SQLite is an in-process library that implements a **self-contained (stand-alone)**, **serverless**, **zero-configuration**, **transactional** SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is an embedded SQL database engine. It reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. The database file format is cross-platform - you can freely copy a database between 32-bit and 64-bit systems.

- **Stand-alone/self-contained:** SQLite has very few dependencies. It runs on any operating system, even stripped-down bare-bones embedded operating systems. The entire SQLite library is encapsulated in a single source code file that requires no special facilities or tools to build.
- **Serverless**: Most SQL database engines are implemented as a separate server process (Figure 1). Programs that want to access the database communicate with the server using some kind of interprocess communication (typically TCP/IP) to send requests to the server and to receive back results. SQLite does not work this way. With SQLite, the process that wants to access the database reads and writes directly from the database files on disk. There is no intermediary server process (Figure 2).
    - The main advantage is that there is no separate server process to install, setup, configure, initialize, manage, and troubleshoot. This is one reason why SQLite is a "zero-configuration" database engine. Programs that use SQLite require no administrative support for setting up the database engine before they are run. Any program that is able to access the disk is able to use an SQLite database.

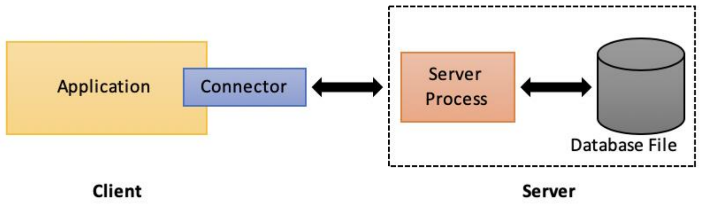


Figure 1: RDBMS Client-Server Architecture

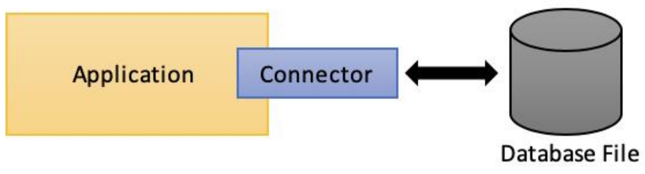


Figure 2: SQLite server-less architecture

- **Zero-configuration:** SQLite does not need to be "installed" before it is used. There is no "setup" procedure. There is no server process that needs to be started, stopped, or configured. There is no need for an administrator to create a new database instance or assign access permissions to users. SQLite uses no configuration files. Nothing needs to be done to tell the system that SQLite is running. No actions are required to recover after a system crash or power failure. There is nothing to troubleshoot.

- **Transactional:** 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 SQL INSERT or UPDATE 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 (Figure 3).

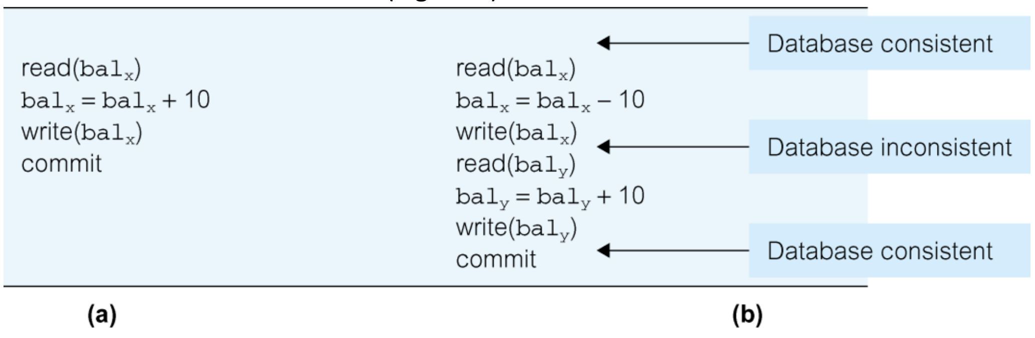


Figure 3: Example of a simple transaction (a) and a more complicated transaction (b)

    - 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.
    - Serializable transactions: When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which schedules are serializable. A serializable schedule is the one that always leaves the database in consistent state.

