CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 10: DML & SQL from a Program

Amit Kumar Dhar

IIT Bhilai

From querying to changing data

- So far: mostly `SELECT`. Now we **modify** data safely.
- And we stop typing SQL by hand — real apps **call SQL from code**.
- Today:
  - full **DML** (`INSERT`, `UPDATE`, `DELETE`, `UPSERT`),
  - constraints & triggers that guard changes,
  - the **Python DB-API**, parameterized queries, and **SQL injection**.

Agenda

- `INSERT` (single, multi-row, from a query)
- `UPDATE` (with `WHERE`, from other tables)
- `DELETE` (and how it differs from `DROP`/`TRUNCATE`)
- `UPSERT` (`ON CONFLICT` / `INSERT OR REPLACE`)
- Constraints revisited; **triggers**
- The Python **DB-API** (`sqlite3`)
- **Parameterized queries** & SQL injection

INSERT — single and multi-row

```sql
INSERT INTO categories (cat_id, cat_name) VALUES (7, 'Garden');

INSERT INTO categories (cat_id, cat_name) VALUES
    (8, 'Automotive'),
    (9, 'Beauty');
```

- Name the columns explicitly (order-independent, survives schema changes).
- Omitted columns take their `DEFAULT` (or `NULL`).

Quick Check: INSERT defaults

Question: If a table `users` has columns `id` (Primary Key, Auto-increment) and `name`, what happens if we execute `INSERT INTO users (name) VALUES ('Alice');`?

Quick Check: INSERT defaults

Question: If a table `users` has columns `id` (Primary Key, Auto-increment) and `name`, what happens if we execute `INSERT INTO users (name) VALUES ('Alice');`?

Answer: The database automatically assigns a new `id` value to the row because it was omitted and has an Auto-increment (default) behavior.

INSERT … SELECT

- Insert the **result of a query** — bulk-load derived data:

```
INSERT INTO archived_orders (order_id, cust_id, order_date)
SELECT order_id, cust_id, order_date
FROM   orders
WHERE  order_date < '2022-01-01';
```

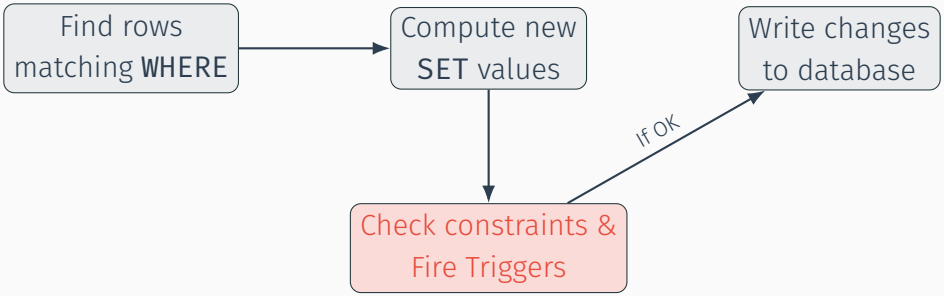
- Great for archiving, copying, and building summary tables.

UPDATE

```sql
UPDATE products
SET    price = price * 1.10
WHERE  cat_id = 1;                 -- 10% raise for Electronics
```

- SET assigns new values; WHERE limits which rows.
- **UPDATE with no WHERE changes EVERY row.** Always check your WHERE.

How UPDATE works



UPDATE from another table

```sql
-- PostgreSQL
UPDATE products p
SET    price = price * 0.9
FROM   categories c
WHERE  p.cat_id = c.cat_id AND c.cat_name = 'Books';
```

- Update using values/conditions from a joined table.
- SQLite uses a correlated subquery or (3.33+) `UPDATE ... FROM`.

Quick Check: UPDATE

Question: What is the consequence of executing `UPDATE employees SET salary = salary * 1.05;` without a `WHERE` clause?

