Open in Colab

Lab 3: Becoming Dangerous with SQL

Review the lab material and go through the entire notebook. The lab contains 5 exercises for you to solve. The entire lab is worth 2.5% of your final grade and each exercise is worth 0.4% of your final grade. Going through the full notebook is worth 0.5% of your final grade. Any extra credit or bonus exercises are worth an additional 0.4%.

Labs are due by Friday at 11:59 PM PST and can be submitted on BCourses assignment page for the corresponding lab.

Data Persistence Techniques

Persisting data is an important task, and not just for data science applications. Programs may need to persist data to ensure state, to share information, and to improve performance. As a result, many different approaches exist for saving data, spanning everything from basic file input/output techniques to enterprise level database management software. In this lab, we explore some of these different techniques with the goal of leveraging them to facilitate data science investigations.

One of the simplest persistence techniques is basic file input/output. In Python, you can open a file for reading and writing and even use binary mode to save storage space (or even directly use a compression technique by using the appropriate Python library like bzip2). To recall, the following code segment demonstrates how to write data to a file called *test.dat*.

In [ ]:
```
data = """Data to write to the file, which can easily include any Python dat
by using string formatting techniques."""

with open('test.dat', 'w') as fout:
    fout.write(data)
```

and we can easily read data back into a Python program (and in this example, simply echo the text to *STDOUT*) in a similar manner:

In [ ]:
```
with open('test.dat', 'r') as fin:
    for line in fin:
        print(line)
```

```
Data to write to the file, which can easily include any Python datatype

by using string formatting techniques.
```

While this works, it is not optimal for several reasons:

1. All data is written and read as Python strings. Complex arrangements of heterogenous data thus require potentially complex (and costly in execution time) transformations.

2. All *concurrency* is provides by the file system, thus we are not guaranteed consistent results if multiple writers work at the same time.

3. Without extra effort, for example, to write to a binary file or to employ compression, this approach is costly in terms of storage space.

4. We rely completely on the underlying file system for *consistency* and *durability*. Thus, persisted application state may have unintentional dependencies on the underlying file system.

An alternative approach is available for more advanced data structures, like the NumPy array.

```
data = np.genfromtxt('rppdm/data/airports.csv', names=True,
    delimiter=',', dtype=None, invalid_raise=False)
```

And if we develop our own data types, we can create our own custom input/output routines to read/write any new objects we have created. But this can be a lot of extra work, especially to verify the routines work accurately as a program is continually developed or maintained. Furthermore, this doesn't solve all of the problems as we still rely entirely on the file system to maintain concurrency, consistency, and durability.

Pickling

Python provides a simple technique, called *pickling*, that we can use to easily save data to a file and to later reconstitute the data into a Python program. Pickling writes the *class* information for any data being written to the file along with the data. When you *unpickle* data, this class information is used to properly reconstitute the data in the pickled file. Pickling is easy to use and can often suffice for simple data persistence tasks. To pickle data to a file, you must import the pickle module and open a file in binary writing mode. After this, simply call the `pickle.dump()` method with the data to write and the file stream.

In [ ]:
```
import numpy as np
import pickle as pkl

data = np.random.rand(100)

with open('test.p', 'wb') as fout:
    pkl.dump(data, fout)
```

Unpickling data is also easy, simply open the appropriate file in binary read mode and call the `pickle.load()` method to retrieve the data from the file and assign to a variable.

In [ ]:
```
with open('test.p', 'rb') as fin:
    newData = pkl.load(fin)

print(newData[0:20:4])
```

```
[0.31189819 0.93054009 0.06209925 0.72638861 0.30226995]
```