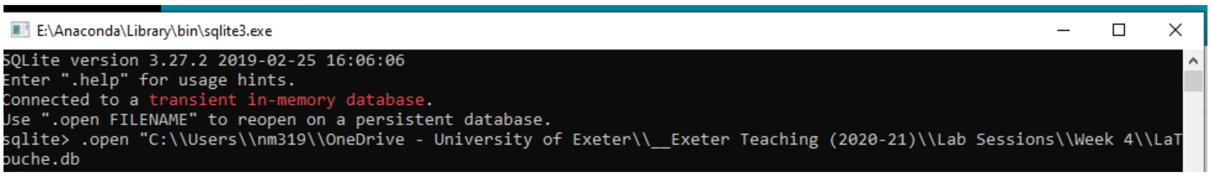
References:

- *https://www.sqlite.org/index.html*
- *https://www.spheregen.com/sqlite/*
- *Connolly, T., Begg, C. E. and Holowczak, R. 2008. Business Database Systems. Harlow, England; New York : Addison-Wesley.*

Command Line Shell for SQLite

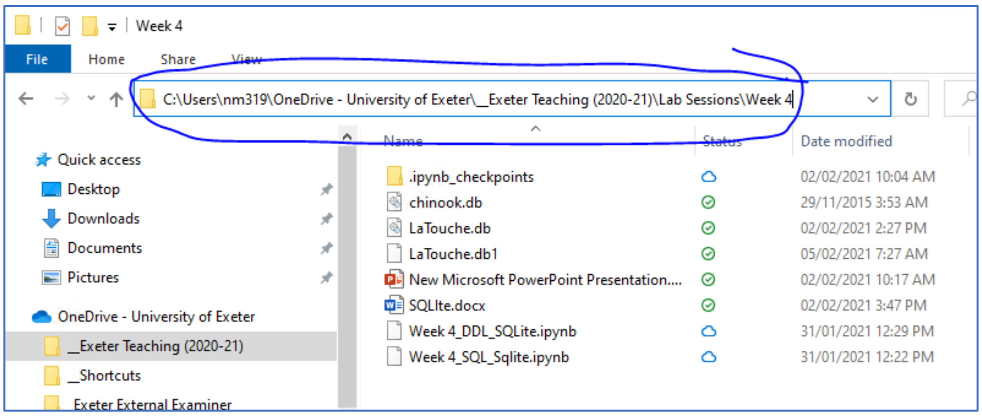
1. Access shell (**sqlite3.exe**). Open an existing db using command line shell for SQLite (change path to point to your OneDrive). **Sqlite>** is the SQLite prompt.

Sqlite> .open "C:\\Users\\nm319\\OneDrive - University of Exeter\\__Exeter Teaching (2020-21)\\Lab Sessions\\Week 4\\LaTouche.db"



```
E:\Anaconda\Library\bin\sqlite3.exe
SQLite version 3.27.2 2019-02-25 16:06:06
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open "C:\\Users\\nm319\\OneDrive - University of Exeter\\__Exeter Teaching (2020-21)\\Lab Sessions\\Week 4\\LaT
ouche.db
```

- Note 1: Create a folder in one drive and then double-click the file explorer bar. You will then see the correct path that is to be used (see screenshot below and the blue highlighted text). Copy your path (which will be different than the one below). When using **.open**, remember to use double-quote and introduce an additional \ in the path (\\).
    - .open "C:\\Users**\\**nm319\\OneDrive - University of Exeter**\\**__Exeter Teaching (2020-21)\\Lab Sessions\\Week 4\\LaTouche.db"


Week 4
File
Home
Share
View
C:\Users\nm319\OneDrive - University of Exeter__Exeter Teaching (2020-21)\Lab Sessions\Week 4
Quick access
Desktop
Downloads
Documents
Pictures
OneDrive - University of Exeter
__Exeter Teaching (2020-21)
__Shortcuts
Exeter External Examiner

