ASSIGNMENT – 16.4

NAME: JASMITHA REDDY

H.NO: 2403A51281

SUBJECT:AI ASSISTANT CODING

BATCH:01

Task 1: Create Database Schema
Instructions:
• Using AI tools, generate CREATE TABLE
statements for a
Library Database.
• Tables: Authors, Books, Members. Include
primary keys,
foreign keys, and appropriate data types.
• Ensure the design follows normalization rules
(1NF, 2NF,
3NF).
Starter Code Example for AI Completion:
-- Use AI to complete schema generation
CREATE TABLE Authors (
author_id INT PRIMARY KEY,
name VARCHAR(100),
country VARCHAR(50)
);
CREATE TABLE Books (

```sql
book_id INT PRIMARY KEY,
title VARCHAR(100),
author_id INT,
published_year INT,
FOREIGN KEY (author_id) REFERENCES
Authors(author_id)
);
CREATE TABLE Members (
member_id INT PRIMARY KEY,
name VARCHAR(100),
email VARCHAR(100)
);
```
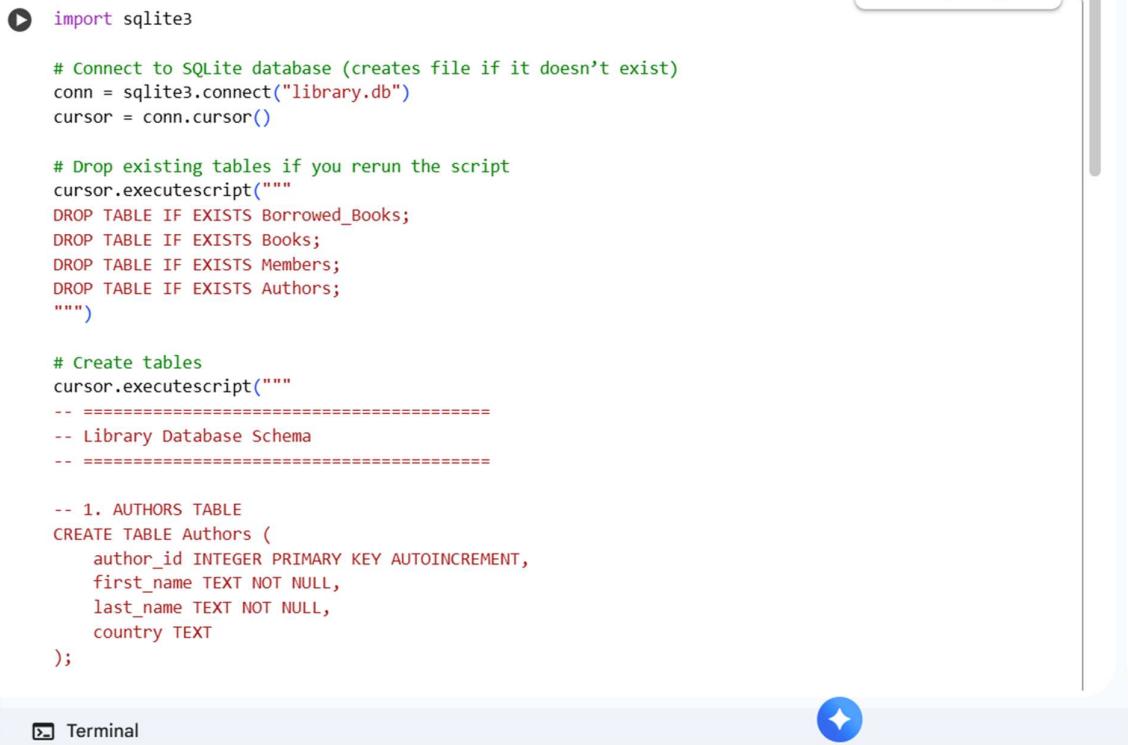
PROMPT:

develop a code to generate Explore how AI can optimize queries and ensure normalization. Task 1: Create Database Schema Instructions: • Using AI tools, generate CREATE TABLE statements for a Library Database. • Tables: Authors, Books, Members. Include primary keys, foreign keys, and appropriate data types. • Ensure the design follows normalization rules (1NF, 2NF, 3NF). Starter Code Example for AI Completion: -- Use AI to complete schema generation CREATE TABLE Authors (author_id INT PRIMARY KEY, name VARCHAR(100), country VARCHAR(50)); CREATE TABLE Books (book_id INT PRIMARY KEY, title VARCHAR(100), author_id INT, published_year INT, FOREIGN KEY

