSERT INTO student (email, name, year, faculty, department)
    VALUES (
  'rikki@gmail.com',
  'RIKKI TAVI',
  '2024-08-15',
  'School of Computing',
  'CS'
);

-- Fails: PK email omitted -> NULL not allowed
INSERT INTO student (name, year, faculty, department) VALUES (
  'RIKKI TAVI',
  '2024-08-15',
  'School of Computing',
  'CS'
);
```

Q2. Update & Delete

(f) Normalize department spelling:

```
UPDATE student
SET department = 'Computer Science'
WHERE department = 'CS';

-- Checks
SELECT * FROM student WHERE department = 'CS';              -- empty
SELECT * FROM student WHERE department = 'Computer Science';  -- rows
```

(g) Case-sensitive delete (no-op):

```
DELETE FROM student WHERE department = 'chemistry';
```

(h) Delete protected by FKs (likely fails unless schema permits):

```
DELETE FROM student WHERE department = 'Chemistry';
```

Q3. DEFERRABLE Constraints (Semantics)

(a) In PostgreSQL, `UNIQUE`/`PRIMARY KEY`/`FOREIGN KEY` may be:

- `NOT DEFERRABLE` (always *IMMEDIATE*),
- `DEFERRABLE INITIALLY IMMEDIATE`,
- `DEFERRABLE INITIALLY DEFERRED`.

Q3. Delete Book + Copy (Immediate vs Deferred)

Insert a copy owned by `tikki`:

```
INSERT INTO copy VALUES (
  'tikki@gmail.com',
  '978-0321197849',
  1,
  'TRUE'
);
```

Transaction #1 (IMMEDIATE) — violates FK on first delete:

```
BEGIN TRANSACTION;

SET CONSTRAINTS ALL IMMEDIATE;
DELETE FROM book WHERE ISBN13 = '978-0321197849';
DELETE FROM copy WHERE book = '978-0321197849';

END TRANSACTION;
```

Q3. Deferred Transaction (Succeeds)

Transaction #2 (DEFERRED) — combined effect OK at commit:

```
1 BEGIN TRANSACTION;
2
3 SET CONSTRAINTS ALL DEFERRED;
4 DELETE FROM book WHERE ISBN13 = '978-0321197849';
5 DELETE FROM copy WHERE book = '978-0321197849';
6
7 END TRANSACTION;
```

Check intermediate state after deleting the book (line 3):

```
-- Book is gone
SELECT * FROM book b WHERE b.ISBN13 = '978-0321197849';

-- Copy still present -> intermediate state inconsistent
SELECT * FROM copy c WHERE c.book = '978-0321197849';
```

Q4. Modifying the Schema — Drop Redundant Availability

(a) Availability is derivable: a copy is unavailable iff it has an open loan.

```
-- Unreturned loans
SELECT owner, book, copy, returned
FROM loan
WHERE returned ISNULL;

-- Remove redundant column
ALTER TABLE copy
DROP COLUMN available;
```

Q4. Modifying the Schema — Drop Redundant Availability

```sql
-- View that recomputes availability
CREATE OR REPLACE VIEW copy_view (owner, book, copy, available)
    AS (
  SELECT DISTINCT c.owner, c.book, c.copy,
    CASE
      WHEN EXISTS (
        SELECT * FROM loan l
        WHERE l.owner = c.owner
          AND l.book  = c.book
          AND l.copy  = c.copy
          AND l.returned ISNULL
      ) THEN 'FALSE'
      ELSE 'TRUE'
    END
  FROM copy c
);
```

Q4. Modifying the Schema — Drop Redundant Availability

```sql
-- Example update attempt (requires INSTEAD OF trigger/rule in
    practice)
UPDATE copy_view
SET owner = 'tikki@google.com'
WHERE owner = 'tikki@gmail.com';

-- Drop when done
DROP VIEW copy_view;
```

Q4. Normalize Department → Faculty Mapping

(b) `department` → `faculty` (FD) — store mapping once.

```sql
CREATE TABLE department (
  department VARCHAR(32) PRIMARY KEY,
  faculty    VARCHAR(62) NOT NULL
);

INSERT INTO department
SELECT DISTINCT department, faculty
FROM student;

ALTER TABLE student
DROP COLUMN faculty;

ALTER TABLE student
ADD FOREIGN KEY (department) REFERENCES department(department);
```

Questions?
Drop a mail at: pratik.karmakar@u.nus.edu