Quick Check: UPDATE

Question: What is the consequence of executing `UPDATE employees SET salary = salary * 1.05;` without a `WHERE` clause?

Answer: Every single employee in the `employees` table will receive a 5% salary increase. The entire table is updated.

DELETE

```sql
DELETE FROM orders WHERE status = 'cancelled';
DELETE FROM orders;                  -- deletes ALL rows (table remains)
```

- Removes rows matching `WHERE`.
- vs `DROP TABLE` (removes the table) and `TRUNCATE` (fast delete-all, Postgres).
- Referential integrity may **block** a delete (or cascade it) — recall `ON DELETE` actions.

UPSERT — insert or update

- ”Insert; if it already exists, update instead.”

```
-- PostgreSQL / SQLite
INSERT INTO products (prod_id, prod_name, cat_id, price, stock)
VALUES (101, 'Wireless Mouse', 1, 799, 50)
ON CONFLICT (prod_id)
DO UPDATE SET price = EXCLUDED.price, stock = EXCLUDED.stock;
```

- `EXCLUDED` refers to the values you tried to insert.
- Older SQLite: `INSERT OR REPLACE` (blunter — deletes then inserts).

RETURNING — get back what changed

```sql
-- PostgreSQL, and SQLite >= 3.35
INSERT INTO categories (cat_id, cat_name) VALUES (10, 'Music')
RETURNING cat_id, cat_name;

UPDATE products SET stock = stock - 1 WHERE prod_id = 101
RETURNING stock;
```

- Avoids a second round-trip to read the new values — very handy from code.

Constraints guard every DML

- Recall: PK, FK, UNIQUE, CHECK, NOT NULL.
- Any `INSERT`/`UPDATE` violating a constraint is **rejected** — the DBMS protects data integrity so your app code doesn't have to.

```
INSERT INTO products (prod_id, prod_name, cat_id, price, stock)
VALUES (101, 'Dup', 1, -5, 0);      -- fails: duplicate PK AND price CHECK
```

- Let the database enforce rules; don't reinvent them in every program.

Triggers — code that runs on DML

- A **trigger** runs automatically **before/after** an `INSERT`/`UPDATE`/`DELETE`.
- Uses: auditing, derived columns, complex validation, maintaining summaries.

```
-- SQLite: keep an audit log of deletions
CREATE TRIGGER log_deletes
AFTER DELETE ON orders
BEGIN
    INSERT INTO order_audit(order_id, deleted_at)
    VALUES (OLD.order_id, datetime('now'));
END;
```

Trigger concepts

- **Timing:** `BEFORE` (validate/modify) vs `AFTER` (react/log).
- **Event:** `INSERT`, `UPDATE`, `DELETE`.
- **`OLD` / `NEW`:** the row before/after the change.
- Postgres triggers call a **function**; SQLite inlines the body.
- Use sparingly — hidden logic in triggers can surprise maintainers.

Quick Check: Triggers

Question: Inside an `UPDATE` trigger, what do the pseudorecords `OLD` and `NEW` represent?

Quick Check: Triggers

Question: Inside an `UPDATE` trigger, what do the pseudorecords `OLD` and `NEW` represent?

Answer: `OLD` represents the state of the row before the update, and `NEW` represents the new values being applied to the row.

Why call SQL from a program?

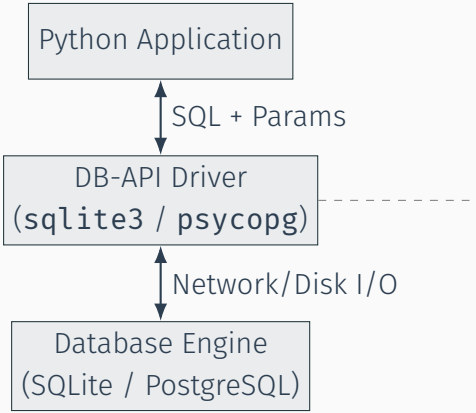
- Applications need SQL embedded in application logic (loops, conditionals, UI).
- Every language has a database interface; Python's is the **DB-API (PEP 249)**.
- `sqlite3` is in the **standard library** — nothing to install.
- The same patterns apply to PostgreSQL via `psycopg` (Lab 4).

