Introduction to SQLite

CIS 6930 - Data Engineering with LLMs

Bonus Material - Week 4

What is SQLite?

SQLite is a **self-contained, serverless** relational database.

- No separate server process needed
- Entire database stored as a single file on disk
- Pre-installed on most Linux distributions and macOS
- The most widely deployed database engine in the world

```
# Check if sqlite3 is installed
sqlite3 --version
```

```
# Install on Ubuntu (GCP VM)
sudo apt install -y sqlite3
```

sqlite.org

SQLite vs Other Databases

Feature	SQLite	PostgreSQL / MySQL
Server	None (file-based)	Separate server process
Setup	Zero configuration	Requires installation and config
Concurrency	Single writer	Multiple concurrent writers
Scale	Small to medium datasets	Large-scale production
Use case	Prototyping, local storage, embedded	Web apps, multi-user systems

SQLite is ideal for local data engineering work. You can always migrate to a full database later.

Creating a Database

Start `sqlite3` with a filename to create or open a database.

```
$ sqlite3 weather.db
SQLite version 3.45.1
Enter ".help" for usage hints.
sqlite>
```

The file `weather.db` is created if it does not exist.

Useful dot-commands:

Command	Description
`.help`	List all commands
`.tables`	Show all tables
`.schema`	Show CREATE statements
`.mode`	Set output format (column, csv, json)
`.quit`	Exit sqlite3

SQLite Data Types

SQLite uses a small set of storage classes.

Type	Description	Examples
`TEXT`	String data	`'Gainesville'`, `'2026-02-01'`
`INTEGER`	Whole numbers	`42`, `-7`, `0`
`REAL`	Floating point	`29.6520`, `3.14`
`BLOB`	Binary data	Images, files
`NULL`	Missing value	`NULL`

SQLite is flexible with types. A `TEXT` column can hold an integer. This is different from PostgreSQL or MySQL, which enforce types strictly.

Creating a Table

```sql
CREATE TABLE IF NOT EXISTS cities (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    state TEXT NOT NULL,
    lat REAL,
    lon REAL
);
```

- `INTEGER PRIMARY KEY` is an alias for SQLite's hidden `rowid` column
- `rowid` auto-increments when you omit `id` during INSERT
- `NOT NULL` prevents missing values in required columns
- `IF NOT EXISTS` avoids errors when the table already exists

Inserting Data

```
INSERT INTO cities (name, state, lat, lon)
VALUES ('Gainesville', 'Florida', 29.6520, -82.3250);

INSERT INTO cities (name, state, lat, lon)
VALUES ('Miami', 'Florida', 25.7617, -80.1918);

INSERT INTO cities (name, state, lat, lon)
VALUES ('Orlando', 'Florida', 28.5383, -81.3792);
```

Verify the data:

```
sqlite> SELECT * FROM cities;
1|Gainesville|Florida|29.652|-82.325
2|Miami|Florida|25.7617|-80.1918
3|Orlando|Florida|28.5383|-81.3792
```

The `id` column was assigned automatically.

Better Output Formatting

The default pipe-separated output is hard to read. Use `.mode` to change it.

```
sqlite> .mode column
sqlite> .headers on
sqlite> SELECT * FROM cities;

id  name         state    lat      lon
--  -----------  -------  -------  --------
1   Gainesville  Florida  29.652   -82.325
2   Miami        Florida  25.7617  -80.1918
3   Orlando      Florida  28.5383  -81.3792
```

Other useful modes: `csv`, `json`, `markdown`, `table`

Basic Queries

```sql
-- Select specific columns
SELECT name, lat, lon FROM cities;

-- Filter with WHERE
SELECT name FROM cities WHERE state = 'Florida';

-- Sort results
SELECT name, lat FROM cities ORDER BY lat DESC;

-- Limit results
SELECT name FROM cities LIMIT 2;
```

Aggregate Functions

```
-- Count rows
SELECT COUNT(*) FROM cities;

-- Average latitude
SELECT AVG(lat) AS avg_latitude FROM cities;

-- Min and max
SELECT MIN(lat) AS southernmost, MAX(lat) AS northernmost
FROM cities;

-- Group by
SELECT state, COUNT(*) AS num_cities
FROM cities
GROUP BY state;
```