Name	Status	Date modified
.ipynb_checkpoints		02/02/2021 10:04 AM
chinook.db		29/11/2015 3:53 AM
LaTouche.db		02/02/2021 2:27 PM
LaTouche.db1		05/02/2021 7:27 AM
New Microsoft PowerPoint Presentation....		02/02/2021 10:17 AM
SQLIte.docx		02/02/2021 3:47 PM
Week 4_DDL_SQLite.ipynb		31/01/2021 12:29 PM
Week 4_SQL_Sqlite.ipynb		31/01/2021 12:22 PM

- Note 2: If the file name does not exist (e.g., LaTouche.db) then a new db file is created.

2. Show database that is connected and .help to display all SQLite command prompt commands

Sqlite> .database

Sqlite> .help

3. Create three tables

```
CREATE TABLE <table_name> (
Attribute1 datatype [width] [default] [constraint],
Attribute2 …,
…
);
```

Sqlite> CREATE TABLE MenuItem(

menuItemNum INTEGER PRIMARY KEY,

menuItemName TEXT (30),

menuItemCost REAL,

menuItemAvailable TEXT (1));

Sqlite> CREATE TABLE Customer(

customerNum INTEGER PRIMARY KEY AUTOINCREMENT,

customerFirstName TEXT (30),

customerLastName TEXT (30),

customerLoyalty TEXT (1));

Sqlite> CREATE TABLE CustomerInvoice(

customerInvoiceNum INTEGER PRIMARY KEY AUTOINCREMENT,

CI_CustomerNum INTEGER,

menuItemNum INTEGER,

CustomerInvoiceQty INTEGER,

FOREIGN KEY (CI_CustomerNum) REFERENCES Customer(CustomerNum),

FOREIGN KEY (menuItemNum) REFERENCES MenuItem(menuItemNum));

4. Display the structure of the tables and structure of tables

Sqlite>.tables (querying tables using SQLite command prompt command)

Sqlite>.schema (**or** .schema <table name> **or** .schema <partial table name>%, e.g., .schema M%)

Sqlite> select * from **sqlite_master** WHERE type='table'; (using SQL to query tables from data dictionary)

5. Inserting Records (take note of database response when integrity constraints violate)

Sqlite> Insert into MenuItem (menuItemNum, menuItemName, menuItemCost, menuItemAvailable) VALUES (1, “Coffee”, 2.50, “Y”);

```
INSERT INTO  <table_name> (attribute1, attribue2,…) VALUES (val1, val2,…);
```

Sqlite> Insert into MenuItem VALUES (2, “Tea”, 1.50, “Y”);

Sqlite> Insert into MenuItem (menuItemNum, menuItemName, menuItemAvailable) VALUES (3, “Hot Chocolate”, “Y”);

Sqlite> Insert into Customer (customerFirstName, customerLastName, customerLoyalty) values (“Hello”, “World”, “Y”);

Sqlite> Insert into Customer (CustomerNum, customerFirstName, customerLastName, customerLoyalty) values (2, “Second”, “Customer”, “Y”);

Sqlite> Insert into Customer values (10, “Tenth”, “Customer”, “N”);

Sqlite> Insert into Customer (customerFirstName, customerLastName, customerLoyalty) values (“Hello”, “World”, “Y”);

- Note: next customer number will be 11 (as previous was 10). Please also use your own examples

Sqlite> Insert into CustomerInvoice (CI_CustomerNum, menuItemNum, CustomerInvoiceQty) values (11,3,2);

5. Foreign key support (referential integrity) has to be explicitly enabled.

- Foreign Key support: https://sqlite.org/foreignkeys.html

Enable foreign key support

```
Sqlite> PRAGMA foreign_keys = ON;
```

Sqlite> Insert into CustomerInvoice values (1,10,1,2);

- FOREIGN KEY Constraint Failed (if either CI_CustomerNum or menuItemNum does not exisit).
- UNIQUE Constraint Failed if customerInvoiceNum already exisits.