The DB-API workflow

```
import sqlite3

conn = sqlite3.connect("store.db")     # 1. connect
cur  = conn.cursor()                   # 2. cursor
cur.execute("SELECT prod_name, price FROM products WHERE cat_id = ?", (1,))
rows = cur.fetchall()                  # 3. run + fetch
for name, price in rows:
    print(name, price)
conn.commit()                          # 4. commit (for writes)
conn.close()                           # 5. close
```

DB-API Architecture



Cursor: execution context

Connection: session & transaction

Connection and cursor

- **Connection** — a session with the database; owns transactions.
- **Cursor** — runs statements and iterates over results.
- `execute(sql, params)` — one statement.
- `executemany(sql, seq)` — the same statement over many parameter tuples.
- `fetchone()` / `fetchall()` / iterate the cursor for rows.

Reading results

```
cur.execute("SELECT prod_name, price FROM products")
row = cur.fetchone()          # a single tuple, or None
all_rows = cur.fetchall()     # list of tuples
for r in cur:                 # cursors are iterable (memory-friendly)
    print(r)
```

- For named access, set `conn.row_factory = sqlite3.Row` → `row["price"]`.

Writing data from code

```
cur.execute(
    "INSERT INTO categories (cat_id, cat_name) VALUES (?, ?)",
    (11, "Stationery"),
)
conn.commit()          # writes are not durable until commit
```

- Nothing is saved until `commit()`.
- `conn.rollback()` discards uncommitted changes (transactions — Lecture 12).

Quick Check: Commits

Question: You run `cur.execute("DELETE FROM users WHERE id=5")` in Python but forget to call `conn.commit()`. What happens?

Quick Check: Commits

Question: You run `cur.execute("DELETE FROM users WHERE id=5")` in Python but forget to call `conn.commit()`. What happens?

Answer: The deletion is not saved permanently. When the script exits or connection closes, the uncommitted transaction is rolled back, and the row remains in the database.

executemany — bulk insert

```
data = [(12, "Pet"), (13, "Grocery"), (14, "Health")]
cur.executemany("INSERT INTO categories (cat_id, cat_name) VALUES (?, ?)", data
conn.commit()
```

- One statement, many rows — far faster than a Python loop of `execute`.

The wrong way: string formatting

```
name = input("category name: ")
cur.execute(f"INSERT INTO categories(cat_name) VALUES ('{name}')")  # DANGER
```

- Building SQL by **string concatenation/f-strings** is dangerous.
- If **name** contains SQL, it becomes **part of the command**.
- This is **SQL injection** — one of the most common security flaws.

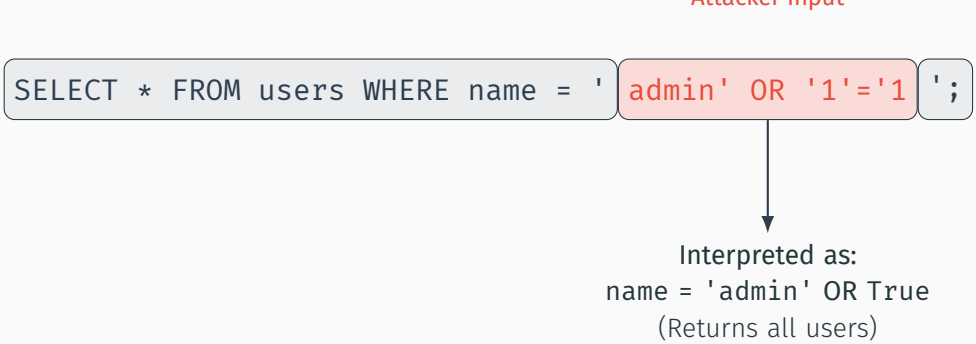
SQL injection — the classic attack

- App builds: `... WHERE name = '<user input>'`.
- User enters: `x'; DROP TABLE categories; --`
- Resulting SQL:

```
SELECT * FROM categories WHERE name = 'x'; DROP TABLE categories; --'
```

- The attacker's statement runs. Data is read, altered, or destroyed.
- xkcd's "Bobby Tables" — `Robert'); DROP TABLE Students;--`.