Creating a Second Table

```sql
CREATE TABLE IF NOT EXISTS temperatures (
    id INTEGER PRIMARY KEY,
    city_id INTEGER,
    recorded_at TEXT,
    celsius REAL,
    FOREIGN KEY (city_id) REFERENCES cities(id)
);
```

```sql
INSERT INTO temperatures (city_id, recorded_at, celsius)
VALUES (1, '2026-02-01 08:00', 12.5);

INSERT INTO temperatures (city_id, recorded_at, celsius)
VALUES (1, '2026-02-01 14:00', 18.3);

INSERT INTO temperatures (city_id, recorded_at, celsius)
VALUES (2, '2026-02-01 08:00', 22.1);

INSERT INTO temperatures (city_id, recorded_at, celsius)
VALUES (3, '2026-02-01 08:00', 16.7);
```

JOIN: Combining Tables

```sql
SELECT c.name, t.celsius, t.recorded_at
FROM temperatures t
JOIN cities c ON t.city_id = c.id;
```

```
name         celsius   recorded_at
-----------  -------   ----------------
Gainesville  12.5      2026-02-01 08:00
Gainesville  18.3      2026-02-01 14:00
Miami        22.1      2026-02-01 08:00
Orlando      16.7      2026-02-01 08:00
```

JOINs are how relational databases combine data from multiple tables. This is a core operation in data integration.

Aggregate Query with JOIN

```sql
SELECT c.name,
       AVG(t.celsius) AS avg_temp,
       MIN(t.celsius) AS min_temp,
       MAX(t.celsius) AS max_temp
FROM temperatures t
JOIN cities c ON t.city_id = c.id
GROUP BY c.name
ORDER BY avg_temp DESC;
```

```
name         avg_temp  min_temp  max_temp
-----------  --------  --------  --------
Miami        22.1      22.1      22.1
Orlando      16.7      16.7      16.7
Gainesville  15.4      12.5      18.3
```

Exporting Data

```
# Export to CSV
sqlite3 weather.db <<EOF
.mode csv
.headers on
.output cities.csv
SELECT * FROM cities;
.quit
EOF
```

```
# Export to JSON
sqlite3 weather.db <<EOF
.mode json
.output cities.json
SELECT * FROM cities;
.quit
EOF
```

You can also import CSV files with `.import filename.csv tablename`.

SQLite from Python

Python: Connect and Create

```python
import sqlite3

# Connect to database (creates file if needed)
conn = sqlite3.connect('weather.db')
cursor = conn.cursor()

# Create table
cursor.execute("""
    CREATE TABLE IF NOT EXISTS cities (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        state TEXT NOT NULL,
        lat REAL,
        lon REAL
    )
""")

# Insert with parameterized query (prevents SQL injection)
cursor.execute(
    "INSERT INTO cities (name, state, lat, lon) VALUES (?, ?, ?, ?)",
    ("Gainesville", "Florida", 29.652, -82.325)
)
conn.commit()
```

Always use `?` placeholders. Never use f-strings or `.format()` for SQL.

Python: Query and Fetch

```python
# Fetch all rows
cursor.execute("SELECT * FROM cities")
rows = cursor.fetchall()
for row in rows:
    print(row)  # (1, 'Gainesville', 'Florida', 29.652, -82.325)

# Fetch one row
cursor.execute("SELECT * FROM cities WHERE name = ?",
               ("Miami",))
row = cursor.fetchone()

# Use as iterator
for row in cursor.execute("SELECT name, lat FROM cities"):
    print(f"{row[0]}: {row[1]}")
```

```python
# Always close when done
conn.close()
```

Python: Using pandas

pandas can read from and write to SQLite directly.

```python
import pandas as pd
import sqlite3

conn = sqlite3.connect('weather.db')

# Read a query into a DataFrame
df = pd.read_sql_query("SELECT * FROM cities", conn)
print(df)
```

```python
# Write a DataFrame to a SQLite table
df.to_sql('cities_backup', conn,
          if_exists='replace', index=False)
```

```python
# Aggregate queries work too
df = pd.read_sql_query("""
    SELECT state, COUNT(*) AS n, AVG(lat) AS avg_lat
    FROM cities GROUP BY state
""", conn)
```

Why SQLite for This Course?

SQLite is useful as a **local data store** in data engineering pipelines.

- Store extracted data (LLM outputs, API results) before integration
- Test SQL queries locally before running on a production database
- Prototype data warehousing workflows on your laptop or GCP VM
- No server setup or configuration required

This week we discuss **data warehousing**: extracting data from multiple sources, transforming it, and loading it into a central repository. SQLite can serve as a simple local warehouse for development and testing.

sqlite.org/whentouse.html