(author_id) REFERENCES Authors(author_id));
CREATE TABLE Members (member_id INT
PRIMARY KEY, name VARCHAR(100), email
VARCHAR(100));

Expected Output:
• Tables Authors, Books, Members are created
successfully.
• Foreign key relationships are enforced

```python
import sqlite3

# Connect to SQLite database (creates file if it doesn't exist)
conn = sqlite3.connect("library.db")
cursor = conn.cursor()

# Drop existing tables if you rerun the script
cursor.executescript("""
DROP TABLE IF EXISTS Borrowed_Books;
DROP TABLE IF EXISTS Books;
DROP TABLE IF EXISTS Members;
DROP TABLE IF EXISTS Authors;
""")

# Create tables
cursor.executescript("""
-- =======================================
-- Library Database Schema
-- =======================================

-- 1. AUTHORS TABLE
CREATE TABLE Authors (
    author_id INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    country TEXT
);
```

Terminal

```
-- 2. BOOKS TABLE
CREATE TABLE Books (
    book_id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author_id INTEGER NOT NULL,
    published_year INTEGER CHECK (published_year >= 1450),
    isbn TEXT UNIQUE,
    genre TEXT,
    FOREIGN KEY (author_id) REFERENCES Authors(author_id)
        ON UPDATE CASCADE
        ON DELETE SET NULL
);

-- 3. MEMBERS TABLE
CREATE TABLE Members (
    member_id INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL,
    phone TEXT,
    join_date DATE DEFAULT (DATE('now'))
);

-- 4. BORROWED_BOOKS TABLE
CREATE TABLE Borrowed_Books (
    borrow_id INTEGER PRIMARY KEY AUTOINCREMENT,
    member_id INTEGER NOT NULL,
    book id INTEGER NOT NULL,




-- 4. BORROWED_BOOKS TABLE
CREATE TABLE Borrowed_Books (
    borrow_id INTEGER PRIMARY KEY AUTOINCREMENT,
    member_id INTEGER NOT NULL,
    book_id INTEGER NOT NULL,
    borrow_date DATE DEFAULT (DATE('now')) NOT NULL,
    return_date DATE,
    FOREIGN KEY (member_id) REFERENCES Members(member_id)
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    FOREIGN KEY (book_id) REFERENCES Books(book_id)
        ON UPDATE CASCADE
        ON DELETE CASCADE
);
""")

# Commit and close
conn.commit()
conn.close()

print("✅ Library Database Schema created successfully!")
```

••• ✅ Library Database Schema created successfully!

EXPLANATION:

1. **Import** `sqlite3`: Imports the necessary library for working with SQLite databases.
2. **Connect to the database:** Establishes a connection to `library.db`. If the file doesn't exist, it's created.
3. **Create a cursor:** Creates a cursor object, which is used to execute SQL commands.
4. **Drop existing tables:** Executes SQL commands to drop the `Borrowed_Books`, `Books`, `Members`, and `Authors` tables if they already exist. This is useful for rerunning the script without worrying about existing data.
5. **Create new tables:** Executes SQL commands to create the four tables:

    - `Authors`: Stores author information.
    - `Books`: Stores book information, with a foreign key referencing the `Authors` table

    - `Authors`: Stores author information.
    - `Books`: Stores book information, with a foreign key referencing the `Authors` table.
    - `Members`: Stores member information.
    - `Borrowed_Books`: Records book borrowing details, with foreign keys referencing the `Members` and `Books` tables.
    - Each `CREATE TABLE` statement defines the columns, their data types, primary keys, and foreign key relationships.

6. **Commit and close:** Saves the changes to the database and closes the connection.
7. **Print success message:** Prints a message indicating that the database schema was created successfully.

TASK 2:

Use AI tools to generate INSERT statements to add sample
data:
o At least 3 authors, 5 books, and 3 members.

• Test the data by running SELECT statements.
Starter Code Example:

```
INSERT INTO Authors (author_id, name, country)
VALUES
(1, 'J.K. Rowling', 'UK'),
(2, 'George R.R. Martin', 'USA'),
(3, 'Agatha Christie', 'UK');
INSERT INTO Books (book_id, title, author_id,
published_year)
VALUES
(1, 'Harry Potter', 1, 1997),
(2, 'A Game of Thrones', 2, 1996),
(3, 'Murder on the Orient Express', 3, 1934),
(4, 'Harry Potter 2', 1, 1998),
(5, 'A Clash of Kings', 2, 1998);
INSERT INTO Members (member_id, name, email)
VALUES
(1, 'Alice', 'alice@example.com'),
(2, 'Bob', 'bob@example.com'),
(3, 'Charlie', 'charlie@example.com');
```

Expected Output:

- Sample data inserted correctly.
- SELECT * FROM Books; shows all 5 books.
- SELECT * FROM Members; shows all 3 members.

