ChatGPT

# Concurrent SQLite Queries Tutorial (Beginner)

**SQLite** is a lightweight, serverless SQL database engine that stores the entire database in a single file [1]. Unlike other databases, it doesn't require a separate server process – you just connect directly to a file on disk. In Python, the `sqlite3` module (built into the standard library) provides a DB-API interface to SQLite [2]. You can use it to create or open a database file, create tables, and run SQL statements (`SELECT`, `INSERT`, etc.). For example:

```python
import sqlite3
conn = sqlite3.connect('example.db')  # opens or creates example.db
cursor = conn.cursor()               # cursor for executing SQL
cursor.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT)')
conn.commit()                        # save changes
conn.close()                         # close connection
```

In this code, `connect()` opens the database (creating `example.db` if needed) [3]. We create a table with an `id` column marked as `PRIMARY KEY`, meaning each row has a unique identifier [4]. After executing SQL, we call `conn.commit()` to save changes [5], then `conn.close()` when done.

## The `exe_sql_qry` Function Explained

The script's `exe_sql_qry(query)` function likely does these steps for each SQL query:

- **Open a connection:** `conn = sqlite3.connect('mydb.db')` opens the database file (or creates it) [3].
- **Create a cursor:** `cursor = conn.cursor()` gives a cursor object to execute SQL.
- **Execute the query:** `cursor.execute(query)` runs the SQL string. For `SELECT`, it fetches rows; for `INSERT/UPDATE`, it prepares changes.
- **Commit (for modifications):** After `INSERT`/`UPDATE`, call `conn.commit()` so changes are saved to disk [5] [6]. (If you forget `commit()`, new data isn't actually stored in the file.)
- **Fetch results:** If it's a `SELECT`, use `cursor.fetchall()` to get all rows into a list of tuples [7] [8]. Each tuple represents one row, with columns in order. For example, `[(1, 'Alice', 25), (2, 'Bob', 30)]` might be returned.
- **Return or print output:** The function may return the fetched data or a summary string. (One example function returns something like `Query "SELECT..." returned X rows`.)
- **Error handling:** The code wraps these steps in a `try/except` block. If an exception occurs (e.g. SQL syntax error), it can catch it and print an error message.
- **Close connection:** In a `finally:` block, the function calls `conn.close()` to free resources [9]. This ensures the file is closed even if an error happens.

```
def exe_sql_qry(query):
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()
        cursor.execute(query)
        conn.commit()                # commit any changes (INSERT/UPDATE/DELETE)
        result = cursor.fetchall()    # fetch results (empty list if no data returned)
        return result
    except Exception as e:
        print("Error:", e)
    finally:
        conn.close()                # always close the connection
```

*Key points:* Always call `commit()` after writing data [6] and use `finally` to close the connection [9]. This prevents database locks and resource leaks. If `cursor.fetchall()` returns an empty list, it means the query produced no rows (common for INSERT).

## SQL Queries: SELECT, INSERT, and More

**SQL** (Structured Query Language) is used to interact with the database. Common commands include:

- **CREATE TABLE:** Defines a new table and its columns. E.g. `CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)`. Here `id` is a **primary key** (unique row ID) [4].
- **SELECT:** Retrieves data. For example, `SELECT * FROM users WHERE age > 20;` gets all columns (`*`) from the `users` table for rows matching the condition. The result is a set of rows (tuples) [7]. In Python, after executing a `SELECT`, you can do `rows = cursor.fetchall()` to get a list of all matching rows.
- **INSERT:** Adds new data. Example: `INSERT INTO users (name, age) VALUES ('Carol', 22);`. After `INSERT`, you must `commit()` to save it [6]. There is no output rows, so `fetchall()` will return an empty list.
- **UPDATE/DELETE:** Modify or remove data, also requiring `commit()` to take effect.

*Example query structure:*

```
-- Create table with a primary key:
CREATE TABLE students (
    student_id INTEGER PRIMARY KEY,
    name TEXT,
    grade INT
);

-- Insert a new student:
INSERT INTO students (name, grade) VALUES ('Alice', 85);
```

```
-- Select students with grade > 80:
SELECT student_id, name, grade FROM students WHERE grade > 80;
```

In Python, you might run these queries via `cursor.execute(...)`. For example, a `SELECT` query in `exe_sql_qry` could return something like `[(1, 'Alice', 85)]`. We can then print each row or count of rows.

## Running Queries in Parallel with ThreadPoolExecutor

The script uses `concurrent.futures.ThreadPoolExecutor` to run multiple SQL queries at the same time in different threads. Multithreading is useful for I/O-bound tasks like database access [10]. Instead of executing queries one after another, a thread pool allows several queries to proceed concurrently (as long as SQLite's thread mode allows it).

Example structure using `executor.map`:

```
from concurrent.futures import ThreadPoolExecutor

queries = [
    "SELECT * FROM users WHERE age > 20",
    "INSERT INTO users (name, age) VALUES ('Dave', 27)",
    "SELECT name FROM users"
]

with ThreadPoolExecutor(max_workers=4) as executor:
    # executor.map applies exe_sql_qry to each query in the list
    results = executor.map(exe_sql_qry, queries)
    for res in results:
        print(res)
```

Here, `executor.map(exe_sql_qry, queries)` submits each query string to be processed by `exe_sql_qry` in separate threads. It returns an **iterator of results**, in the *same order* as the queries list [11]. Even though the tasks run in parallel, `executor.map` will yield results in sequence. (If you needed out-of-order results, you could use `executor.submit` and `as_completed` instead, but `map` is simpler for beginners.)

