LECTURE 31

SQL AND SQLITE

MCS 275 Spring 2021
Emily Dumas

LECTURE 31: SQL AND SQLITE

Course bulletins:

- Four quizzes left (11,12,13,14). The last is due on Tuesday April 27.
- Project 4 will be due on Friday April 30. The project description will be available by April 12, and the autograder by April 19.
- There are no course activities during final exam week; after April 30, you're done!

SQLITE

In lecture, I'll be using this RDBMS in two ways.

- `sqlite3` module — You need this. You already have it, because it is part of the Python standard library.
- The command line shell `sqlite3.exe` — Optional. I'll use it to run SQL commands, which you could also do from a Python program.

SQLITE COMMAND LINE SHELL INSTALLATION

- Windows — Download the "sqlite-tools" zip file for windows from the sqlite download page. You want the tools, NOT the DLL. Unzip it to find an executable called "sqlite.exe". You can double-click to run it, or put it somewhere you can find in the terminal and run it from powershell.
- MacOS — You already have it as `/usr/bin/sqlite3`. You can probably just type `sqlite3` in a terminal.
- Linux — You may already have it (try `sqlite3` in a terminal), otherwise check your distro package manager for a package that installs it. In Debian and Ubuntu that package is called `sqlite3`.

Then you use it as: `sqlite3[.exe] DBFILENAME`.

SQLITE HELLO WORLD

Let's write a Python program to make SQLite database, add one table to it, add a couple of rows of data to the table, then read them back.

CONNECTING TO A DATABASE

In `sqlite3`, opening a "connection" means opening or creating a database file.

```
import sqlite3
con = sqlite3.connect("solarsystem.sqlite")  # .db also popula
con.execute( ...sql_statement_goes_here... )
con.commit() # Save any changes to disk
con.close()  # Close the database file
```

DIFFERENCES WITH OTHER RDBMS

Python has a standard interface (DB-API) for database modules, which `sqlite3` follows. So you can almost use these code examples with MySQL, PostgreSQL, or others.

However, we've used one non-standard feature specific to the `sqlite3` module.

In other DB-API modules, you cannot call `.execute()` on a connection object directly. Instead you need to build a "cursor", e.g.

```
import sqlite3
con = sqlite3.connect("solarsystem.sqlite")
cur = con.cursor() # thing that can execute commands
cur.execute( ...sql_statement_goes_here... )
con.commit()  # It's still the connection that commits
con.close()   # and the connection that is closed
```

`sqlite3` offers a `.execute()` method directly on connection objects as a convenience.

OUR FIRST SQL

```
CREATE TABLE planets (
    name TEXT,  -- name shouldn't be highlighted there
    dist REAL,
    year_discovered INTEGER
);
INSERT INTO planets VALUES ("Earth", 1.0, null);
INSERT INTO planets VALUES ("Neptune", 30.1, 1846);
SELECT * FROM planets; -- returns all the rows
```

Creates a table with three columns. Two rows are added. Then we ask for all of the rows from that table.

More on SQL commands later!

PLANETS

Name	Distance from sun (AU)	Year discovered
Mercury	0.4	?
Venus	0.7	?
Earth	1	?
Mars	1.5	?
Jupiter	5.2	?
Saturn	9.5	?
Uranus	19.2	1781
Neptune	30.1	1846

* Data for the solar system. If you are attending MCS 275 remotely from another star system, you may subtitute local data.

GETTING DATA FROM SQLITE

`con.execute("SELECT ...")` doesn't return the rows directly. It returns a **Cursor*** object which is ready to give you those rows upon request.

It is iterable, giving rows as tuples. Alternatively:

`Cursor.fetchone` gets the next row (or None).

`Cursor.fetchall` gets a list of all the result rows.

* Database cursors are a whole separate topic, but for our purposes I suggest mentally replacing the name Cursor with ResultRowsIterable.

GIVING DATA TO SQLITE

To pass values to a statement in `execute()`, use `?` characters as placeholders and then give a tuple of values in the second argument.

```
# works, but don't do this*
con.execute("INSERT INTO planets VALUES (\"Earth\", 1.0, null)

# do this instead
con.execute(
    "INSERT INTO planets VALUES (?,?,?);",
    ("Earth", 1.0, None)
)
```

* Relevant XKCD comic.

SQL STATEMENTS

Now I'll talk about and demonstrate more SQL features and syntax, focusing on the most important statements—SELECT, INSERT, UPDATE, DELETE, CREATE TABLE, and DROP TABLE.

I'll work in the command line shell using a sample database* of stars.

- SQL syntax (SQLite dialect):
  - sqlite language docs (nice syntax diagrams)
  - sqlitetutorial.net
- `sqlite3` Python module (and some SQL):
  - sqlite3 module docs
  - Deitel and Deitel Section 17.2 (O'Reilly text)

CREATE TABLE

```
CREATE TABLE 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`

INSERT

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 );
```

Values for columns not specified will be set to null (or autogenerated, if a primary key).

Can fail in various ways (e.g. type mismatch, null value in NOT NULL column; primary key value duplicates existing row).

SELECT

Find and return rows.

```
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"`.

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.

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.

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.

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.)
- Computer Science: An Overview by Brookshear and Brylow, Chapter 9.

REVISION HISTORY

- 2021-03-31 Typos fixed
- 2021-03-31 Initial publication