SQLite implements a **transactional** SQL database engine. A transactional database is one in which all changes and queries appear to be **Atomic, Consistent, Isolated, and Durable (ACID)**. SQLite implements serializable transactions that are atomic, consistent, isolated, and durable, even if the transaction is interrupted by a program crash, an operating system crash, or a power failure to the computer. In other words, all changes within a single transaction in SQLite either occur completely or not at all, even if the act of writing the change out to the disk is interrupted by a program crash, an operating system crash, or a power failure.

- Transaction: A transaction is a *logical unit of work* on the database. It may be a single statement (e.g., one INSERT, UPDATE or DELETE statement) or it may involve a number of operations on the database. A transaction should always transfer the database from one consistent state to another.
- There are four basic, or so-called ACID, properties that all transactions should possess:
  - **Atomicity: '**All or nothing property'. A transaction is an indivisible unit that is either performed in its entirety or it is not performed at all.
  - **Consistency:** A transaction must transform the database from one consistent state to another consistent state.
  - **Isolation:** Transactions execute independently of one another (the partial effects of incomplete transactions should not be visible to other transactions).
  - **Durability**: The effects of a successfully completed (committed) transaction are permanently recorded in the database and must not be lost because of a subsequent failure.

SQLite provides two statements for transaction control – **COMMIT** and **ROLLBACK**. The operation performed by INSERT, UPDATE and DELETE results in a transaction. In SQLite, all CRUD (CREATE, RETREIEVE, UPDATE and DELETE) operations first take effect in memory and then they are permanently saved (committed) the secondary storage. SQLite transactions are automatically committed without giving any chance to undo (rollback) the changes.

To control commit and rollback, start transactions after issuing the statement **BEGIN TRANSACTION**. Subsequent operations will not be written to the disk until **COMMIT** is issued and will be reverted back to the previous state of the database (i.e. the state before BEGIN TRANSACTION) if **ROLLBACK** is issued.

Note: We are using *SQLite console* to learn SQL. Although SQLite does not provide its own Graphical User Interface (GUI) for database management, you can use a third-party open-source tool like *SQLiteStudio* (https://sqlitestudio.pl/) - installed on Exeter WVD. However, the best way to learn SQL is using a command line interface!

References:

- *https://www.sqlite.org/index.html*
- *https://www.sqlitetutorial.net*
- *Connolly, T., Begg, C. E. and Holowczak, R. 2008. Business Database Systems. Harlow, England; New York : Addison-Wesley.*
- *Lathkar M. 2019. Python Data Persistence. BPB Publications.*

1. Access shell (**sqlite3.exe**). Open the database created in Week 4 (BEMM459.db) using command line shell for SQLite (change path to point to your OneDrive). **Sqlite>** is the SQLite prompt.

sqlite> .open "C:\\Users\\nm319\\OneDrive - University of Exeter__Exeter Teaching (2020-21)\\Lab Sessions\\Week 4\\BEMM459.db" (use path specific to your DB file)

sqlite> .exit (exits SQLite console)

SQL Transaction Control Language (TCL)

2. ROLLBACK

Display the contents of the table.

sqlite> Select * from Customer;

sqlite> BEGIN TRANSACTION;

Update, Insert and Delete records.

sqlite> Insert into Customer values (25,"Testing Rollback", "Test Rollback", "N");

sqlite> Insert into Customer values (26,"Testing Rollback", "Test Rollback", "N");

sqlite> Update Customer set customerFirstName = "Lion King" where customerFirstName = "Hello";

sqlite> Delete from Customer where customerNum = 22;

Display the contents of the table – what do you see?

sqlite> Select * from Customer;

Issue rollback command.

sqlite> ROLLBACK;

Display the contents of the table – what do you see?

sqlite> Select * from Customer;

3. COMMIT

Sqlite> BEGIN TRANSACTION;

Update, Insert or Delete records.

Sqlite> Update Customer set customerLoyalty="Y" where customerNum=10;

Issue commit command.

Sqlite> COMMIT;

Display the contents of the table – what do you see?

Sqlite> Select * from Customer;

A few more useful SQLite Commands

4. Dump entire database into a file

(By default, the .dump command outputs the SQL statements on screen. To issue the output to a file, you use the .output FILENAME command)

sqlite> .open **<database name>** (if not already connected with the database you wish to dump)

sqlite> .output **<filename>** (the filename to dump data to)

sqlite> .dump

5. Dump a specific table and data using the SQLite dump command

sqlite> .output **<filename>** (the filename to dump data to)

sqlite> .dump **<tablename>** (table whose data will be dumped)

6. Dump tables structures only

sqlite> .output **<filename>** (the filename to dump data to)

sqlite> .schema (only structure of tables in a database are dumped)

7. Dump only data from one or more tables

sqlite> .mode insert

(Set the mode to insert using the .mode command. From now on, every SELECT statement will issue the result as the INSERT statements instead of pure text data.)

sqlite> .output **<filename>**

(The filename to dump data to. All subsequent queries will be saved to the file specified.)

sqlite> select * from **<tablename>**

sqlite> .output (all subsequent commands will be displayed in the SQLite shell)

sqlite> .once **<filename>** (saves the result of the next query only to <filename>)

8. Backup database

sqlite> .backup **<backup database>** (command backs up the current database to a new database)

9. Execute SQL command from a file.

sqlite> .read **<filename with command>**

(The filename should consist of a SQL statement and should terminate with ;)

/End