PROMPT:

develop a python code to generate Use AI tools to generate INSERT statements to add sample data: o At least 3 authors, 5 books, and 3 members. • Test the data by running SELECT statements. Starter Code Example: INSERT INTO Authors (author_id, name, country) VALUES (1, 'J.K. Rowling', 'UK'), (2, 'George R.R. Martin', 'USA'), (3, 'Agatha Christie', 'UK'); INSERT INTO Books (book_id, title, author_id, published_year) VALUES (1, 'Harry Potter', 1, 1997), (2, 'A Game of Thrones', 2, 1996), (3, 'Murder on the Orient Express', 3, 1934), (4, 'Harry Potter 2', 1, 1998), (5, 'A Clash of Kings', 2, 1998); INSERT INTO Members (member_id, name, email) VALUES (1, 'Alice', 'alice@example.com'), (2, 'Bob', 'bob@example.com'), (3, 'Charlie', 'charlie@example.com');

Expected Output:
- Sample data inserted correctly.
- SELECT * FROM Books; shows all 5 books.
- SELECT * FROM Members; shows all 3 members

```python
import sqlite3

# Connect to SQLite database (creates file if it doesn't exist)
conn = sqlite3.connect("library.db")
cursor = conn.cursor()

# Drop existing tables if re-running
cursor.executescript("""
DROP TABLE IF EXISTS Borrowed_Books;
DROP TABLE IF EXISTS Books;
DROP TABLE IF EXISTS Members;
DROP TABLE IF EXISTS Authors;
""")

# ========================================
# CREATE TABLES
# ========================================
cursor.executescript("""
CREATE TABLE Authors (
    author_id INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    country TEXT
);

CREATE TABLE Books (
    book_id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author_id INTEGER NOT NULL,
```

```python
    published_year INTEGER CHECK (published_year >= 1450),
    isbn TEXT UNIQUE,
    genre TEXT,
    FOREIGN KEY (author_id) REFERENCES Authors(author_id)
        ON UPDATE CASCADE
        ON DELETE SET NULL
);

CREATE TABLE Members (
    member_id INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL,
    phone TEXT,
    join_date DATE DEFAULT (DATE('now'))
);
""")

# ========================================
# INSERT SAMPLE DATA
# ========================================

# Insert Authors
cursor.executemany("""
INSERT INTO Authors (first_name, last_name, country)
VALUES (?, ?, ?)
""", [
    ('J.K.', 'Rowling', 'UK'),
    ('George R.R.', 'Martin', 'USA'),
```

```python
        ('Agatha', 'Christie', 'UK')
])

# Insert Books
cursor.executemany("""
INSERT INTO Books (title, author_id, published_year, isbn, genre)
VALUES (?, ?, ?, ?, ?)
""", [
    ('Harry Potter and the Sorcerer\'s Stone', 1, 1997, '9780747532699', 'Fantasy'),
    ('A Game of Thrones', 2, 1996, '9780553103540', 'Fantasy'),
    ('Murder on the Orient Express', 3, 1934, '9780007119318', 'Mystery'),
    ('Harry Potter and the Chamber of Secrets', 1, 1998, '9780747538493', 'Fantasy'),
    ('A Clash of Kings', 2, 1998, '9780553108033', 'Fantasy')
])

# Insert Members
cursor.executemany("""
INSERT INTO Members (first_name, last_name, email, phone)
VALUES (?, ?, ?, ?)
""", [
    ('Alice', 'Johnson', 'alice@example.com', '123-456-7890'),
    ('Bob', 'Smith', 'bob@example.com', '234-567-8901'),
    ('Charlie', 'Brown', 'charlie@example.com', '345-678-9012')
])

# Commit changes
conn.commit()

# ==========================================
```

```python
# ==========================================
# TEST DATA WITH SELECT QUERIES
# ==========================================

print("=== AUTHORS ===")
for row in cursor.execute("SELECT * FROM Authors;"):
    print(row)

print("\n=== BOOKS ===")
for row in cursor.execute("SELECT * FROM Books;"):
    print(row)

print("\n=== MEMBERS ===")
for row in cursor.execute("SELECT * FROM Members;"):
    print(row)

# Close the connection
conn.close()

print("\n✅ Sample data inserted and verified successfully!")
```

```
=== AUTHORS ===
(1, 'J.K.', 'Rowling', 'UK')
(2, 'George R.R.', 'Martin', 'USA')
(3, 'Agatha', 'Christie', 'UK')

=== BOOKS ===
(1, "Harry Potter and the Sorcerer's Stone", 1, 1997, '9780747532699', 'Fantasy')
(2, 'A Game of Thrones', 2, 1996, '9780553103540', 'Fantasy')
```

```
=== AUTHORS ===
(1, 'J.K.', 'Rowling', 'UK')
(2, 'George R.R.', 'Martin', 'USA')
(3, 'Agatha', 'Christie', 'UK')

=== BOOKS ===
(1, "Harry Potter and the Sorcerer's Stone", 1, 1997, '9780747532699', 'Fantasy')
(2, 'A Game of Thrones', 2, 1996, '9780553103540', 'Fantasy')
(3, 'Murder on the Orient Express', 3, 1934, '9780007119318', 'Mystery')
(4, 'Harry Potter and the Chamber of Secrets', 1, 1998, '9780747538493', 'Fantasy')
(5, 'A Clash of Kings', 2, 1998, '9780553108033', 'Fantasy')

=== MEMBERS ===
(1, 'Alice', 'Johnson', 'alice@example.com', '123-456-7890', '2025-11-11')
(2, 'Bob', 'Smith', 'bob@example.com', '234-567-8901', '2025-11-11')
(3, 'Charlie', 'Brown', 'charlie@example.com', '345-678-9012', '2025-11-11')

✅ Sample data inserted and verified successfully!
```