In [ ]:
```
!ls -l .
```

```
total 2312
-rw-r--r--  1 vincentnguyen  staff    5108 Oct  8 19:08 2001.csv
-rw-r--r--  1 vincentnguyen  staff       8 Oct  8 19:08 README.md
-rw-r--r--@ 1 vincentnguyen  staff     853 Oct  8 19:08 airport.sql
-rw-r--r--@ 1 vincentnguyen  staff   16384 Oct  8 19:08 assignment.db
-rw-r--r--  1 vincentnguyen  staff   68450 Oct  8 19:08 collab_lab2_vincent-
nguyen-sys.ipynb
-rw-r--r--@ 1 vincentnguyen  staff  156427 Oct  8 19:31 collab_lab3-vincent-
nguyen-sys.ipynb
-rw-r--r--@ 1 vincentnguyen  staff      19 Oct  8 19:08 count_lines_flights.
sql
-rw-r--r--@ 1 vincentnguyen  staff     491 Oct  8 19:08 create.sql
-rw-r--r--@ 1 vincentnguyen  staff     360 Oct  8 19:08 delete.sql
-rw-r--r--@ 1 vincentnguyen  staff     262 Oct  8 19:08 delete2.sql
-rw-r--r--@ 1 vincentnguyen  staff       6 Oct  8 19:08 help.txt
-rw-r--r--  1 vincentnguyen  staff  850313 Oct  8 19:08 iata.csv
-rw-r--r--@ 1 vincentnguyen  staff    1354 Oct  8 19:08 insert.sql
-rw-r--r--@ 1 vincentnguyen  staff     496 Oct  8 19:08 orderby.sql
-rw-r--r--@ 1 vincentnguyen  staff     549 Oct  8 19:08 rename.sql
-rw-r--r--@ 1 vincentnguyen  staff     142 Oct  8 19:08 select.sql
-rw-r--r--@ 1 vincentnguyen  staff   16384 Oct  8 19:08 test
-rw-r--r--@ 1 vincentnguyen  staff     110 Oct 11 20:07 test.dat
-rw-r--r--@ 1 vincentnguyen  staff     950 Oct 11 20:08 test.p
-rw-r--r--@ 1 vincentnguyen  staff     310 Oct  8 19:08 update.sql
-rw-r--r--@ 1 vincentnguyen  staff     326 Oct  8 19:08 update2.sql
```

While easier than custom read/write routines, pickling still requires the file system to provide support for concurrency, consistency, and durability. To go any further with data persistence, we need to move beyond Python language constructs and employ additional software tools.

Database Systems

Whether you realize it or not, as you surf the Internet you're interacting with a variety of database-backed Web applications. This nomenclature may be unfamiliar, but it simply means that a website you visit is dynamically created using data saved in a database. To demonstrate, consider the following types of Web sites that you may visit:

- An information portal, like Google
- A newspaper Web site to catch up on the local news or sports
- A financial Web site, like that of a bank or investment institution, to monitor your financial portfolio
- A map website to find driving directions
- A search engine where you can identify interesting Web sites for more detailed information on a subject

Each of these examples use databases to store, locate, and retrieve information dynamically. In each of these applications, the website collects necessary information from the user (such as a street address), queries the application database, and collects the data that has been requested into a suitable visual result.

Many of these database systems are large and complex-imagine holding all the map information needed to provide accurate driving directions with pictures! Clearly, storing data and making it available to applications is a big task, one that has been addressed by a number of commercial vendors, that provide different solutions that are optimized for different tasks. Many of these open-source or commercial database systems provide full, enterprise-class capabilities. As a result, they can hold enormous quantities of data, concurrently interact with a large number of users, and scale across large computational systems.

We can broadly classify these systems into two categories:

1. Relational Database Management Systems like the open-source MySQL and PostgreSQL, and commercial systems like IBM DB2, Microsoft SQL Server, or Oracle Database that rely on a tabular data model.

2. NoSQL (or *Not only SQL*) systems that abondon the tabular data model to achieve a simpler design, better scaling or higher availability than is traditionally possible with relational databases. NoSQL databases can be classifid based on their data model, and include key-store databases like Amazon's Dynamo, Object Databases like ZopeDB, Document Store databases like MongoDB, and Column Databases like Cassandra or HBase, which is an open source implementation of Google's BigTable model.

While the NoSQL databases are extremely interesting, many of them have been developed to meet the **big data** challenges faced by companies like Google, Facebook, or Amazon. For the rest of this lab, we will focus on relational database systems.

Database Roles

As you might expect, working with these systems isn't trivial, and they can be expensive to operate. Historically, the tasks involved in working with these databases have been divided into three categories. Although the roles sometimes overlap, their individual responsibilities are easy to comprehend:

Database administrator (DBA): Responsible for the overall operation of the database system, which includes the selection and layout of the underlying hardware, the installation and optimization of the database server (especially given the hardware being used), and the day-to-day operations of the database server, such as data backup and recovery.

Database developer: Responsible for the actual databases in operation, including designing databases, schemas, tables, table relationships, and indexes as well as optimizing queries.

Database application developer: Responsible for integrating application code with the underlying database by using database application programming interfaces (APIs) to store and retrieve data as necessary.

If the previous discussion leaves you feeling intimidated, that's OK, working with databases has historically been difficult. To understand why, let's examine a specific example in more detail: online banking. When you connect to your bank's Web site, you provide your credentials (most likely a username and password) and thereby gain access to your financial accounts. You can view your data, pay bills, and transfer funds. The database your bank uses must quickly locate the relevant information, safely manage the transactions, securely interact with users, and *most important* not lose any data! And the bank must do this for a large number of users concurrently. To ensure these tasks are performed correctly, relational database systems are given a special test, known as the **ACID Test**.