Sqlite> Insert into CustomerInvoice (CI_CustomerNum, menuItemNum, CustomerInvoiceQty) values (2,4,2);

- FOREIGN KEY Constraint Failed (if either CI_CustomerNum or menuItemNum does not exist in the parent tables).

5. Display records

Sqlite>.header on

Sqlite>.mode column

6. Select statements

SELECT attribute1, attribute2, …, attributeN **FROM** <table_name> **WHERE** [expression];

Sqlite> select * from customerInvoice;

Sqlite> select customerinvoiceNum, customerinvoiceqty from customerInvoice;

Sqlite> select * from MenuItem **order by menuItemName**;

Sqlite> select * from MenuItem **order by menuItemName DESC**;

6A. Operators: <, >, <=, >=, =, <>, IN, LIKE, IS NULL

Sqlite> select * from MenuItem where **menuItemName = "Tea"**;

Sqlite> select * from MenuItem where **menuItemName LIKE "C%"**;

Sqlite> select * from MenuItem where **menuItemNum IN (1,5,10,21)**;

Sqlite> select * from MenuItem where **menuItemCost > 2**;

Sqlite> select * from MenuItem where **menuItemCost <> 2**;

Sqlite> select * from MenuItem where **menuItemCost IS NULL**;

6B. Joins

Sqlite> Select customerInvoiceNum, CustomerInvoice .menuItemNum, MenuItem .menuItemName, MenuItem.menuItemCost, CustomerInvoiceQty, CI_CustomerNum from **CustomerInvoice, MenuItem** where **CustomerInvoice. menuItemNum= MenuItem. menuItemNum**;

Sqlite> Select customerInvoiceNum, CustomerInvoice .menuItemNum, MenuItem .menuItemName, MenuItem.menuItemCost, CustomerInvoiceQty, CI_CustomerNum, CustomerNum, customerFirstName, customerLastName from **CustomerInvoice, MenuItem, Customer** where **CustomerInvoice. menuItemNum= MenuItem. menuItemNum**;

- Cartesian product (above example)

Sqlite> Select customerInvoiceNum, CustomerInvoice .menuItemNum, MenuItem .menuItemName, MenuItem.menuItemCost, CustomerInvoiceQty, CI_CustomerNum, CustomerNum, customerFirstName, customerLastName from CustomerInvoice, MenuItem, Customer where **CustomerInvoice. menuItemNum= MenuItem. menuItemNum** AND **CustomerInvoice. CI_CustomerNum= Customer. customerNum**;

Sqlite> Select customerInvoiceNum, CustomerInvoice .menuItemNum, MenuItem .menuItemName, MenuItem.menuItemCost, CustomerInvoiceQty, CI_CustomerNum, **Customer.**customerFirstName, **Customer.**customerLastName from CustomerInvoice, MenuItem, Customer where CustomerInvoice. menuItemNum= MenuItem. menuItemNum AND CustomerInvoice. CI_CustomerNum= Customer. customerNum;

- Good practice – prefix table names

6C. Calculated columns

Sqlite> Select customerInvoiceNum, MenuItem .menuItemName, MenuItem.menuItemCost, CustomerInvoiceQty, **MenuItem.menuItemCost* CustomerInvoiceQty as TOTAL**, Customer.customerFirstName, Customer.customerLastName from CustomerInvoice, MenuItem, Customer where CustomerInvoice. menuItemNum= MenuItem. menuItemNum AND CustomerInvoice. CI_CustomerNum= Customer. customerNum;

7. Update Statements

UPDATE <table_name> **SET** attribute1=val1, attribute2=val2, … ,attributeN=valN **WHERE** [expression];

Sqlite> Update MenuItem set menuItemCost=4.75 **where menuItemNum=3**;

