

# LECTURE 28

## SQLITE

MCS 275 Spring 2023

Emily Dumas

# POPULAR SQL DATABASES

- MySQL (open source)
- PostgreSQL (open source)
- Oracle
- Microsoft SQL Server
- IBM DB2
- SQLite

# USING SQLITE

## Method 1: From a Python script

```
import sqlite3
con = sqlite3.connect("mydbfile.name") # often .db or .sqlite
res = con.execute("SELECT * FROM evil_plans WHERE year=2023;")
print(res.fetchall()) # or iterate over res in a for loop
con.close()
```

## Method 2: Run sqlite command line shell and type

```
.open "mydbfile.name" -- Alternative to command line arg
SELECT * FROM evil_plans WHERE year=2023;
```

Today, we'll mostly practice making queries and learning more of SQL.

# SAMPLE DATABASES

`solarsystem.sqlite` — Planets orbiting the sun.

`powerplants.sqlite` — Information about approximately 35,000 power plants around the world (derived from the `Global Power Plant Database`).

# SQLITE COMMAND LINE SHELL

Useful commands specific to the shell:

- `.open FILENAME` - Use a certain DB
- `.tables` - List tables in this DB
- `.schema TABLENAME` - Show table columns
- `.quit` - Exit sqlite3
- `.headers on` - Put column headings in output
- `.mode columns` - Pretty output
- `.mode box` - Even prettier output

# POWERPLANTS TABLE

- **id** - An integer uniquely identifying the row
- **gppd\_id** - Alphanumeric code identifying this plant in the GPPD
- **country** - Name of country in which plant is located
- **name** - Name of the power plant
- **capacity\_mw** - Generation capacity in megawatts (MW)
- **latitude, longitude** - Location
- **primary\_fuel** - Name of primary fuel or generation type (e.g. Nuclear, Solar, Oil)
- **secondary\_fuel** - If plant has multiple generation methods
- **year\_commissioned** - Year the plant was commissioned
- **owner** - Name of organization that owns the plant, if known
- **output\_gwh\_XXXX** - Gigawatt-hours (GWH) of output in the year XXXX, available for 2013-2019. (1MW × 1 year = 8.77 GWH.)

# SELECT

Find and return rows. The most common query.

```
SELECT * FROM table_name; -- give me everything
SELECT * FROM table_name WHERE condition; -- some rows
SELECT col3, col1 FROM table_name; -- some columns
SELECT * FROM table_name LIMIT 10; -- at most 10 rows

SELECT * FROM table_name
ORDER BY col2; -- sort by col2, smallest first

SELECT * FROM table_name
ORDER BY col2 DESC; -- sort by col2, biggest first
```

Conditions can be e.g. equalities and inequalities.

WHERE, ORDER BY, LIMIT can be used together, but must appear in that "WOBL" order. ([Details.](#))



# SQL CONDITIONS

Examples of things that can appear after WHERE:

```
col = value    -- Also supports >, >=, <, <=, !=
col IN (val1, val2, val3)
col BETWEEN lowval AND highval
col IS NULL
col IS NOT NULL
stringcol LIKE pattern    -- string pattern matching
condition1 AND condition2
condition1 OR condition2
```

# LIKE

```
coursetitle LIKE "Introduction to %"  
itemtype LIKE "electrical adapt_r"
```

In a pattern string:

- % matches any number of characters (including 0)
- \_ matches any single character

e.g. "%d\_g" matches "fossil dig" and "dog" but does not match "hypersonic drag", "dog toy", or "dg".

# GETTING DATA FROM SQLITE

After `SELECT`, where are the data?

`execute()` doesn't return the rows directly. It returns a **Cursor** object which is ready to give them to you.

To request rows from a Cursor `c`, several options:

- Use it as an iterable (it yields one tuple per row).
- `c.fetchone()` returns next row as a tuple.
- `c.fetchall()` returns a list of tuples.

# CREATE TABLE

Creates a table. The set of tables doesn't change very often in most databases, and this setup step is often performed manually or by a separate program.

```
CREATE TABLE [IF NOT EXISTS] table_name (  
    col1 TYPE1 [MODIFIERS],  
    col2 TYPE2 [MODIFIERS], ...  
); -- or you could write it all on one line!
```

Types include: TEXT, REAL, INTEGER

Modifiers include: UNIQUE, NOT NULL, PRIMARY KEY, and DEFAULT [val]

# REMINDER

Creating a table twice generates an error unless `IF NOT EXISTS` is given.

# PRIMARY KEY

A unique identifier for each row. Recommended to use `INTEGER PRIMARY KEY` as type.

Useful to uniquely refer to a row in an `UPDATE` or `DELETE` query.

If you don't include one, then SQLite makes one and keeps it hidden.

# INSERT INTO ... VALUES

Add one row to an existing table.

```
-- Set every column (need to know column order!)  
INSERT INTO table_name  
VALUES ( val1, val2, val3, val4, val5, val6, val7 );  
  
-- Set some columns, in an order I specify  
INSERT INTO table_name ( col1, col7, col3 )  
VALUES ( val1, val7, val3 );
```

Missing columns are set to default values (often null).

Exceptions indicate constraint violations (e.g. typing).

There is also a way to insert many rows at once, taken from the result of another query.

# GIVING DATA TO SQLITE

Don't use string formatting to embed data in a call to `execute()`. Instead, use `?` characters as placeholders and then give a tuple of values in the second argument.

```
# do this instead; it keeps data in native types
# separate from the SQL code
con.execute(
    "INSERT INTO planets VALUES (?, ?, ?);",
    ("Earth", 1.0, None)
)
```



# UPDATE

Change values in a row (or rows).

```
UPDATE table_name SET col1=val1, col5=val5 WHERE condition;
```

Warning: Every row meeting the condition is changed!

Also supports ORDER BY and LIMIT.

Use ? placeholders for values when executing from Python.

# DELETE

Remove rows matching a condition.

```
DELETE FROM table_name WHERE condition;
```

Also supports ORDER BY and LIMIT (e.g. to remove  $n$  rows with largest values in a given column).

Immediate, irreversible.

Omit WHERE clause to delete all rows.

# DROP TABLE

Deletes an entire table.

```
DROP TABLE table_name;           -- no such table = ERROR  
DROP TABLE IF EXISTS table_name; -- no such table = ok
```

Immediate, irreversible. Think of it as "throw the only copy of this table into a pool of lava". Use caution.

# TRANSACTION CONTEXT MANAGER

You can use a sqlite3 Connection object as a context manager (i.e. in `with`) to create a **transaction**.

```
with con:
    # Make all the changes necessary to reflect the closing
    # of the Scranton office.
    con.execute("UPDATE...")
    con.execute("UPDATE...")
```

Another connection to the same database will never see it in a state other than "everything in the transaction happened" (if no exceptions) or "nothing in the transaction happened" (if an exception occurs).

# REFERENCES

- [SQLite home page](#)
- [sqlitetutorial.net](#) has a nice tutorial where you can run SQL command directly in your browser. Their SQLite install instructions are detailed and easy to follow, too.
- [Intro to Python for Computer Science and Data Science](#) by Deitel and Deitel, Section 17.2. (This is an O'Reilly book, free for anyone with a UIC email; see course page for login details.)
- [Getting Started with SQL](#) by Thomas Nield is a nice introduction to SQL that focuses on SQLite. It's another O'Reilly book you can access with your UIC email.
- *Computer Science: An Overview* by Brookshear and Brylow, Chapter 9.

# REVISION HISTORY

- 2022-04-15 Last year's lecture on this topic finalized
- 2023-03-28 Updated for 2023