The ACID Test

Diamonds are obviously a valuable commodity, so valuable that counterfeits are a serious concern. One simple and (at least, in the movies) popular test to determine whether a diamond is real is to run it across a piece of glass. Because diamonds are one of the hardest materials known, a real diamond easily cuts the glass surface; a fake, especially if it's made of glass itself, won't. Similarly, we might check whether a Persian rug is counterfit by trying to light it on fire (sorry Aunt Suzie, but at least you now know it's a fake!).

To a software developer, databases are equally valuable. If you use a database, you want to be sure it will safely store your data and let you easily retrieve the data later. You also want your database to allow multiple programs (or people) to work with the database without interfering with each other. To demonstrate, imagine you own a bank. The database for your bank must do the following, among other things:

- Safely store the appropriate data
- Quickly retrieve the appropriate data
- Support multiple, concurrent user sessions

These tasks can be collectively referred to as the ACID test; ACID is an acronym for Atomicity, Consistency, Isolation, and Durability.

Atomicity means that operations with the database can be grouped together and treated as a single unit.

Consistency guarantees that either all the operations in this single unit (or transaction) are performed successfully, or none of them is performed. In other words, a database can't be in an unfinished state. To understand why these characteristics are important, think about a bank transaction during which money is transferred from a savings account into a checking account. If the transfer process fails after subtracting the money from your savings account and before it was added to your checking account, you would become poorer, and the bank would have an angry (ex)customer! Atomicity enables the two operations -- the subtraction from the savings account and the addition to the checking account -- to be treated as a single transaction. Consistency guarantees that both operations in the transaction either succeed or fail. That way, your money isn't lost.

Isolation means that independent sets of database transactions are performed in such a way that they don't conflict with each other. Continuing the bank analogy, consider two customers who transfer funds between accounts at the same time. The database must track both transfers separately; otherwise, the funds could go into the wrong accounts, and the bank might be left with two angry (ex)customers.

Durability guarantees that the database is safe against unexpected terminations. It may be a minor inconvenience if your television or computer won't work when the power goes out, but the same can't be said for a database. If the bank's computers lose power when transferring your funds, you won't be a happy customer if the transaction is lost. Durability guarantees that if the database terminates abnormally during a funds transfer, then when the database is brought back up, it will be able to recover the transaction and continue with normal operations.

Passing the ACID test is nontrivial, and many simple databases fall short. For critical e-business or Web-based applications, passing the ACID test is a must. This is one of the reasons so many companies and individuals utilize enterprise-level database systems, such as IBM DB2, Oracle Database, or Microsoft SQL Server. These databases are fully compliant with the ACID test, and can meet many of the data persistence needs of large corporations or organizations. To do so, however, often requires a large team that includes database administrators, database developers, and database application developers to ensure that data is effectively persisted and available as necessary for business applications.

