ELEC50009 Information Processing

Lab 5

Building a Relational Database with SQLite

In this lab, we will install a simple RDBMS (SQLite) on EC2 and then:

i. Create a relational database and populate it with data.

ii. Run SQL queries from the shell on EC2.

iii. Write Python scripts to query the database on EC2 remotely, collect responses, and process them locally on our machine.

1 SQLite as RDBMS

In this lab on relational databases we will be using an RDBMS called SQLite. It is a lightweight, serverless database that stores all its data in a single file instead of running as a separate server. It is fast, easy to install, and requires almost no setup, which makes it ideal for small applications. For learning, SQLite is especially useful because you can focus on writing and understanding SQL without worrying about servers or configuration.

1.1 Installing SQLite on the EC2 Instance

After logging into your Ubuntu EC2 instance, SQLite can be installed directly using the system package manager. Follow the steps below.

– **Step 1 – Log into the EC2 instance:** Connect to your running Ubuntu EC2 machine using SSH (as in Lab 4).

– **Step 2 – Update the package list:** Refresh the package index:

```
$ sudo apt update
```

– **Step 3 – Install SQLite:** Install the SQLite command-line tool:

```
$ sudo apt install sqlite3 -y
```

– **Step 4 – Verify the installation:** Check that SQLite is available:

```
$ sqlite3 --version
```

You should now be able to run the `sqlite3` command from the terminal and begin creating databases.

1.2 Setting Up the Chinook Sample Database

The Chinook database is a small sample relational database that models a digital music store and is useful for practising SQL queries involving joins, grouping, and aggregation. It contains tables such as `Artist`, `Album`, `Track`, `Customer`, `Invoice`, `InvoiceLine`, `Playlist`, and `PlaylistTrack`, which together capture customers, purchases, and music catalog information. The simplified schema is illustrated below in Figure 1 for reference.

The Chinook database creation script is available online. Before proceeding further, please skim through this schema to see how database tables are created, populated, and constrained in SQL.

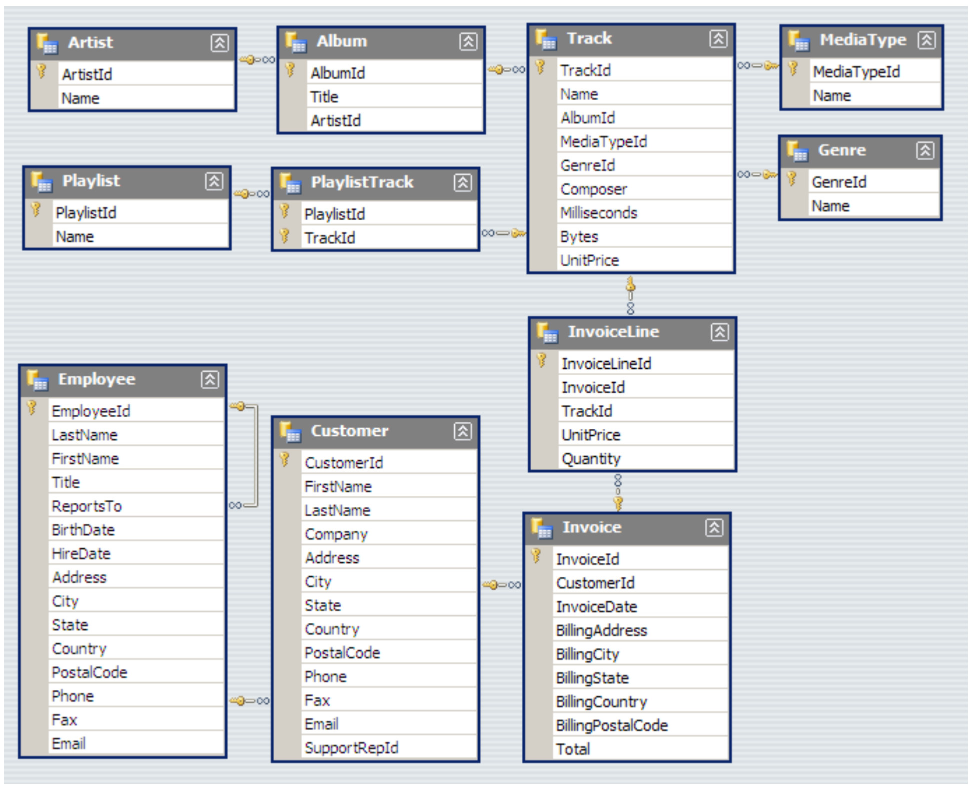

Figure 1: Simplified schema of the Chinook database.

