SQLite

SQLite is a really good tool for "just getting stuff done", especially when you hit the limits of Microsoft Excel (or other spreadsheet software).

According to its authors, SQLite is used in some form in every single Windows 10, Mac, Android and iOS device as well as built into the Firefox, Chrome (including Edge) and Safari browsers: the browsers store their browsing history and cookies in SQLite-format databases and you can manually edit these with the command line tool if you like.

Currently in version 3.34.1, the command-line program is called `sqlite3` and exists for all major operating systems, you should be able to download it as a single file executable, put in in any folder you like and just run it. On Alpine, you can install the `sqlite` package with `apk`.

SQLite is an embedded database, which means there is no separate server process. Databases are simply files. When you run the command line tool, the tool fulfils both the client and server roles at once; when you use SQLite from a program, the SQLite driver also acts as the "server".

Census exercise

You will need the census files from databases activity 1: make a folder with the `setup.sql` file and the census csv files.

Download or install sqlite and run `sqlite3 census.db` to create a database file.

To create the tables, the command `.read setup.sql` reads and executes the commands in a file (similar to `source` on the shell). SQLite system commands start with a dot and do not take a semicolon at the end - see `.help` for a list. (Don't run the `import.sql` file, that's MariaDB-specific.)

Use `.tables` to show the tables and check that they have been created. With `.schema Ward` for example you can show the create statement for a single table.

The source data is in simple CSV files, for example if (back on the shell) you did a `head -n 5 County.csv` you would see this:

```
E09000001,"City of London",E12000007,E92000001
E09000002,"Barking and Dagenham",E12000007,E92000001
E09000003,Barnet,E12000007,E92000001
E09000004,Bexley,E12000007,E92000001
E09000005,Brent,E12000007,E92000001
```

which matches the following, from the setup script:

```
CREATE TABLE County (
    code VARCHAR(10) PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    parent VARCHAR(10),
    country VARCHAR(10),

    FOREIGN KEY (parent) REFERENCES Region(code),
    FOREIGN KEY (country) REFERENCES Country(code)
);
```

To import the data, run the following on the SQLite prompt:

```
.mode csv
.header off
.import Country.csv Country
.import Region.csv Region
.import County.csv County
.import Ward.csv Ward
.import Occupation.csv Occupation
.import Statistic.csv Statistic
```

You should now have all the data imported and you could try any SQL queries from the previous exercises in this database. The SQL dialect of SQLite is not entirely the same as MariaDB: for example, SQLite can do INNER and LEFT OUTER JOIN (and CROSS JOIN) but it cannot do RIGHT OUTER or FULL OUTER JOIN.

SQLite is also extremely forgiving when it comes to data types: it treats types more or less as comments, so it will let you store a string in an integer column if you try. You can also write a CREATE TABLE statement and just leave out the types, SQLite will not mind. SQLite also *does not enforce your FOREIGN KEY constraints by default*, though you can turn this on if you want to. These are all design choices made by the SQLite developers that might or might not be the best ones for any particular use case.

Modes and file output

SQLite has different ways of displaying a table of data. Try `.mode csv`, `.header on` then `SELECT * FROM Region;` to see the regions as a CSV file with headers (mode and header settings apply until you change them or quit SQLite). If you want to actually produce a CSV file, `.once FILENAME` tells SQLite to send the result of the following query only to a file, so you can work on a query until it does what you want, then run the `.once` command and re-run the query (SQLite supports readline on most platforms so the UP key gets the last line back) to send it to a file.

The default mode, `list`, shows columns separated by bar characters. You can also try `.mode column` which is a bit more human-readable, printing in lined-up columns with spaces in between. Or, if you want to include a table in a HTML page, `.mode html` prints the table with HTML tags. If you are on your host OS (not the alpine VM) and you have Excel installed, `.excel` opens the result of the following query in Excel.

You have now learned an extremely powerful way to analyze data that is presented to you in one or more CSV files - or any data format that you can easily export to CSV (for example most Excel files).

SQLite from Java

You can access a SQLite database from a Java program by using the JDBC driver. In Maven, declare the following dependency:

```
<dependency>
  <groupId>org.xerial</groupId>
  <artifactId>sqlite-jdbc</artifactId>
  <version>3.34.0</version>
</dependency>
```

The connection string takes the format `jdbc:sqlite:FILEPATH`. An absolute path to a file should always work (you have to spell it out and not use the `~` shortcut though), or if you are running a JAR file from the same folder as the database file then just giving the file name with no path should work too.

Exercise: write a minimal Maven/Java application that prints the region names from the sqlite database you just created.

SQLite from Python

Python is a great language for Data Science, and if you ever need to connect to a SQLite database from Python then here is how:

```
# you may have to 'pip install sqlite3'
import sqlite3
with sqlite3.connect('FILENAME') as connection:
    cursor = connection.cursor()
    for row in cursor.execute('SELECT * FROM Ward WHERE name LIKE ?',
WARDNAME):
        # do something with row
        print(row)
```

- The `with` statement is the equivalent to Java's try-with-resources.

- The `execute` statement executes a prepared statement. You put the statement itself in the first arguments, then the values to replace the placeholders as additional arguments.
- The `execute` command returns an iterator that you can loop over, the rows themselves are python tuples (you can also do `.fetchall()` to get them in a list).

See the Python sqlite documentation for more information.

One important warning here: if you write any data to the database, you have to call `connection.commit()` afterwards otherwise the data will not actually be written to disk.

SQLite from C

The final exercise here is to write a C program, following the instructions in the slides, that reads your SQLite `census.db`, iterates over the regions table and prints "Region NAME is code CODE", where NAME/CODE are replaced with the region's name and code.

Of course you need to handle potential errors from the `sqlite3_` functions: it's only safe to continue if they return `SQLITE_OK`.

You will need to install the `sqlite-dev` package which contains the `sqlite3.h` header file, and link against the library with `-lsqlite3` in your `gcc` command.

If you are using `sqlite3_exec` with a callback function, the callback must return 0 on success and nonzero if it encountered an error, which will in turn cause `sqlite_exec` to return an error too.

For other methods, see the SQLite C API. For example `sqlite3_step` lets you iterate over the rows of a SQL result without using a callback function.