The `ThreadPoolExecutor` automatically manages a pool of worker threads [12]. You typically use it with a `with` block, which handles thread startup and shutdown automatically. When all threads finish, control returns and you can print the elapsed time.

## Measuring Execution Time with `time.time()`

To see how fast the parallel execution is, the script likely records the time before and after running the threads. For example:

```
import time

start_time = time.time()  # record start (seconds since epoch)
# ... run ThreadPoolExecutor tasks ...
end_time = time.time()    # record end
print(f"Elapsed time: {end_time - start_time:.2f} seconds")
```

Here, `time.time()` returns the current time in seconds. The difference `end_time - start_time` is the total execution time of the block [13]. We format it (e.g. with 2 decimals) to show how long the queries took. This simple timing method is common for quick measurements.

## Collecting User Input for Queries

The script may allow the user to type in SQL queries interactively. In Python, you can use a loop with `input()` to collect lines until the user stops. For example:

```
queries = []
while True:
    q = input("Enter SQL query (blank to finish): ")
    if not q.strip():
        break
    queries.append(q)
```

This code repeatedly prompts the user. If the user enters a blank line, the loop breaks. Otherwise, each query string is added to the `queries` list using `append()` [14]. After this, `queries` contains all the SQL commands the user entered, ready to be executed in parallel. This lets the program run *any* set of queries given at runtime.

## Example Output

Here is a hypothetical example of how the script might run. Imagine we have a `students` table with some data. The user enters two queries, one `SELECT` and one `INSERT`, then leaves input blank to finish:

```
Enter SQL query (blank to finish): SELECT student_id, name, grade FROM students WHERE grade >= 85
Enter SQL query (blank to finish): INSERT INTO students (name, grade) VALUES ('Eve', 91)
Enter SQL query (blank to finish):

Query 'SELECT student_id, name, grade FROM students WHERE grade >= 85' returned 2 rows:
 [(1, 'Alice', 85), (2, 'Bob', 90)]
Query 'INSERT INTO students (name, grade) VALUES ('Eve', 91)' returned 0 rows
```

```
Elapsed time: 0.04 seconds
```

- For the **SELECT** query, the program prints the list of matching rows (two tuples). Each tuple has `(student_id, name, grade)`.
- For the **INSERT** query, it prints an empty list `[]` (zero rows returned) since INSERT does not return data, only adds a new row.
- Finally, it shows how long all queries took (e.g. `0.04 seconds`).

In practice, the exact print format depends on how `exe_sql_qry` and the main code use `print()`. But generally, you will see each query and its result or number of rows, followed by the elapsed time.

## Key Concepts Summary

- **Primary Key:** A unique ID for each row (e.g. `id INTEGER PRIMARY KEY`) [4]. Ensures each row can be identified.
- **SELECT:** Retrieves data. Example: `SELECT * FROM table WHERE condition;` returns matching rows as tuples [7].
- **INSERT:** Adds new data. Example: `INSERT INTO table(col1, col2) VALUES(val1, val2);`. Must call `commit()` to save.
- **ThreadPoolExecutor:** Runs tasks in parallel threads. Use `executor.map(func, iterable)` to apply a function concurrently to many inputs [11]. Ideal for I/O tasks like database queries [10].
- `time.time()`**:** Gets current time. Compute differences to measure elapsed time [13].
- `input()`**:** Reads a line from the user. Use in a loop with `append()` to build a list of queries [14].

Each of these steps is straightforward in Python. By combining them, the script allows multiple database queries to run concurrently and reports their results and total time. Following this pattern, you can efficiently execute and time multiple SQLite operations in a beginner-friendly way.

**Sources:** Official SQLite and Python documentation and tutorials have been used to clarify these concepts [1] [2] [3] [7] [10] [11] [13] [14] [4].

[1] **About SQLite**
https://www.sqlite.org/about.html

[2] [4] **SQLite with Python**
https://www.tutorialspoint.com/sqlite/sqlite_python.htm

[3] [5] [6] [7] [8] **sqlite3 — DB-API 2.0 interface for SQLite databases — Python 3.13.3 documentation**
https://docs.python.org/3/library/sqlite3.html

[9] **How to manage sqlite3 transaction commits | LabEx**
https://labex.io/tutorials/python-how-to-manage-sqlite3-transaction-commits-446986

[10] [12] **How to use ThreadPoolExecutor in Python3 ? | GeeksforGeeks**
https://www.geeksforgeeks.org/how-to-use-threadpoolexecutor-in-python3/

[11] **map() vs. submit() With the ThreadPoolExecutor in Python - Super Fast Python**
https://superfastpython.com/threadpoolexecutor-map-vs-submit/

[13] **How to check the execution time of Python script ? | GeeksforGeeks**
https://www.geeksforgeeks.org/how-to-check-the-execution-time-of-python-script/

[14] **Get a list as input from user in Python | GeeksforGeeks**
https://www.geeksforgeeks.org/python-get-a-list-as-input-from-user/