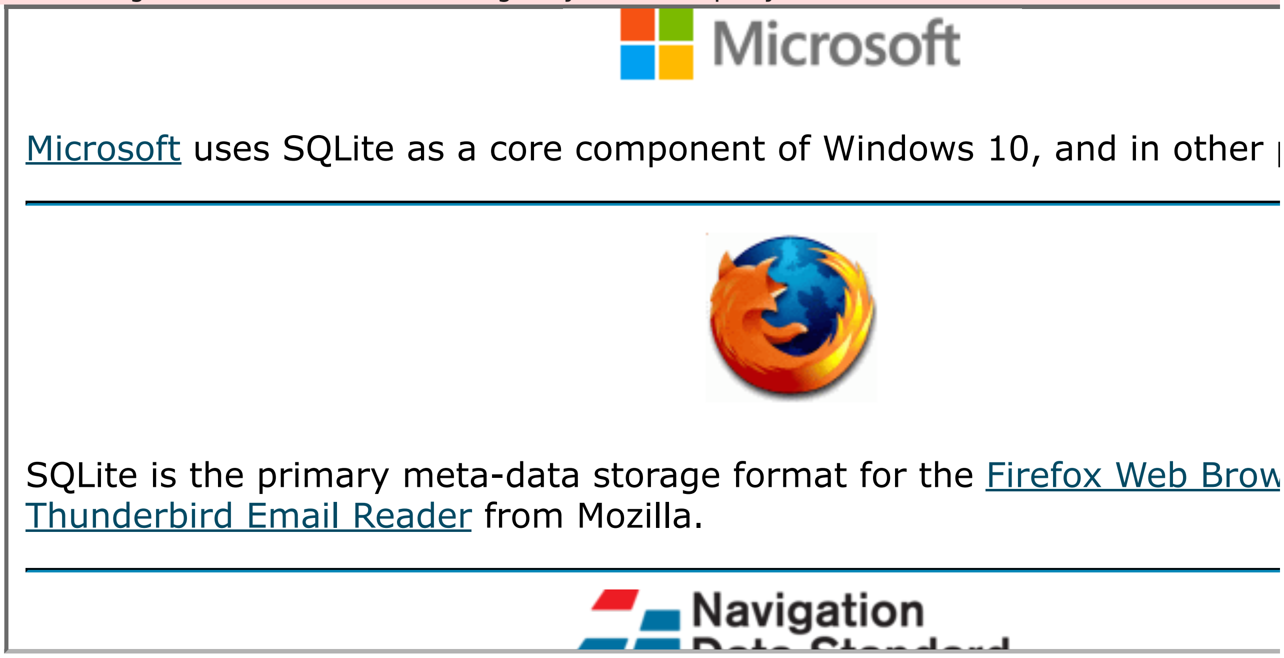
SQLite

But not all applications are this demanding, especially when you're starting out and trying to learn the basic relational database concepts. If you're just learning to work with databases, or if you want to quickly prototype a database application, most commercial database systems can be cumbersome. Fortunately, open-source, ACID compliant database systems exist, including the zero-configuration, serverless relational database system known as SQLite. By using SQLite, you can learn to work with a relational database by using SQL as well as the Python programming language. If you later find your application needs require a more powerful database system, you can always migrate your efforts to a more powerful database system; however, many systems continue to embed SQLite within their own applications, as demonstrated in the following webpage.

```
In [ ]: from IPython.display import HTML
        HTML('<iframe src=https://www.sqlite.org/famous.html width=750 height=300></
```

```
/opt/miniconda3/envs/swe-molecular-sciences/lib/python3.10/site-packages/IPy
thon/core/display.py:431: UserWarning: Consider using IPython.display.IFrame
instead
  warnings.warn("Consider using IPython.display.IFrame instead")
```

Out[ ]:



Microsoft uses SQLite as a core component of Windows 10, and in other

SQLite is the primary meta-data storage format for the Firefox Web Brow
Thunderbird Email Reader from Mozilla.

What is SQLite?

SQLite is quite different than traditional relational database systems. SQLite does not have a separate server process, instead SQLite is a software library that, as the website states:

> implements a self-contained, serverless, zero-configuration, transactional SQL database engine.

Before progressing, lets examine each of these concepts in turn:

- *self-contained*: Nothing else is needed to use SQLite but the software library. Since, by default, this comes with Python, we can use SQLite without any additional software downloads or installs. in addition, if you want to embed SQLite in your own application, you can obtain a single ANSI-C file that contains the entire SQLite library.
- *serverless*: We interact with the SQLite database by using the SQLite library. The database is stored in a single file that is platform independent (so you can simply

copy it over to a new machine with no further effort).

- *zero-configuration*: SQLite does not use a server process, so there is no configuration required. While you can customize sqlite to change default limits, for most applications this is unnecessary. You can also pre-specify certain options for the `sqlite3` command line client in a separate configuration file (e.g., `.sqliterc`, which is located in the current user's home directory).
- *transactional*: A transaction is a logical set of operations. SQLite is ACID-complaint by implementing atomic commits, which means that either every operation within the transaction completes successfully or none of them do. No partial writes are persisted, so that the database is always in a consistent state.

With this power, it is even more surprising that the SQLite library is quite small, and can be compacted to as small as **300 kb** if required.

SQLite by default will store data in a single database file; however, it can also be used as an *in memory* database. SQLite has been distributed as a component within the Python language for many years, but also has a stand-alone command line interface client, called `sqlite3` that we will use in this lesson use to create a database, create schema within that database, and to import data.

Additional References

1. SQLite Documentation
2. Free SQLite Tutorial

Introduction to SQL

We now focus on a basic component of relational database management systems, SQL. In this Notebook we will use the SQLite database to build a fictitious database. We will cover SQL data types and how to create SQL schemas before moving on to creating and executing queries and finishing with updating and deleting data.

We can now start using the SQLite database. First we will test SQLite from within the IPython Notebook, before switching to the command line to actually create and populate a database. Note that if we run the `sqlite3` command line client from within an IPython Notebook cell, the process will continue to run *in the background* since we can not directly enter commands. Thus, you should either redirect *STDIN* for the `sqlite3` client to be a file of commands, or work directly with this tool at the command line.

