CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 5: SQL DDL

Amit Kumar Dhar

IIT Bhilai

From algebra to a real language

- Relational algebra told us **what operations exist**.
- **SQL** (Structured Query Language) is how we actually talk to a DBMS.
- Declarative, ANSI/ISO-standard, spoken (with dialects) by Postgres, SQLite, MySQL, Oracle, SQL Server…
- Today: **DDL** (define schema) + **single-table SELECT**.

Agenda

- SQL sublanguages (DDL / DML / DCL / TCL)
- Data types (SQLite vs PostgreSQL)
- `CREATE / ALTER / DROP TABLE`
- Constraints: PK, FK, UNIQUE, CHECK, NOT NULL, DEFAULT

SQL Sublanguages & Types

The SQL sublanguages

- **DDL** — Data Definition: `CREATE`, `ALTER`, `DROP`.
- **DML** — Data Manipulation: `SELECT`, `INSERT`, `UPDATE`, `DELETE`.
- **DCL** — Data Control: `GRANT`, `REVOKE`.
- **TCL** — Transaction Control: `COMMIT`, `ROLLBACK`, `SAVEPOINT`.

One language, four jobs. Today = DDL + the `SELECT` part of DML.

SQL data types (the essentials)

Category	PostgreSQL	SQLite
Integer	`INTEGER`, `BIGINT`	`INTEGER`
Decimal	`NUMERIC(p,s)`, `REAL`	`NUMERIC`, `REAL`
Text	`VARCHAR(n)`, `TEXT`	`TEXT`
Date/time	`DATE`, `TIMESTAMP`	`TEXT`/`INTEGER` (no native type)
Boolean	`BOOLEAN`	`INTEGER` (0/1)

- Use portable types (`INTEGER`, `TEXT`, `NUMERIC`) in this course.

SQLite's surprise: dynamic typing

- Most DBMSs are **strictly typed** — a column's declared type is enforced.
- **SQLite uses "type affinity"** — it *prefers* a type but will store almost anything ("flexible typing").
- Practically: declare sensible types and insert sensible values; don't rely on SQLite to reject a string in an `INTEGER` column (older versions won't).
- PostgreSQL will reject it — good discipline for when you switch in Week 4.

Data Definition Language (DDL)

CREATE TABLE

```sql
CREATE TABLE department (
    dept_name  TEXT     PRIMARY KEY,
    building   TEXT,
    budget     NUMERIC  CHECK (budget > 0)
);
```

- Lists columns, their types, and **constraints**.
- The table is empty until you `INSERT`.

Column constraints

- **NOT NULL** — value required.
- **DEFAULT v** — value used if none supplied.
- **UNIQUE** — no two rows share the value.
- **CHECK (cond)** — value must satisfy a predicate.
- **PRIMARY KEY** — unique **and** not null (the row's identity).

```
credits INTEGER NOT NULL DEFAULT 3 CHECK (credits > 0)
```

Table-level constraints

- Needed for **composite** keys and multi-column checks:

```
CREATE TABLE section (
    course_id TEXT, sec_id TEXT, semester TEXT, year INTEGER,
    building TEXT, room_no TEXT,
    PRIMARY KEY (course_id, sec_id, semester, year),
    FOREIGN KEY (building, room_no)
        REFERENCES classroom (building, room_no)
);
```

Foreign keys

```
CREATE TABLE student (
    id        TEXT PRIMARY KEY,
    name      TEXT NOT NULL,
    dept_name TEXT REFERENCES department (dept_name),
    tot_cred  INTEGER CHECK (tot_cred >= 0)
);
```

- `dept_name` must match an existing `department`, or be NULL.
- SQLite: run `PRAGMA foreign_keys = ON;` per connection or FKs are **not enforced**. PostgreSQL enforces them always.

Referential actions

- Control what happens when a referenced row changes:

```
dept_name TEXT REFERENCES department(dept_name)
    ON DELETE SET NULL
    ON UPDATE CASCADE
```

- `CASCADE` — propagate the change.
- `SET NULL` / `SET DEFAULT` — blank out the reference.
- `RESTRICT` / `NO ACTION` — forbid the change (the default).

ALTER TABLE

- Evolve a schema without recreating it:

```
ALTER TABLE student ADD COLUMN email TEXT;
ALTER TABLE student RENAME COLUMN tot_cred TO credits_earned;
ALTER TABLE student ADD CONSTRAINT chk_email CHECK (email LIKE '%@%');
```

- SQLite supports a **subset** of `ALTER` (add/rename column, rename table); PostgreSQL supports the full set.

DROP vs DELETE vs TRUNCATE

- `DROP TABLE student;` — removes the **table and all its data**.
- `DELETE FROM student;` — removes **rows**, keeps the table.
- `TRUNCATE student;` — fast "remove all rows" (Postgres; not in SQLite).
- `DROP TABLE IF EXISTS student;` — safe for re-running scripts.

SQLite vs PostgreSQL — quick cheatsheet

Topic	SQLite	PostgreSQL
Typing	flexible (affinity)	strict
FK enforcement	needs `PRAGMA foreign_keys=ON`	always on
Auto id	`INTEGER PRIMARY KEY`	`SERIAL` / `GENERATED`
Show tables	`.tables`	`\dt`

Write portable SQL now; note the differences for Week 4.

Common mistakes

- Assuming SQLite enforces types/FKs like Postgres does.
- Forgetting to add `PRAGMA foreign_keys = ON;` in SQLite.
- Confusing `DROP` with `DELETE`.

Summary — key takeaways

- **DDL** defines structure: `CREATE`, `ALTER`, `DROP`.
- **Constraints** (PK, FK, UNIQUE, CHECK, NOT NULL, DEFAULT) keep data valid.
- SQLite uses **dynamic typing** (affinity), while PostgreSQL is strictly typed.

Exercises (on scratch database)

1. Create a `course` table with appropriate constraints.
2. Create a `section` table that has a foreign key to `course`.
3. Alter the `section` table to add a `capacity` column.
4. Test inserting a row that violates the foreign key constraint.

Before next lecture

- Review SQLite constraints and table creation.
- Optional reading: Silberschatz §3.2.

Next time — Lecture 6: Single-table queries, joins & set operations in SQL.

Questions?