SQL Injection Mechanism



The fix: parameterized queries

```
name = input("category name: ")
cur.execute("SELECT * FROM categories WHERE cat_name = ?", (name,))
```

- The **?** (SQLite) / **%s** (psycopg) is a **placeholder**.
- The driver sends SQL and **data separately**; user input is **never** parsed as SQL. Injection becomes impossible.
- **Rule: never interpolate user input into SQL strings. Always parameterize.**

Quick Check: Parameterized Queries

Question: Why doesn't parameterization work for dynamic table or column names, e.g., `execute("SELECT ? FROM users", ("age",))`?

Quick Check: Parameterized Queries

Question: Why doesn't parameterization work for dynamic table or column names, e.g., `execute("SELECT ? FROM users", ("age",))`?

Answer: Placeholders are strictly for literal **values**, not identifiers (table/column names). The driver safely escapes them as strings, so the query would literally select the string `'age'` instead of the column `age`.

Placeholders across drivers

Driver	Placeholder	Example
`sqlite3`	`?`	`execute("... WHERE id = ?", (x,))`
`sqlite3` (named)	`:name`	`execute("... = :id", {"id": x})`
`psycopg` (Postgres)	`%s`	`execute("... = %s", (x,))`

- Placeholders are for **values**, not identifiers (table/column names).
- Validate identifiers against an allow-list if they must be dynamic.

Error handling & context managers

```
import sqlite3
try:
    with sqlite3.connect("store.db") as conn:     # commits on success,
        conn.execute("UPDATE products SET stock = stock - 1 WHERE prod_id = ?",
                     (101,))                        # rolls back on exception
except sqlite3.Error as e:
    print("DB error:", e)
```

- The `with` block manages the transaction; exceptions trigger a rollback.
- Catch `sqlite3.Error` / `psycopg.Error` — don't let DB errors crash silently.

Putting it together (mini pattern)

```
def add_product(conn, pid, name, cat, price, stock):
    conn.execute(
        """INSERT INTO products (prod_id, prod_name, cat_id, price, stock)
           VALUES (?, ?, ?, ?, ?)""",
        (pid, name, cat, price, stock))
    conn.commit()
```

- Small functions, parameterized SQL, explicit commit — the Lab 4 blueprint.

Bulk loading data

- For large loads, row-by-row `INSERT` is slow. Use bulk paths:

```
-- PostgreSQL: fast bulk load from a file
COPY products FROM '/path/products.csv' WITH (FORMAT csv, HEADER true);
```

```
# SQLite CLI
.mode csv
.import products.csv products
```

- Wrap many inserts in **one transaction** — commit once, not per row.

Batching & commit frequency

```
conn.execute("BEGIN")
for row in big_iterable:
    conn.execute("INSERT INTO t VALUES (?, ?)", row)
conn.commit()          # one commit for the whole batch
```

- Each `commit()` forces a durable write (`fsync`) — expensive.
- Committing once per 10^4–10^5 rows is dramatically faster than once per row.
- Balance: bigger batches = faster, but more to redo if it fails.

fetchmany — streaming large results

```
cur.execute("SELECT * FROM order_items")
while True:
    batch = cur.fetchmany(1000)
    if not batch:
        break
    process(batch)
```

- `fetchall()` loads **everything** into memory — bad for huge results.
- `fetchmany(n)` / iterating the cursor streams in chunks.

