Lab 003: Solution

COMP1048: Databases and Interfaces (2024-2025)

Matthew Pike & Yuan Yao

Table of contents

1 Lab Overview

In this lab, you will work with SQL to `CREATE`, `INSERT`, and `SELECT` data within a database. You'll be provided with a problem description and required to create the necessary tables, insert data, and select specific information using SQL queries.

The purpose of this lab is to give you hands-on experience with SQL and help you understand how to create and manipulate data in a database. By the end, you should have a clearer understanding of basic SQL operations and be able to apply these skills to real-world database management. You will also gain practical knowledge of using SQLite for these tasks.

2 Getting Started with SQLite

In lab 001, you installed SQLite and ensured it was functioning correctly. Please refer back to lab 001 if you need to install SQLite on your machine. In this lab, you'll use SQLite to create and manipulate data in a database. Follow the steps below based on your operating system to get started.

2.1 UNIX (Linux, macOS, etc.)

To start SQLite on UNIX-based systems, open a CLI (instructions in lab 001) and navigate to the directory where you want to create your database file using the `cd` command. For example:

```
cd /DBI/Labs/LBDBI_004-SQL1
```

2.2 Windows

Create a directory where you want to store your database file. For ease of use, we recommend copying the SQLite executable (`sqlite3.exe`) to the same directory where you want to store your database file. Open the command prompt and navigate to the directory you just created.

2.3 Creating a Database (All Platforms)

> **File Extensions**
>
> In this module, we will use the `.db` file extension to represent SQLite database files. While SQLite does not require this extension, it is a helpful convention. Similarly, we will store our SQL code in `.sql` files. Although SQLite does not mandate this either, following these conventions makes it easier to differentiate between file types.
> It's important not to confuse the two: - `.db` files contain the **database data**. - `.sql` files contain **SQL code** (queries, commands, etc.).

Once you're in the desired directory, run the following command to start the SQLite command line interface:

```
sqlite3 lab004.db
```

On windows, if you double clicked on the `sqlite3.exe` file, it will open a command prompt window with the SQLite prompt. You can then open the database file by running the following command (this also works on UNIX-based systems):

```
.open lab004.db
```

This will create a file name `lab004.db` in the directory where you ran the `sqlite3` command. All future commands run during this session will take effect on the data stored in `lab004.db`, until you exit the `sqlite3` CLI or open a new connection to a different database file. To exit the `sqlite3` CLI, execute the following command:

```
.quit
```

For a complete list of the available dot commands in `sqlite3`, use the `.help` command. A selection of relevant dot commands are shown below (Note: This is not an complete list of commands). For a complete list of dot commands, please refer to the SQLite documentation - https://sqlite.org/cli.html.

```
sqlite> .help

.cd DIRECTORY            Change the working directory to DIRECTORY
.databases               List names and files of attached databases
.dump ?OBJECTS?          Render database content as SQL
.excel                   Display the output of next command in spreadsheet
.headers on|off          Turn display of headers on or off
.help ?-all? ?PATTERN?   Show help text for PATTERN
.import FILE TABLE       Import data from FILE into TABLE
.mode MODE ?OPTIONS?     Set output mode
.open ?OPTIONS? ?FILE?   Close existing database and reopen FILE
.output ?FILE?           Send output to FILE or stdout if FILE is omitted
.quit                    Exit this program
.read FILE               Read input from FILE or command output
.save ?OPTIONS? FILE     Write database to FILE (an alias for .backup ... )
.schema ?PATTERN?        Show the CREATE statements matching PATTERN
.separator COL ?ROW?     Change the column and row separators
.show                    Show the current values for various settings
.tables ?TABLE?          List names of tables matching LIKE pattern TABLE
```

In this lab, you should write your SQL solutions in a `.sql` file and execute them using the `.read` command within the SQLite CLI. This ensures you can easily re-run your code without re-typing it. For example, to execute a file named lab004.sql (following the naming convention described in Section 4), use:

```
.read lab004.sql
```

This will run all SQL statements in the file sequentially. Your required workflow is:

1. Write your SQL statements in a `.sql` file.
2. Execute the file using `.read`.
3. Check the output to ensure correctness.
4. Repeat until all exercises are completed.