Working with SQLite

By default, the `sqlite3` command line client will operate in interactive mode. However, this tool will also read and execute commands either in from a separate file by redirecting *STDIN* or by enclosing the commands in quotes. Since SQLite databases are files, unless explicitly created from within a program as in memory databases, we pass the name of the database as a command line argument. Thus, to connect to a database with the `sqlite3` command line client in interactive mode, we simply enter the following at a command prompt in our Docker container:

```
/home/database: $ sqlite3 <name of database>
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

To exit from the `sqlite3` client, simply enter either ctrl-d or use the `.quit` command. The `sqlite3` client can either accept SQL commands, which we will discus in more detail in the next lesson, or the client can accept dot commands, which are instructions to the SQLite database engine that begin with a `.` character. These commands can be explicitly listed by entering `.help` at the `sqlite3` client prompt. We can do this from within our Notebook by creating and using a file as shown below.

In [ ]:
```
!apt-get install sqlite3
!brew install sqlite3
```

```
/bin/bash: apt-get: command not found
==> Auto-updating Homebrew...
Adjust how often this is run with HOMEBREW_AUTO_UPDATE_SECS or disable with
HOMEBREW_NO_AUTO_UPDATE. Hide these hints with HOMEBREW_NO_ENV_HINTS (see `m
an brew`).
==> Auto-updated Homebrew!
Updated 2 taps (homebrew/core and homebrew/cask).
==> New Formulae
afl++               icu4c@75            m68k-elf-binutils   python-tk@3.13
argtable3           jikken              m68k-elf-gcc        python@3.13
blisp               jxl-oxide           openapi-diff        setconf
crossplane          libcss              oxlint              sysprof
djlint              libdom              paperjam            tllist
dug                 libhubbub           pipet               tmux-sessionizer
gptme               libparserutils      python-gdbm@3.13
==> New Casks
ableset                                  font-matemasie
anytype@alpha                            font-moderustic
anytype@beta                             font-new-amsterdam
backyard-ai                              font-noto-serif-todhri
bentobox                                 font-sour-gummy
cap                                      font-suse
colemak-dh                               furtherance
colemak-dhk                              homerow
displaybuddy                             imaging-edge
excalidrawz                              keyguard
font-bungee-tint                         magicquit
font-edu-au-vic-wa-nt-arrows             microsoft-edge@canary
font-edu-au-vic-wa-nt-dots               pixel-shift-combiner
font-edu-au-vic-wa-nt-guides             quba
font-edu-au-vic-wa-nt-pre                sanctum
font-funnel-display                      thunderbird@esr
font-funnel-sans                         typefully

You have 8 outdated formulae installed.

Warning: sqlite 3.46.1 is already installed and up-to-date.
To reinstall 3.46.1, run:
  brew reinstall sqlite
```

In [ ]:
```
# Run the SQLite command line client

!sqlite3 -help
```

```
Usage: sqlite3 [OPTIONS] [FILENAME [SQL]]
FILENAME is the name of an SQLite database. A new database is created
if the file does not previously exist. Defaults to :memory:.
OPTIONS include:
   --                   treat no subsequent arguments as options
   -A ARGS...           run ".archive ARGS" and exit
   -append              append the database to the end of the file
   -ascii               set output mode to 'ascii'
   -bail                stop after hitting an error
   -batch               force batch I/O
   -box                 set output mode to 'box'
   -column              set output mode to 'column'
   -cmd COMMAND         run "COMMAND" before reading stdin
   -csv                 set output mode to 'csv'
   -deserialize         open the database using sqlite3_deserialize()
   -echo                print inputs before execution
   -init FILENAME       read/process named file
   -[no]header          turn headers on or off
   -help                show this message
   -html                set output mode to HTML
   -interactive         force interactive I/O
   -json                set output mode to 'json'
   -line                set output mode to 'line'
   -list                set output mode to 'list'
   -lookaside SIZE N    use N entries of SZ bytes for lookaside memory
   -markdown            set output mode to 'markdown'
   -maxsize N           maximum size for a --deserialize database
   -memtrace            trace all memory allocations and deallocations
   -mmap N              default mmap size set to N
   -newline SEP         set output row separator. Default: '\n'
   -nofollow            refuse to open symbolic links to database files
   -nonce STRING        set the safe-mode escape nonce
   -no-rowid-in-view    Disable rowid-in-view using sqlite3_config()
   -nullvalue TEXT      set text string for NULL values. Default ''
   -pagecache SIZE N    use N slots of SZ bytes each for page cache memory
   -pcachetrace         trace all page cache operations
   -quote               set output mode to 'quote'
   -readonly            open the database read-only
   -safe                enable safe-mode
   -separator SEP       set output column separator. Default: '|'
   -stats               print memory stats before each finalize
   -table               set output mode to 'table'
   -tabs                set output mode to 'tabs'
   -unsafe-testing      allow unsafe commands and modes for testing
   -version             show SQLite version
   -vfs NAME            use NAME as the default VFS
   -zip                 open the file as a ZIP Archive
```

In [ ]:
```
%%writefile help.txt
.help
```