In this section, you will download the script directly onto your EC2 instance and use it to generate an SQLite database file.

Follow the steps given below:

- **Step 1 – Download the Chinook SQL script:** Fetch the script directly onto the EC2 instance and save it as `chinook.sql`:

```
$ wget -O chinook.sql https://raw.githubusercontent.com/
    lerocha/chinook-database/master/ChinookDatabase/
    DataSources/Chinook_Sqlite.sql
```

- **Step 2 – Confirm the file is present:** Verify that the script has been downloaded:

```
$ ls
```

- **Step 3 – Create the database file:** Execute the script using SQLite to generate `chinook.db`:

```
$ sqlite3 chinook.db < chinook.sql
```

- **Step 4 – Open the database:** Launch SQLite with the new database:

```
$ sqlite3 chinook.db
```

- **Step 5 – Verify the tables:** List the tables to confirm successful creation:

```
sqlite> .tables
```

You can now run SQL queries on `chinook.db` for the exercises in this lab.

1.3 Inspecting the Database

Run the following queries inside `sqlite3 chinook.db` and examine the outputs to understand the database structure, table contents, and relationships.

1. **Show all tables in the database.** List the names of every table created by the Chinook script.

   ```
   .tables
   ```

2. **Show the schema of `Customer`.** Display the table structure and column definitions.

   ```
   .schema Customer
   ```

3. **Preview customers.** Display the first few customer records.

   ```
   SELECT * FROM Customer LIMIT 5;
   ```

4. **Preview artists.** Display a small sample of artists in the catalogue.

   ```
   SELECT * FROM Artist LIMIT 5;
   ```

5. **Preview albums with artist names.** Perform a join between `Album` and `Artist` to show album titles with their artists.

   ```
   SELECT Album.Title, Artist.Name
   FROM Album
   JOIN Artist ON Album.ArtistId = Artist.ArtistId
   LIMIT 5;
   ```

6. **Count tracks.** Compute the total number of tracks in the database.

   ```
   SELECT COUNT(*) FROM Track;
   ```

7. **List invoices.** Display a few invoices together with their totals.

   ```
   SELECT InvoiceId, CustomerId, Total
   FROM Invoice
   LIMIT 5;
   ```

8. **Find the largest invoice.** Compute the maximum purchase amount recorded.

   ```
   SELECT MAX(Total) FROM Invoice;
   ```

9. **Group tracks by album.** Perform grouping to count how many tracks each album contains.

   ```
   SELECT AlbumId, COUNT(*) AS NumTracks
   FROM Track
   GROUP BY AlbumId
   LIMIT 5;
   ```

10. **Show playlists and track counts.** Perform a join and grouping to compute the number of tracks in each playlist.

    ```
    SELECT Playlist.Name, COUNT(*) AS NumTracks
    FROM Playlist
    JOIN PlaylistTrack
      ON Playlist.PlaylistId = PlaylistTrack.PlaylistId
    GROUP BY Playlist.PlaylistId;
    ```

2 Remote Query Server and Client

In this section you will write a small Python server that accepts an SQL query sent remotely, runs it on your `chinook.db` database on EC2, and returns the results as JSON. You will implement it here using Flask to demonstrate the concept; the exact tool chain you use in practice will depend on your project needs.

– **Step 1 – Create a Python virtual environment:** Install the required tools and create an isolated environment for this project.

```
$ sudo apt install python3-venv -y
$ python3 -m venv venv
$ source venv/bin/activate
```

– **Step 2 – Install Flask:** With the virtual environment activated, install Flask using `pip`.

```
(venv) $ pip install flask
```

– **Step 3 – Create the server file:** Create a file called `db_server.py` and add the following code. (The file is also available on the course GitHub).

```
from flask import Flask, request, jsonify
import sqlite3

app = Flask(__name__)
DB_PATH = "chinook.db"

@app.post("/query")
def query_db():
    sql = request.json.get("sql", "")

    con = sqlite3.connect(DB_PATH)
    con.row_factory = sqlite3.Row
    cur = con.cursor()

    cur.execute(sql)
    rows = [dict(r) for r in cur.fetchall()]
    con.close()

    return jsonify(rows)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

This program starts a small Flask web server that listens for HTTP requests on port `5000`. A client sends an SQL query in JSON format to the `/query` endpoint, the server executes the query on `chinook.db` using SQLite, and returns the results back to the client as JSON. In this way, the database is accessed remotely through a simple HTTP API.

– **Step 4 – Ensure the database file is present:** Confirm that `chinook.db` is in the same directory as `db_server.py`.