EXPLANATION:

1. **Connects to the database:** `conn = sqlite3.connect("library.db")` establishes a connection.

2. **Creates a cursor:** `cursor = conn.cursor()` creates a cursor object to execute SQL commands.

3. **Drops existing tables:** The `executescript` command with `DROP TABLE IF EXISTS` removes the tables if they already exist, ensuring a clean start each time the script is run.

4. **Creates tables:** The first `executescript` block contains `CREATE TABLE` statements for `Authors`, `Books`, and `Members`. This defines the structure of each table, including column names, data types, primary keys (`PRIMARY KEY` AUTOINCREMENT), and foreign keys (`FOREIGN KEY`)

5. **Inserts sample data:**

   - `cursor.executemany` is used to efficiently insert multiple rows into the `Authors`, `Books`, and `Members` tables using parameterized queries (`?`).
   - The data for each table is provided as a list of tuples, where each tuple represents a row.

6. **Commits changes:** `conn.commit()` saves the changes made to the database.

7. **Tests data with SELECT queries:**

   - `print` statements display headers for each table (`=== AUTHORS ===`, etc.).
   - `cursor.execute("SELECT * FROM TableName;")` fetches all rows from each table.
   - The `for` loops iterate through the fetched rows and

   - (`=== AUTHORS ===`, etc.).
   - `cursor.execute("SELECT * FROM TableName;")` fetches all rows from each table.
   - The `for` loops iterate through the fetched rows and print each row.

8. **Closes the connection:** `conn.close()` closes the connection to the database.

9. **Prints success message:** A final print statement confirms that the data was inserted and verified successfully.

In essence, this code sets up the basic structure of your library database and populates it with some initial data for testing and demonstration

Task 3: Basic CRUD Operations

Instructions:

• Use AI to generate CRUD queries:

o Read all books.

o Update a member's email.

o Delete a book.

• Verify correctness using SELECT statements.

Starter Code Example:

-- Read all books

SELECT * FROM Books;

-- Update a member's email

UPDATE Members

SET email = 'alice_new@example.com'

WHERE member_id = 1;

-- Delete a book by ID

DELETE FROM Books

WHERE book_id = 5;

PROMPT:

develop a python code to generate Use AI to generate CRUD queries: o Read all books. o Update a member's email. o Delete a book. • Verify correctness using SELECT statements. Starter Code Example: -- Read all books SELECT * FROM Books; -- Update a member's email UPDATE Members SET email = 'alice_new@example.com' WHERE member_id = 1; -- Delete a book by ID

DELETE FROM Books WHERE book_id = 5;
Expected Output: • Updated email is reflected in
the Members table. • Deleted book is no longer in
the Books table