```
Writing help.txt
```

In [ ]:
```
# List 'dot' commands

!sqlite3 < help.txt
```

```
.archive ...             Manage SQL archives
.auth ON|OFF             Show authorizer callbacks
.backup ?DB? FILE        Backup DB (default "main") to FILE
.bail on|off             Stop after hitting an error.  Default OFF
.cd DIRECTORY            Change the working directory to DIRECTORY
.changes on|off          Show number of rows changed by SQL
.check GLOB              Fail if output since .testcase does not match
.clone NEWDB             Clone data into NEWDB from the existing database
.connection [close] [#]  Open or close an auxiliary database connection
.databases               List names and files of attached databases
.dbconfig ?op? ?val?     List or change sqlite3_db_config() options
.dbinfo ?DB?             Show status information about the database
.dump ?OBJECTS?          Render database content as SQL
.echo on|off             Turn command echo on or off
.eqp on|off|full|...     Enable or disable automatic EXPLAIN QUERY PLAN
.excel                   Display the output of next command in spreadsheet
.exit ?CODE?             Exit this program with return-code CODE
.expert                  EXPERIMENTAL. Suggest indexes for queries
.explain ?on|off|auto?   Change the EXPLAIN formatting mode.  Default: auto
.filectrl CMD ...        Run various sqlite3_file_control() operations
.fullschema ?--indent?   Show schema and the content of sqlite_stat tables
.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
.indexes ?TABLE?         Show names of indexes
.limit ?LIMIT? ?VAL?     Display or change the value of an SQLITE_LIMIT
.lint OPTIONS            Report potential schema issues.
.load FILE ?ENTRY?       Load an extension library
.log FILE|on|off         Turn logging on or off.  FILE can be stderr/stdout
.mode MODE ?OPTIONS?     Set output mode
.nonce STRING            Suspend safe mode for one command if nonce matches
.nullvalue STRING        Use STRING in place of NULL values
.once ?OPTIONS? ?FILE?   Output for the next SQL command only to FILE
.open ?OPTIONS? ?FILE?   Close existing database and reopen FILE
.output ?FILE?           Send output to FILE or stdout if FILE is omitted
.parameter CMD ...       Manage SQL parameter bindings
.print STRING...         Print literal STRING
.progress N              Invoke progress handler after every N opcodes
.prompt MAIN CONTINUE    Replace the standard prompts
.quit                    Stop interpreting input stream, exit if primary.
.read FILE               Read input from FILE or command output
.recover                 Recover as much data as possible from corrupt db.
.restore ?DB? FILE       Restore content of DB (default "main") from FILE
.save ?OPTIONS? FILE     Write database to FILE (an alias for .backup ...)
```

```
.scanstats on|off|est    Turn sqlite3_stmt_scanstatus() metrics on or off
.schema ?PATTERN?        Show the CREATE statements matching PATTERN
.separator COL ?ROW?     Change the column and row separators
.sha3sum ...             Compute a SHA3 hash of database content
.shell CMD ARGS...       Run CMD ARGS... in a system shell
.show                    Show the current values for various settings
.stats ?ARG?             Show stats or turn stats on or off
.system CMD ARGS...      Run CMD ARGS... in a system shell
.tables ?TABLE?          List names of tables matching LIKE pattern TABLE
.timeout MS              Try opening locked tables for MS milliseconds
.timer on|off            Turn SQL timer on or off
.trace ?OPTIONS?         Output each SQL statement as it is run
.version                 Show source, library and compiler versions
.vfsinfo ?AUX?           Information about the top-level VFS
.vfslist                 List all available VFSes
.vfsname ?AUX?           Print the name of the VFS stack
.width NUM1 NUM2 ...     Set minimum column widths for columnar output
```

SQLite Cliet Tool Options

The previous code block listed the available dot commands that we can use within the `sqlite3` client. We will use several of these, including

- `.header`: If this is `on`, headers will be displayed. Generally this is used to see the names of columns in a table.
- `.separator`: This specifies the separator character for fields, either in output displays or when importing data. By default, the separator character is the vertical bar, `|`. However, to import data from a comma-separated-value (CSV) file, you would change the separator to a comma, `.separator ","`.
- `.import`: This command is used to read data from a file and insert this data into a specific table in the database.
- `.schema`: This command will list the schema commands required to recreate the tables contained in the database.
- `.stats`: This command lists statistics after each command entered at the sqlite prompt. This can be useful for profiling SQL commands.
- `.width`: This command changes the default width for columns, which can improve the visual formatting of the results from database queries that are displayed to the screen.

Creating and Populating a Database

We can easily create and populate a database by using the `sqlite3` client. While we could do this at the command line (and advanced users are encouraged to do so), we can also complete these tasks from within this IPython Notebook. The steps we must complete include

1. Create the new database. We do this by simply passing the name of our new database to the `sqlite3` client. If the file does not exist, a new file will be created. This file will hold the entire contents of the new database.

2. Create the schema for our new database. A relational database is built on a tabular data model. Thus our schema consists of the table definitions as well as the relationships that might exist between tables. To accomplish this, we must execute SQL `CREATE TABLE` statements. For now, we simply create the schema, the next lesson explores SQL statements in more detail. A schema file for the airline data is included in this lesson, under the `schema` directory.

3. Populate the tables with data. For simplicity, we will use the `.import` command within the `sqlite3` client to import data from a file directly into the relevant table in our database.

In [ ]:
```
!pwd
```
```
/Users/vincentnguyen/chem_274b/labs/MSSE-1
```

In [ ]:
```
# First we make a new directory to hold our database