Auto-generated keys from code

```
cur.execute("INSERT INTO categories (cat_name) VALUES (?)", ("Music",))
new_id = cur.lastrowid          # SQLite: the new INTEGER PRIMARY KEY
conn.commit()
```

- SQLite: `cur.lastrowid`. PostgreSQL: use `RETURNING id` and `fetchone()`.
- Don't hard-code ids in real apps — let the DB assign them.

Prepared statements & reuse

- A parameterized statement can be **prepared once** and executed many times.
- The driver/DBMS caches the parse/plan → faster repeated execution.
- `executemany` leverages this. In Postgres, `PREPARE`/`EXECUTE` make it explicit.
- Bonus: prepared + parameterized = the injection-safe path *and* the fast path.

Transactions and DML (preview)

```sql
BEGIN;
UPDATE accounts SET bal = bal - 100 WHERE id = 'A';
UPDATE accounts SET bal = bal + 100 WHERE id = 'B';
COMMIT;          -- both, or (on ROLLBACK) neither
```

- Multiple DML statements often must succeed **together**.
- That ”all-or-nothing” guarantee is the transaction — full treatment in Lecture 12.

MERGE / ON CONFLICT recap

- Two ways to ”insert-or-update”:
  - `INSERT ... ON CONFLICT ... DO UPDATE` (Postgres, SQLite) — recommended.
  - `MERGE` (SQL standard; Postgres 15+, not SQLite) — richer multi-action.
- Prefer `ON CONFLICT` for portability across the two engines we use.

Common mistakes

- `UPDATE`/`DELETE` without `WHERE` → whole-table change.
- Forgetting `commit()` → changes vanish on close.
- Building SQL with f-strings/`%` → injection.
- Using placeholders for table names (they don't work there).
- Ignoring DB exceptions.

Summary — key takeaways

- DML: `INSERT` (incl. `... SELECT`), `UPDATE`, `DELETE`, `UPSERT` (`ON CONFLICT`), `RETURNING`.
- Constraints & **triggers** enforce integrity on every change.
- The **DB-API**: connect → cursor → execute → fetch/commit → close.
- **Always parameterize** — it's the fix for SQL injection *and* cleaner code.

Exercises

1. Insert 3 new products with `executemany`; commit and verify.
2. Give all 'Books' a 5% discount; show affected rows via `RETURNING`.
3. Write an UPSERT that sets stock to 100 for an existing product.
4. Write a Python function that searches products by a user-supplied name — **parameterized**. Then show why the f-string version is unsafe.

Quick reference — DML & DB-API

```
INSERT INTO t (a,b) VALUES (1,2);          INSERT INTO t SELECT ...;
UPDATE t SET a = a+1 WHERE id = 5;         DELETE FROM t WHERE ...;
INSERT ... ON CONFLICT (k) DO UPDATE SET ...;   -- upsert
... RETURNING id;                           -- get changed values back
```

```
conn = sqlite3.connect(db); cur = conn.cursor()
cur.execute("... WHERE x = ?", (val,))       # ALWAYS parameterize
rows = cur.fetchall(); conn.commit(); conn.close()
```

Security checklist for DB code

- ✓ Every user value passes through a **placeholder** (`?` / `%s`).
- ✓ No f-strings / `+` building SQL from input.
- ✓ Least-privilege DB account for the app (Lecture 11).
- ✓ Errors caught; transactions rolled back on failure.
- ✓ Dynamic table/column names validated against an **allow-list**.

Before next lecture

- Ensure PostgreSQL + `psycopg` work (Lab 4 uses both) — see `infrastructure.md`.
- Optional reading: Silberschatz §4.3 (DML), §5.3 (triggers); Python `sqlite3` docs.

Next time — Lecture 11: views (updatable/materialized), stored functions, and access control (`GRANT`/`REVOKE`).

Questions?