> **Improving Output Formatting**
>
> You can improve the formatting of your output by using the following commands at the beginning of your `.sql` file:
>
> ```
> .headers on
> .mode column
> ```

3 Exercises

3.1 Creating Tables with CREATE

Based on the problem description below, write the necessary SQL code to create **two tables** for storing the required data. Be sure to include primary and foreign key attributes, and choose appropriate data types based on the sample data in Section 3.2. Also, consider any necessary constraints.

> "A film production company wants to create a database to store details of its movie collections. Each movie must include: price, title, year, and genre. Each movie will have one leading actor, and each leading actor may appear in multiple movies. Leading actors have names (only), but different actors can share the same name."

Solution

```sql
-- Drop any tables that exist from a previous session
DROP TABLE IF EXISTS Movie;
DROP TABLE IF EXISTS Actor;

CREATE TABLE Actor (
    actId INTEGER PRIMARY KEY,
    actName TEXT NOT NULL
);

CREATE TABLE Movie (
    movId INTEGER PRIMARY KEY,
    movTitle TEXT NOT NULL,
    movPrice REAL NOT NULL,
    movYear INTEGER NOT NULL,
    movGenre TEXT NOT NULL,
    actId INTEGER,
    CONSTRAINT fk_mv_act
        FOREIGN KEY (actId) REFERENCES Actor (actId)
);
```

3.2 Inserting Data with INSERT

> **Make reasonable assumptions**
>
> With the example data provided below, you will need to make reasonable assumptions about how they are represented in your database. For example, it's likely that rows 1 and 2 in the table below refer to the same 'Barry Nelson', so should you store them as one or two separate actors?

Using the tables you created in Section 3.1, write the SQL `INSERT` statements to add the following data:

title	price	year	genre	leading_actor
Die Hard with a Vengeance	12.24	1999	Action	Barry Nelson
Black Snake Moan	9.99	2007	Adventure	Barry Nelson
Snakes on a Plane	9.99	2011	Comedy	Arethan Franklin
Freeway of Love	9.99	2018	Drama	Bullet Prakash
I knew you were waiting for me	12.25	1997	Comedy	Daniel Craig
The Black Panther	10.99	2018	Action	James Bond
The Jungle Book	9.99	2015	Adventure	Jonny Walker
Infinity War	8.5	1975	Horror	Laura Dern
Coming to Europe	12.99	2001	Adventure	Laura Dern
The Midnight	10.99	2019	Drama	Ryan Reynolds

Table 1: Data to be inserted into the database

Solution

```
INSERT INTO
    Actor (actName)
VALUES
    ('Arethan Franklin'),
    ('Barry Nelson'),
    ('Bullet Prakash'),
    ('Daniel Craig'),
    ('James Bond'),
    ('Jonny Walker'),
    ('Laura Dern'),
    ('Ryan Reynolds');

INSERT INTO
    Movie (actId, movTitle, movGenre, movYear, movPrice)
VALUES
    -- Arguably the approach used below for getting
    -- the actId is not the best.
    -- Can you think why this may be problematic?
    ((SELECT actId from Actor WHERE actName = 'Barry Nelson'),
              'Die Hard with a Vengeance', 'Action',1999,12.24),
    ((SELECT actId from Actor WHERE actName = 'Barry Nelson'),
              'Black Snake Moan', 'Adventure', 2007, 9.99),
    ((SELECT actId from Actor WHERE actName = 'Arethan Franklin'),
              'Snakes on a Plane', 'Comedy',2011, 9.99),
    ((SELECT actId from Actor WHERE actName = 'Bullet Prakash'),
              'Freeway of Love', 'Drama', 2018, 9.99),
    ((SELECT actId from Actor WHERE actName = 'Daniel Craig'),
              'I knew you were waiting for me', 'Comedy', 1997, 12.25),
    ((SELECT actId from Actor WHERE actName = 'James Bond'),
              'The Black Panther', 'Action', 2018,10.99),
    ((SELECT actId from Actor WHERE actName = 'Jonny Walker'),
              'The Jungle book', 'Adventure', 2015, 9.99),
    ((SELECT actId from Actor WHERE actName = 'Laura Dern'),
              'Infinity War', 'Horror', 1975, 8.50),
    ((SELECT actId from Actor WHERE actName = 'Laura Dern'),
```

```
            'Coming to Europe', 'Adventure', 2001, 12.99),
    ((SELECT actId from Actor WHERE actName = 'Ryan Reynolds'),
            'The Midnight', 'Drama',2019, 10.99);
```