Sqlite> Update MenuItem **set menuItemCost= menuItemCost+(menuItemCost*20/100)** where menuItemNum=3;

- Increasing price of an item by 20% (above)

Sqlite> Update MenuItem set **menuItemCost= menuItemCost+(menuItemCost*20/100)**;

- Increasing all prices by 5% (no Where clause)

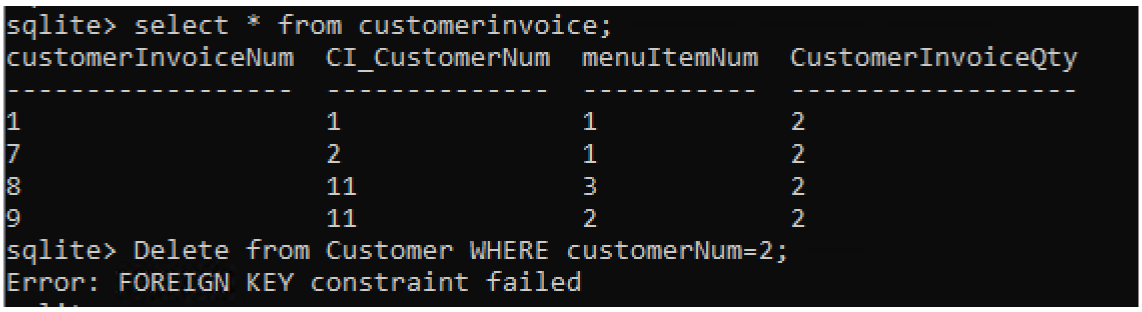
8. Delete Statement

DELETE FROM <table_name> **WHERE** [expression];

Note: WHERE clause should be specified unless the intention is to remove all the records from the table.

Sqlite> Delete from Customer WHERE customerNum=2;

- FOREIGN KEY constraint. Cannot delete record from the parent table (CUSTOMER) as child table (customerInvoice) has customerNum =2.



```
sqlite> select * from customerinvoice;
customerInvoiceNum  CI_CustomerNum  menuItemNum  CustomerInvoiceQty
------------------  --------------  -----------  ------------------
1                   1               1            2
7                   2               1            2
8                   11              3            2
9                   11              2            2
sqlite> Delete from Customer WHERE customerNum=2;
Error: FOREIGN KEY constraint failed
```

9. ALTER TABLE Statement

ALTER TABLE <table_name> **ADD COLUMN** attributeNew datatype [width] [default] [constraint];

Sqlite> ALTER TABLE Customer ADD COLUMN customerMembership text [20];

Sqlite> .schema Customer

10. DROP TABLE Statement

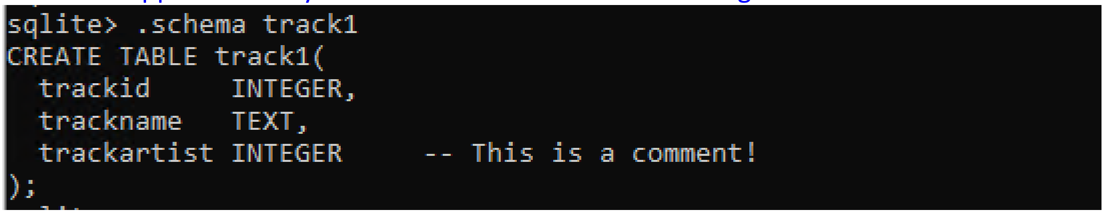
DROP TABLE <table_name> ;

Sqlite> **DROP TABLE** Customer;

11. Including a comment

```
CREATE TABLE track1(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER     -- This is a comment!
);
```

Comment appears when you enter .schema command. Make good use of comments.



```
sqlite> .schema track1
CREATE TABLE track1(
  trackid      INTEGER,
  trackname    TEXT,
  trackartist INTEGER     -- This is a comment!
);
```

/End