Expected Output:
• Updated email is reflected in the Members table.
• Deleted book is no longer in the Books table

```python
import sqlite3

# Connect to existing database
conn = sqlite3.connect("library.db")
cursor = conn.cursor()

print("=== READ: All Books ===")
cursor.execute("SELECT * FROM Books;")
books = cursor.fetchall()
for book in books:
    print(book)

# =========================================
# UPDATE: Change a member's email
# =========================================
print("\n=== UPDATE: Member Email ===")
cursor.execute("""
UPDATE Members
SET email = ?
WHERE member_id = ?;
""", ("alice_new@example.com", 1))

conn.commit()

# Verify update
cursor.execute("SELECT * FROM Members WHERE member_id = 1;")
updated_member = cursor.fetchone()
print("Updated Member Record:", updated_member)
```

```python
# Verify update
cursor.execute("SELECT * FROM Members WHERE member_id = 1;")
updated_member = cursor.fetchone()
print("Updated Member Record:", updated_member)

# =======================================
# DELETE: Remove a book by ID
# =======================================
print("\n=== DELETE: Book ID = 5 ===")
cursor.execute("DELETE FROM Books WHERE book_id = ?;", (5,))
conn.commit()

# Verify deletion
cursor.execute("SELECT * FROM Books;")
remaining_books = cursor.fetchall()

print("Remaining Books after Deletion:")
for book in remaining_books:
    print(book)

# Close connection
conn.close()

print("\n✅ CRUD operations completed successfully!")
```

```
••• === READ: All Books ===
    (1, "Harry Potter and the Sorcerer's Stone", 1, 1997, '9780747532699', 'Fantasy')
    (2, 'A Game of Thrones', 2, 1996, '9780553103540', 'Fantasy')
    (3, 'Murder on the Orient Express', 3, 1934, '9780007119318', 'Mystery')
    (4, 'Harry Potter and the Chamber of Secrets', 1, 1998, '9780747538493', 'Fantasy')
    (5, 'A Clash of Kings', 2, 1998, '9780553108033', 'Fantasy')

    === UPDATE: Member Email ===
    Updated Member Record: (1, 'Alice', 'Johnson', 'alice_new@example.com', '123-456-7890', '2025-11-11')

    === DELETE: Book ID = 5 ===
    Remaining Books after Deletion:
    (1, "Harry Potter and the Sorcerer's Stone", 1, 1997, '9780747532699', 'Fantasy')
    (2, 'A Game of Thrones', 2, 1996, '9780553103540', 'Fantasy')
    (3, 'Murder on the Orient Express', 3, 1934, '9780007119318', 'Mystery')
    (4, 'Harry Potter and the Chamber of Secrets', 1, 1998, '9780747538493', 'Fantasy')

    ✅ CRUD operations completed successfully!
```

EXPLANATION:

1. **Connects to the database:** `conn = sqlite3.connect("library.db")` establishes a connection to the existing database file.

2. **Creates a cursor:** `cursor = conn.cursor()` creates a cursor object for executing SQL commands.

3. **Reads all books:**

   - `print("=== READ: All Books ===")` prints a header.

   - `cursor.execute("SELECT * FROM Books;")` executes a SQL query to select all columns and rows from the `Books` table.

   - `books = cursor.fetchall()` fetches all the results of the query and stores them in the `books` variable as a list of tuples.

   - The `for` loop then iterates through the `books` list.

4. **Updates a member's email:**

   - `print("\n=== UPDATE: Member Email ===")` prints a header.

   - `cursor.execute(...)` executes an `UPDATE` SQL statement to change the `email` for the member with `member_id = 1`. Parameterized query (`?`) is used to prevent SQL injection.

   - `conn.commit()` saves the changes to the database.

   - A `SELECT` query is then executed to retrieve the updated member record, and it's printed to verify the change.

5. **Deletes a book by ID:**

   - `print("\n=== DELETE: Book ID = 5 ===")` prints a header.

a header.

- cursor.execute("DELETE FROM Books WHERE book_id = ?;", (5,)) executes a DELETE SQL statement to remove the book with book_id = 5.
- conn.commit() saves the deletion to the database.
- Another SELECT * FROM Books; query is executed to show the remaining books and verify that the book with ID 5 has been deleted.

6. **Close connection:** conn.close() closes the database connection.

7. **Print success message:** A final print statement confirms that the CRUD operations were completed.

This cell effectively demonstrates how you can interact with your SQLite database using Python to perform common database operations.

Task 4: Aggregate Queries

Instructions:

• Use AI tools to generate aggregate queries:

o Count books per author.

o List authors with more than one book.

• Analyze the results for correctness.