3.3 Answering questions using SELECT

> **Enforcing Foreign Key Constraints**
>
> In SQLite, foreign key constraints are off by default. Enable them with `PRAGMA foreign_keys = ON;`. This must be done for each new database connection.

Using the tables you created in Section 3.1 and populated in Section 3.2, write SQL `SELECT` statements to answer the following:

1. List the `title` and `year` of all movies.

Solution

```
SELECT movTitle, movYear FROM Movie;
```

2. List the `title` of movies with a `price` greater than 10.

Solution

```
SELECT movTitle FROM Movie WHERE movPrice > 10;
```

3. List the `title` of movies with a price less than or equal to 9.99 in the 'Adventure' `genre`.

Solution

```
SELECT movTitle
FROM Movie
WHERE
    movPrice ⩾ 9.99
    AND
    movGenre = 'Adventure';
```

4. List the `title` and `genre` of all movies in the `Action` or `Comedy` genres.

Solution

```sql
SELECT movTitle, movGenre
FROM Movie
WHERE
    movGenre = 'Action'
    OR
    movGenre = 'Comedy';
```

5. List the name of all actors, removing duplicates, and sorting in descending order.

Solution

```sql
SELECT DISTINCT actName FROM Actor ORDER BY actName DESC;
```

4 Submitting your lab work

Submit a single **.sql** file containing the CREATE, INSERT and SELECT statements you wrote in Section 3.1, Section 3.2 and Section 3.3 respectively. Complete solutions will first DROP any existing tables, then CREATE the tables, INSERT the data, and finally SELECT the data as specified in the questions. There is no need to include the output of your SQL statements in your submission.

Compile your solution into a single SQL file. Your submission should include the following details at the top of the file in a comment block:

- Your name
- Student ID
- University email address
- Module code (COMP1048) and title (Databases and Interfaces)
- Lab number (004) and title (SQL CREATE, INSERT and SELECT')
- Date of submission

In SQL, comments are written using `--` for single line comments, and `/* */` for multi-line comments. For example, the following is one way for you to include the required details at the top of your SQL file:

```sql
-- Name: John Smith
-- Student ID: z123456
-- Email:
-- Module: COMP1048 Databases and Interfaces
-- Lab 004: SQL CREATE, INSERT and SELECT
-- Date:

.mode column
.headers on

DROP TABLE IF EXISTS ... ;

CREATE TABLE ... ;

INSERT INTO ... ;
```

```
SELECT ... ;
```

Name your SQL file using the following format - `DBI_lab004-<student_id>.sql`, where, `<student_id>` is your student ID. For example, if your student ID is `z123456`, you should name your SQL file `DBI_lab004-z123456.sql`.

Please ensure you submit your work by the deadline - 4 November 2024 at 15:00. Late submissions will not be accepted, as stated on the coursework issue sheet. Your submission must be less than 500kb in size.

Submissions that are unreadable, corrupted, missing the required details, or do not demonstrate a reasonable attempt to answer all questions will receive a mark of zero for the lab.

This lab contributes 1% of your overall module grade.