!mkdir /content/database
```

In [ ]:
```
!ls -l /content/database
```
```
ls: /../database: No such file or directory
```

In [ ]:
```
# We could create a schema SQL file, then use Bulk imort to load the databas
```

The basics of relational database systems

Before you can begin to develop database applications, you need to understand the basic concepts. Relational databases hold data. This data can be of different types, such as numbers, characters, or dates. Within the database, the data are organized into logical units called tables. A table is like a spreadsheet, because it contains rows of data. Each row is made up of a number of columns. The columns hold data of a specific data type, like integer values or strings of characters. In most cases, a database has more than one table. To relate the tables together, a database designer takes advantage of natural (or artificial) links between the tables. In a spreadsheet, you can link rows in different sheets by cell values. The same idea holds in a relational database, and the column used to make the link is referred to as a key column.

To make it easier to understand the purpose of a table or a particular column, you should select appropriate names. The naming conventions can vary between databases. For a SQLite database, individual names:

- by default, case insensitive (although this can be changed),
- are unlimited in length, but should be kept to a reasonable length (given readability constraints),
- must begin with a letter or an underscore character, and
- must only code alphanumeric characters or underscores.

You can escape these rules by placing the name in double quotation marks, which allows names to be case sensitive and to include additional characters (including spaces). Doing this, however, is generally a bad practice: It requires the name to always be enclosed in double quotation marks and can easily confuse someone else who may be maintaining your code. Finally, a name can not be one of the reserved keywords.

Class Style

For this lab, we will follow a specific style: All SQL commands are presented entirely in uppercase, and item names use camelCase. In camelCase style, words are joined together, and the first letter of each word-following the first one-is capitalized, such as aLongIdentifier. Combining these two styles together, these articles write SQL commands using the following style: SELECT aLongIdentifier FROM dataTable ;.

Schema

Related tables are often grouped together into a schema. You can think of a schema as a container for all the related structure definitions within a particular database. A table name must be unique within a given schema. Thus, by using schemas, you can have identically named objects (such as tables) enclosed within different schemas.

You can use the schema name to qualify a name. By default, SQLite uses the database name as the schema, and you do not need to prefix names to indicate the correct schema. For other databases, however, this is not the case. For these databases, you specify the schema name followed by a period and then the table name. For example, bigdog.products references the products table in the bigdog schema. Without the relevant schema name, a table name is said to be unqualified, as in products. When the schema name and the table name are completely specified, as in bigdog.products, the name is said to be fully qualified.

In an abstract sense, these database concepts may seem confusing, but in practice they're fairly straightforward. For example, imagine you own a store called Bigdog's Surf Shop that sells a variety of items like sunglasses, shirts, and so on. If you want to be profitable, you must keep a close eye on your inventory so you can easily order additional inventory or change vendors to keep your overhead to a minimum. One simple method for tracking this information is to write entries in a table-like format:

Product Table

Item#	Price	Stock Date	Description
1	29.95	1/15/15	Basic Sunglasses
2	9.95	12/14/14	Generic Shirt
3	99.95	8/04/14	Boogie Board

Vendors Table

Item#	Vendor#	Vendor Name
1	101	Mikal Arroyo
2	102	Quiet Beach Industries
3	103	Vista Luna

From this simple visual design you can easily map the business logic straight into database tables. You have two database tables, Products and Vendors, which are naturally linked by the item number. The data types for the columns in each table are easy to determine. Later in this lesson we will actually create this sample schema for Bigdog's Surf Shop, which consists of these two tables, in a SQLite database. But first, we need to address how data is stored in a relational database table.

SQL: Structured Query Language

Database systems can be complex pieces of software, especially when they scale to support enterprise-level applications. As a result, you may expect that every database has its own application programming interface (API) and that these APIs may be different from one system to the next. When relational databases were first developed, this was the case; but, fortunately, a number of vendors agreed to develop a standard language for accessing and manipulating relational databases. This language is officially called Structured Query Language (or SQL, pronounced sea-quill). Several official standard versions have been produced, including one in 1992 that is referred to as SQL-92, and one in 1999 that is referred to as SQL-99. The Apache Derby database provides a nearly complete implementation of the SQL-92 standard, so applications developed with Derby can be easily transported to other database systems.

SQL has two main components: a Data Definition Language (DDL) and a Data Manipulation Language (DML). DDL commands are used to create, modify, or delete items (such as tables) in a database. DML commands are used to add, modify, delete, or select data from a table in the database. The rest of this article provides a basic introduction to the DDL components of SQL. Future articles will focus on the DML commands and more advanced DDL commands.

SQL data types

SQL, being a programming language in its own right, defines a rich data-type hierarchy. Persisting these data types is one of the most important responsibilities of the database. As databases have become more powerful, this type hierarchy has grown more complex. But most simple databases don't require the full range of allowed types, and often they need to store only numerical, character, and date or time data.

While the SQL standard defines basic data types, different database systems can support the standard to varying degrees. While this might seem odd, doing so provides more flexibility in allowing a particular implementation to achieve a market niche. In the case of SQLite, the design decisions support a compact, zero-configuration database file that is platform-independent. As a result, SQLite does not support a rich data type hierarchy, and instead focuses on ease-of-use.

SQLite supports five storage classes:

- **NULL**: A null value.
- **INTEGER**: A signed integer, the number of bytes (1, 2, 3, 4, 6, or

8. used depends on the magnitude of the value.

- **REAL**: A floating-point value stored as an 8 byte IEEE floating-point value.
- **TEXT**: A string of character values stored in the default database encoding (e.g.,

UTF-8).

- **BLOB**: A blob of data stored *exactly* as is in the database.

Note that SQLite does not support Boolean or Date/Time valus directly. Instead, Boolean values are encoded as INTEGERs (0 = False, 1 = True). Likewise Date/Time values can be encoded either as TEXT, REAL, or INTEGER values. For full details, see the SQLite documentation. In addition, SQLite supports the concept of *Type Affinity*, whereby different data types can be easily mapped into each other. This simplifies moving schemas from other database systems to SQLite.

The SQL NULL type

Before you begin creating database tables, you must know what to do when no value is specified for a column. To illustrate this point, imagine that you've been asked to fill out a Web form. If you leave a particular column blank, what is inserted into the database? As you can imagine, this problem could be cumbersome if you had to track no value markers. Fortunately, SQL defines a special value, NULL, to indicate that a column has no value.

CREATE TABLE

So far, you've learned how to design a table, including mapping out the table columns and defining the data type for each column. After you've properly designed a table, the method for creating a table in SQL is straightforward. Listing 1 shows the formal syntax for creating a table in Derby.

