Databases for the Web — SQL & ORMs

Web Technologies — Lecture 9

Masoud Hamad

State University of Zanzibar (SUZA)

Semester II, 2025/2026

Outline

Relational vs NoSQL

	Relational (SQL)	**NoSQL**
Schema	Fixed, enforced	Flexible, often schemaless
Joins	Powerful	Limited or absent
Strength	Structured data, ACID	Scale, semi-structured data
Examples	PostgreSQL, MySQL, SQLite	MongoDB, Redis, DynamoDB
Best for	Most web apps	Caching, big data, docs

Default choice:

PostgreSQL for production, **SQLite** for development and small projects.

Creating Tables

```
CREATE TABLE students (
  id          INTEGER PRIMARY KEY AUTOINCREMENT,
  name        TEXT NOT NULL,
  email       TEXT UNIQUE NOT NULL,
  course      TEXT,
  year        INTEGER CHECK(year BETWEEN 1 AND 4),
  created_at  DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_students_course ON students(course);
```

CRUD — Create, Read, Update, Delete

```sql
-- Create
INSERT INTO students (name, email, course, year)
VALUES ('Amina Said', 'amina@suza.ac.tz', 'BSc CS', 2);

-- Read
SELECT id, name FROM students
WHERE course = 'BSc CS' AND year >= 2
ORDER BY name ASC
LIMIT 10;

-- Update
UPDATE students SET year = 3 WHERE id = 42;

-- Delete
DELETE FROM students WHERE id = 42;
```

Joins

```sql
CREATE TABLE enrolments (
  student_id INTEGER REFERENCES students(id),
  course_id  TEXT,
  grade      TEXT
);

-- INNER JOIN
SELECT s.name, e.course_id, e.grade
FROM students s
JOIN enrolments e ON e.student_id = s.id
WHERE e.grade = 'A';

-- LEFT JOIN -- include students with no enrolments
SELECT s.name, COUNT(e.course_id) AS courses
FROM students s
LEFT JOIN enrolments e ON e.student_id = s.id
GROUP BY s.id;
```

Aggregations

```
SELECT course, COUNT(*) AS total, AVG(year) AS avg_year
FROM students
GROUP BY course
HAVING COUNT(*) > 5
ORDER BY total DESC;
```

Aggregate functions: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`.

NEVER Do This

```
-- BAD -- string concatenation in your code
const sql = "SELECT * FROM users WHERE email = '" + email + "'";

-- attacker enters: ' OR '1'='1
-- result: SELECT * FROM users WHERE email = '' OR '1'='1'
-- returns ALL users
```

Always Use Parameterised Queries

```
// Node + better-sqlite3
const stmt = db.prepare("SELECT * FROM users WHERE email = ?");
const user = stmt.get(email);

// Java + JDBC
PreparedStatement ps = conn.prepareStatement(
  "SELECT * FROM users WHERE email = ?");
ps.setString(1, email);

// Python + psycopg
cur.execute("SELECT * FROM users WHERE email = %s", (email,))
```

The driver escapes values safely; SQL injection becomes impossible.

ACID & Transactions

ACID: Atomicity, Consistency, Isolation, Durability — guarantees the database makes about transactions.

```
BEGIN;
  UPDATE accounts SET balance = balance - 100 WHERE id = 1;
  UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
-- ROLLBACK if anything fails
```

Wrap any **multi-step operation that must succeed or fail as a unit** in a transaction.

What is an ORM?

ORM = Object-Relational Mapping. Map database tables to objects in your language.

- Write Java/Python/JS, not raw SQL (most of the time)
- Type-safe, refactor-friendly
- Auto-generate migrations as your schema changes
- Risk: "N+1 queries" if you're not careful

Popular ORMs

Spring Data JPA / Hibernate (Java) • Sequelize / Prisma (Node) • Django ORM / SQLAlchemy (Python) • Active Record (Rails)

Example — Prisma (Node)

```
// schema.prisma
model Student {
  id      Int      @id @default(autoincrement())
  name    String
  email   String   @unique
  course  String?
}
```

```
// usage
const students = await prisma.student.findMany({
  where: { course: "BSc CS" },
  orderBy: { name: "asc" },
  take: 10
});
const created = await prisma.student.create({
  data: { name: "Hassan", email: "h@suza.ac.tz" }
});
```

Example — Django ORM (Python)

```
# models.py
class Student(models.Model):
    name   = models.CharField(max_length=100)
    email  = models.EmailField(unique=True)
    course = models.CharField(max_length=20, null=True)

# usage
Student.objects.filter(course="BSc CS").order_by("name")[:10]
Student.objects.create(name="Hassan", email="h@suza.ac.tz")
```

Migrations

- A **migration** is a versioned change to your schema
- Stored as files in your repo → everyone gets the same DB
- Tools: Flyway/Liquibase (Java), Prisma Migrate (Node), Django migrations, Alembic (Python)
- Always commit migrations **with** the code that needs them

Indexes & Performance

- Index columns you **filter** or **join** on, not every column
- Use `EXPLAIN` to see how the DB plans your query
- Watch for **N+1**: loading 100 students then 1 query per student
- Use `JOIN` or eager-loading (`include` / `select_related`)

Summary

- Default to **relational + SQL** for web apps
- CRUD via INSERT, SELECT, UPDATE, DELETE; combine with JOIN & GROUP BY
- **Always parameterise** queries to prevent SQL injection
- Wrap multi-step changes in transactions
- ORMs map tables to objects — powerful but watch for N+1
- Schema changes go through versioned migrations

Next: Backend REST APIs — Spring Boot, Node, Django.

Questions?