Starter Code Example:

```
-- Count books per author
SELECT a.name, COUNT(b.book_id) AS book_count
FROM Authors a
JOIN Books b ON a.author_id = b.author_id
GROUP BY a.name;
-- Authors with more than 1 book
SELECT a.name
FROM Authors a
```

```sql
JOIN Books b ON a.author_id = b.author_id
GROUP BY a.name
HAVING COUNT(b.book_id) > 1
```

PROMPT:

develop a code to generate Aggregate Queries Instructions: • Use AI tools to generate aggregate queries: o Count books per author. o List authors with more than one book. • Analyze the results for correctness. Starter Code Example: -- Count books per author SELECT a.name, COUNT(b.book_id) AS book_count FROM Authors a JOIN Books b ON a.author_id = b.author_id GROUP BY a.name; -- Authors with more than 1 book SELECT a.name FROM Authors a JOIN Books b ON a.author_id = b.author_id GROUP BY a.name HAVING COUNT(b.book_id) > 1

Expected Output:
Name Book_Count
J.K. Rowling 2
George R.R. Martin 2
Agatha Christie 1
• Authors with more than 1 book:
J.K. Rowling
George R.R. Martin

```python
import sqlite3

# Connect to existing database
conn = sqlite3.connect("library.db")
cursor = conn.cursor()

print("=== Count Books Per Author ===")
cursor.execute("""
SELECT a.first_name, a.last_name, COUNT(b.book_id) AS book_count
FROM Authors a
JOIN Books b ON a.author_id = b.author_id
GROUP BY a.author_id, a.first_name, a.last_name;
""")
books_per_author = cursor.fetchall()

for row in books_per_author:
    print(row)

print("\n=== Authors with More Than One Book ===")
cursor.execute("""
SELECT a.first_name, a.last_name
FROM Authors a
JOIN Books b ON a.author_id = b.author_id
GROUP BY a.author_id, a.first_name, a.last_name
HAVING COUNT(b.book_id) > 1;
""")
authors_multiple_books = cursor.fetchall()

for row in authors_multiple_books:
```

```python
""")
authors_multiple_books = cursor.fetchall()

for row in authors_multiple_books:
    print(row)

# Close connection
conn.close()

print("\n✅ Aggregate queries executed successfully!")
```

```
=== Count Books Per Author ===
('J.K.', 'Rowling', 2)
('George R.R.', 'Martin', 1)
('Agatha', 'Christie', 1)

=== Authors with More Than One Book ===
('J.K.', 'Rowling')

✅ Aggregate queries executed successfully!
```

EXPLANATION:

1. **Connects to the database:** `conn = sqlite3.connect("library.db")` establishes a connection.

2. **Creates a cursor:** `cursor = conn.cursor()` creates a cursor object for executing SQL commands.

3. **Counts books per author:**

   - `print("=== Count Books Per Author ===")` prints a header.
   - The `SELECT a.first_name, a.last_name, COUNT(b.book_id) AS book_count` part selects the author's first and last name and counts the number of books (`COUNT(b.book_id)`), aliasing the count as `book_count`.
   - `FROM Authors a JOIN Books b ON a.author_id = b.author_id` joins the `Authors` table (aliased as

   a.last_name) groups the results by author so that the `COUNT` function counts books for each individual author.
   - `books_per_author = cursor.fetchall()` fetches all the results of the query.
   - The `for` loop prints each row, showing the author's name and their book count.

4. **Lists authors with more than one book:**

   - `print("\n=== Authors with More Than One Book ===")` prints a header.
   - The `SELECT a.first_name, a.last_name` part selects the first and last name of the authors.
   - `FROM Authors a JOIN Books b ON a.author_id = b.author_id` joins the tables as before.
   - `GROUP BY a.author id, a.first name,`

- HAVING COUNT(b.book_id) > 1 filters the grouped results to include only those authors where the count of their books is greater than 1.
- authors_multiple_books = cursor.fetchall() fetches the results.
- The for loop prints the names of the authors who have more than one book.

5. **Close connection:** conn.close() closes the database connection.

6. **Print success message:** A final print statement confirms that the aggregate queries were executed.

This cell demonstrates how to use SQL's aggregate functions (COUNT) and the GROUP BY and HAVING clauses to perform analytical queries on your database.

