

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.

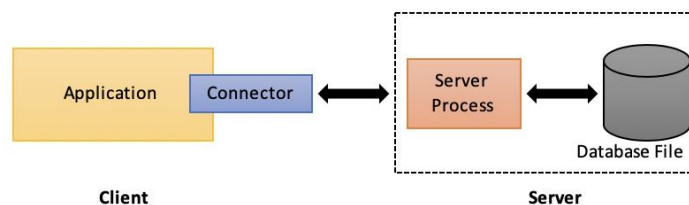


Figure 1: RDBMS Client-Server Architecture

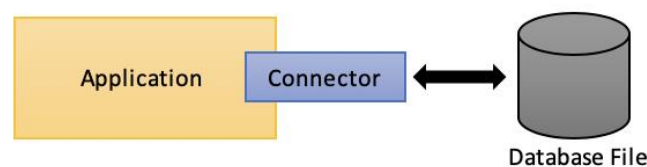


Figure 2: SQLite server-less architecture

- **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).

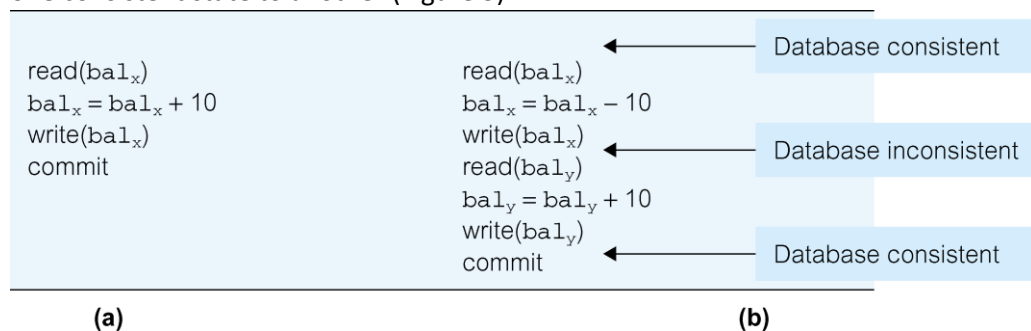


Figure 3: Example of a simple transaction (a) and a more complicated transaction (b)

- 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:

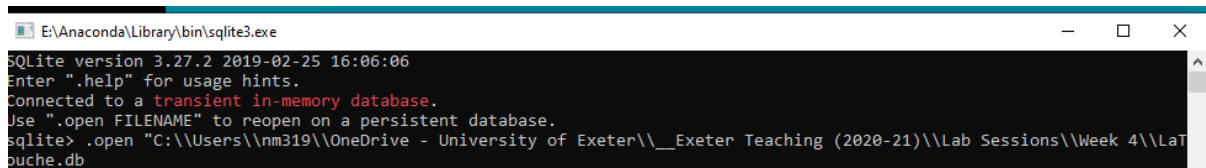
- <https://www.sqlite.org/index.html>
- <https://www.spheregen.com/sqlite/>
- Connolly, T., Begg, C. E. and Holowczak, R. 2008. *Business Database Systems*. Harlow, England; New York : Addison-Wesley.

## Command Line Shell for SQLite

-- We are going to use an existing database called chinook.db --

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 "S:\\SQLite\\Week 3\\chinook.db"



- Create a folder in your **S:** drive (Cloud persistent storage) and copy **chinook.db** from week 3 Github repository (<https://github.com/NavonilNM/bemm459J-Term3>)
- When using **.open**, remember to use double-quote and introduce an additional \ in the path (\\).
- If the file name does not exist (e.g., chinook.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. 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 M%**)

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

4. Display records

**Sqlite>**.header on

**Sqlite>**.mode column

**Sqlite>** select \* from genres;

-- Next, we are going to create a new table --

## 5. Create a new database

**Sqlite>** .open "S:\\SQLite\\Week 3\\bemm459J.db"

*(The name of the database is **bemm459j.db**)*

## 6. Create a table

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

**Sqlite>** CREATE TABLE Customer(  
 customerNum INTEGER PRIMARY KEY AUTOINCREMENT,  
 customerFirstName TEXT (30),  
 customerLastName TEXT (30),  
 customerLoyalty TEXT (1));

### 6A. 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!  
);
```

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

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

**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");

**Sqlite>** Insert into Customer (customerFirstName, customerLastName, customerLoyalty) values ("Hello", "World", NULL);

- Note: next customer number will be 11 (as previous was 10)

## 8. Select statements

```
SELECT attribute1, attribute2, ..., attributeN FROM <table_name> WHERE [expression];
```

**Sqlite>** select \* from Customer;

**Sqlite>** select customerFirstName, customerLastName from Customer;

**Sqlite>** select \* from Customer **order by** customerFirstName;

**Sqlite>** select \* from Customer **order by** customerFirstName **DESC**;

*(Try out other SQL statements).*

### 8A. Operators: <, >, <=, >=, =, <>, IN, LIKE, IS NULL

**Sqlite>** select \* from Customer where **CustomerNum** = 10;

**Sqlite>** select \* from Customer where **customerFirstName** **LIKE** "H%";

**Sqlite>** select \* from Customer where **CustomerNum** **IN** (1,10,21);

**Sqlite>** select \* from Customer where **CustomerNum** > 2;

**Sqlite>** select \* from Customer where **CustomerNum** <> 2;

**Sqlite>** select \* from Customer where **CustomerLoyalty** **IS NULL**;

**Sqlite>** select \* from Customer where **CustomerLoyalty** **IS NOT NULL**;

**UPDATE** <table\_name> **SET** attribute1=val1, attribute2=val2, ... ,attributeN=valN **WHERE** [expression];

**Sqlite>** Update Customer set **CustomerLoyalty** = "Y" where **CustomerNum** = 12;

**Sqlite>** Update Customer set **CustomerLoyalty** = "N";

- Change all records (Where clause is not used)

## 10. 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;

## 11. 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

**Sqlite>** Update Customer set **customerMembership** = "New Member";

## 12. DROP TABLE Statement

**DROP TABLE** <table\_name> ;

**Sqlite>** **DROP TABLE** Customer;