```
(venv) $ ls
```

– **Step 5 – Run the server:** Start the Flask application.

```
(venv) $ python3 db_server.py
```

The database server is now running.

– **Step 6 – Run the client program locally:** A simple client script `db_client.py` is provided on the course GitHub. Download it to your local machine, inspect the code to understand how it sends an HTTP request to the server, and then run it to query your remote database server.

```
$ python db_client.py
```

If the server is running correctly, the response will contain the query results in JSON.

3 API-Based Queries

In this section you will redesign the client–server interaction so that the client does not construct SQL queries and send them to EC2. Instead, the client calls specific API endpoints (high-level functions), and the server constructs the SQL internally and returns results as JSON. This is closer to how real applications expose database functionality safely. Read the tasks below and implement them incrementally.

Note: The snippets below are partial hints only; you must complete both the server and client implementations. (In each of the following, IP means your EC2's public IP).

i. **Retrieve a list of artists (with a configurable limit).** Implement `GET /artists?limit=5` and return artist names. Use a parameterised query and pass the limit value separately when executing the SQL.

   Server-side:

```
@app.get("/artists")
def artists():
    limit = int(request.args.get("limit", "5"))

    sql = "SELECT Name FROM Artist LIMIT ?"

    cur = con.cursor()
    cur.execute(sql, (limit,))   # pass parameters
       separately
    rows = cur.fetchall()
```

   Client-side:

```
import requests
r = requests.get(f"http://{IP}:5000/artists", params={"
   limit": 5})
print(r.json())
```

ii. **Retrieve albums for a given artist name (exact match).** Implement `GET /albums?artist=AC%2FDC` and return album titles.

   Server-side:

```
@app.get("/albums")
def albums():
    artist = request.args.get("artist", "")
    sql = """
    SELECT Album.Title
    FROM Album JOIN Artist ON Album.ArtistId = Artist.
       ArtistId
    WHERE Artist.Name = ?
    """
```

Client-side:

```
import requests
r = requests.get(f"http://{IP}:5000/albums", params={"
   artist": "AC/DC"})
print(r.json())
```

iii. **Search tracks by a keyword (partial match) and return a small preview.** Implement `GET /tracks/search?q=love&limit=10` and return track names. Use `LIKE` with wildcards.

Server-side:

```
@app.get("/tracks/search")
def track_search():
    q = request.args.get("q", "")
    limit = int(request.args.get("limit", "10"))
    sql = "SELECT Name FROM Track WHERE Name LIKE ? LIMIT ?
       "
    like = f"%{q}%"
```

Client-side:

```
import requests
r = requests.get(f"http://{IP}:5000/tracks/search", params
   ={"q": "love", "limit": 10})
print(r.json())
```

iv. **Retrieve invoice totals for a given customer (by email), ordered by date.** Implement `GET /customer/invoices?email=someone@example.com` and return invoice id, date, and total.

Server-side:

```
@app.get("/customer/invoices")
def customer_invoices():
    email = request.args.get("email", "")
    sql = """
    SELECT Invoice.InvoiceId, Invoice.InvoiceDate, Invoice.
       Total
    FROM Invoice JOIN Customer ON Invoice.CustomerId =
       Customer.CustomerId
    WHERE Customer.Email = ?
    ORDER BY Invoice.InvoiceDate DESC
    """
```

Client-side:

```
import requests
r = requests.get(f"http://{IP}:5000/customer/invoices",
   params={"email": "someone@example.com"})
print(r.json())
```

v. **Return a summary report of the top customers by total spending.** Implement `GET /reports/top-customers?limit=10` using grouping and ordering.

Server-side:

```
@app.get("/reports/top-customers")
def top_customers():
    limit = int(request.args.get("limit", "10"))
    sql = """
    SELECT Customer.CustomerId, Customer.FirstName,
       Customer.LastName,
           SUM(Invoice.Total) AS Spend
    FROM Customer JOIN Invoice ON Customer.CustomerId =
       Invoice.CustomerId
    GROUP BY Customer.CustomerId
    ORDER BY Spend DESC
    LIMIT ?
    """
```

Client-side:

```
import requests
r = requests.get(f"http://{IP}:5000/reports/top-customers",
    params={"limit": 10})
print(r.json())
```

As you implement each endpoint, the client should only send simple parameters (name, email, keyword, or limit), and the server should construct the SQL internally and return a JSON response.

Note: In practice, this communication should use HTTPS (TLS encryption) so that client requests and server responses are protected while travelling over the network.

■