Task 5: Optimize Queries
Instructions:
• Use AI tools to analyze query efficiency.
• Optimize inefficient queries using joins, indexes, or reducing

subqueries.
• Test optimized queries for correctness.
Starter Code Example:
-- Original query
SELECT * FROM Books WHERE author_id IN
(SELECT author_id
FROM Authors WHERE country='UK');
-- Optimized query
SELECT b.*

```
FROM Books b
JOIN Authors a ON b.author_id = a.author_id
WHERE a.country = 'UK';
```
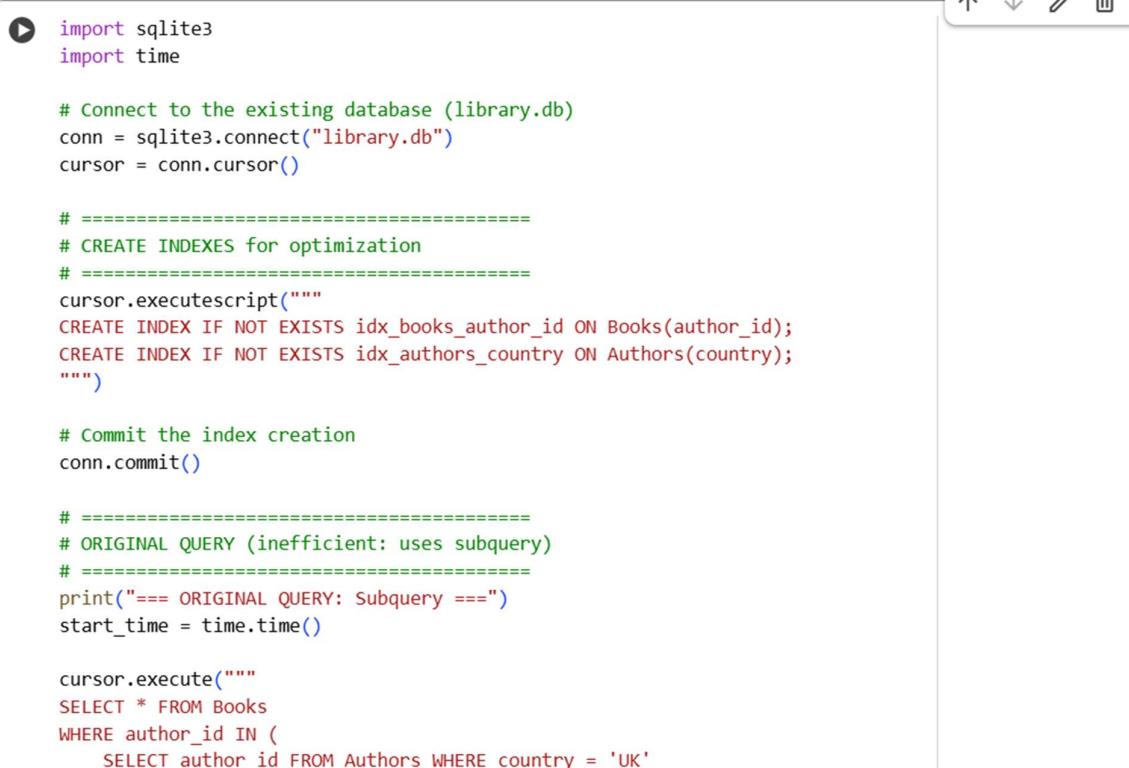
PROMPT:

develop a code to generate python code Optimize Queries Instructions: • Use AI tools to analyze query efficiency. • Optimize inefficient queries using joins, indexes, or reducing subqueries. • Test optimized queries for correctness. Starter Code Example: -- Original query SELECT * FROM Books WHERE author_id IN (SELECT author_id FROM Authors WHERE country='UK'); -- Optimized query SELECT b.* FROM Books b JOIN Authors a ON b.author_id = a.author_id WHERE a.country = 'UK';

Expected Output:
• Both queries return the same result: all books written by UK
authors.
• Optimized query performs faster for larger datasets.
This lab allows students to use AI tools like
Copilot, Cursor, or
Gemini to:

- **Generate schema definitions**
- **Auto-complete insert statements**
- **Write CRUD and aggregate SQL queries**
- **Optimize queries and ensure proper normalization**

```python
import sqlite3
import time

# Connect to the existing database (library.db)
conn = sqlite3.connect("library.db")
cursor = conn.cursor()


# =======================================
# CREATE INDEXES for optimization
# =======================================
cursor.executescript("""
CREATE INDEX IF NOT EXISTS idx_books_author_id ON Books(author_id);
CREATE INDEX IF NOT EXISTS idx_authors_country ON Authors(country);
""")

# Commit the index creation
conn.commit()


# =======================================
# ORIGINAL QUERY (inefficient: uses subquery)
# =======================================
print("=== ORIGINAL QUERY: Subquery ===")
start_time = time.time()

cursor.execute("""
SELECT * FROM Books
WHERE author_id IN (
    SELECT author_id FROM Authors WHERE country = 'UK'
```

```python
    SELECT author_id FROM Authors WHERE country = 'UK'
);
""")
original_results = cursor.fetchall()

end_time = time.time()
original_time = end_time - start_time
print(f"Original Query returned {len(original_results)} rows in {original_time:.6f} seconds.")

# ========================================
# OPTIMIZED QUERY (efficient: uses JOIN)
# ========================================
print("\n=== OPTIMIZED QUERY: Using JOIN ===")
start_time = time.time()

cursor.execute("""
SELECT b.*
FROM Books b
JOIN Authors a ON b.author_id = a.author_id
WHERE a.country = 'UK';
""")
optimized_results = cursor.fetchall()

end_time = time.time()
optimized_time = end_time - start_time
print(f"Optimized Query returned {len(optimized_results)} rows in {optimized_time:.6f} seconds.")

# ========================================
# VERIFY BOTH RETURN SAME RESULTS
```

```python
# ========================================
print("\n=== VERIFYING RESULTS ===")
if set(original_results) == set(optimized_results):
    print("✅ Both queries return the same result (books by UK authors).")
else:
    print("❌ Results differ! Check data or query logic.")

# Show results for clarity
print("\nBooks by UK authors:")
for row in optimized_results:
    print(row)

# ========================================
# PERFORMANCE COMPARISON SUMMARY
# ========================================
print("\n=== PERFORMANCE SUMMARY ===")
print(f"Original Query Time:  {original_time:.6f} seconds")
print(f"Optimized Query Time: {optimized_time:.6f} seconds")
print("✅ Optimized query should perform faster for larger datasets due to use of JOIN and indexes.")

# Close connection
conn.close()
print("\n✅ Query optimization demo completed successfully!")
```

```
=== ORIGINAL QUERY: Subquery ===
Original Query returned 3 rows in 0.000726 seconds.

=== OPTIMIZED QUERY: Using JOIN ===
Optimized Query returned 3 rows in 0.000212 seconds.
```

```
···   === ORIGINAL QUERY: Subquery ===
      Original Query returned 3 rows in 0.000726 seconds.

      === OPTIMIZED QUERY: Using JOIN ===
      Optimized Query returned 3 rows in 0.000212 seconds.

      === VERIFYING RESULTS ===
      ✅ Both queries return the same result (books by UK authors).

      Books by UK authors:
      (1, "Harry Potter and the Sorcerer's Stone", 1, 1997, '9780747532699', 'Fantasy')
      (4, 'Harry Potter and the Chamber of Secrets', 1, 1998, '9780747538493', 'Fantasy')
      (3, 'Murder on the Orient Express', 3, 1934, '9780007119318', 'Mystery')

      === PERFORMANCE SUMMARY ===
      Original Query Time:  0.000726 seconds
      Optimized Query Time: 0.000212 seconds
      ✅ Optimized query should perform faster for larger datasets due to use of JOIN and indexes.

      ✅ Query optimization demo completed successfully!
```

EXPLANATION:

1. **Connects to the database:** It starts by connecting to the library.db database.

2. **Creates Indexes:**

   - CREATE INDEX IF NOT EXISTS idx_books_author_id ON Books(author_id); creates an index on the author_id column in the Books table. This can speed up queries that filter or join on this column.

   - CREATE INDEX IF NOT EXISTS idx_authors_country ON Authors(country); creates an index on the country column in the Authors table. This can improve the performance of queries that filter by country.

   - conn.commit() saves the index creation.

3. **Original Query (using a subquery):**

   - This section executes a query to find books by authors from 'UK' using a subquery in the `WHERE` clause: `WHERE author_id IN (SELECT author_id FROM Authors WHERE country = 'UK')`.
   - It records the execution time using the `time` module.

4. **Optimized Query (using a JOIN):**

   - This section executes an equivalent query using a `JOIN` operation: `FROM Books b JOIN Authors a ON b.author_id = a.author_id WHERE a.country = 'UK';`. Joining tables is often more efficient than using subqueries for this type of filtering, especially with appropriate indexes.

   - It also records the execution time.

5. **Verifies Results:**

   - It compares the results of the original and optimized queries using `set()` to ensure they return the same set of books.

6. **Performance Summary:**

   - It prints the execution times for both queries, highlighting that the optimized query using JOIN and indexes is expected to be faster, particularly on larger datasets.

7. **Prints Results:** It prints the books found by the optimized query.

8. **Closes connection:** The database connection is closed.