```
-- Comment describing the purpose and layout of the table

CREATE TABLE tableName ( { <columnDefinition> |
<tableLevelConstraint> }
    [, { <columnDefinition> | <tableLevelConstraint> } ]* ) ;
```

You may feel bewildered after looking at this syntax for the first time. But it's easy to follow once you have the basics down. The square brackets ([and]) enclose optional parameters. As you can see from the formal syntax, any column definitions or table-level constraints after the required initial one (it wouldn't make sense to create a table with no columns!) are optional.

You probably understand what is meant by a column definition, but you might not understand the idea of a constraint. Constraints come in two types: table-level constraints and column constraints. A constraint limits either a column or a table in some manner. For example, you can use a constraint to require that a column always be assigned an actual value (no NULL values), or that every entry in a column must be unique, or that a column is automatically assigned a default value.

The asterisk (*) after the last closing square bracket indicates that more than one of the enclosing items can be included. This implies that the table must have one or more columns or table-level constraints. The vertical line (|) indicates an either/or condition. In this syntax example, you must either define a new column or define a new table-level constraint. The curly brackets ({ and }) group related items together, and the parentheses ((and)) are required elements. Finally, the semicolon (;) indicates the end of a SQL statement.

SQLite Schema Creation

In the following code block, we create our schema for Bigdog's Surf Shop, which includes two new tables: `myProducts` and `myVendors`. The myProducts table has four columns: itemNumber, price, stockDate, and description. The itemNumber column provides a unique identity for each item (or row) and has an attached column-level constraint that enforces a valid value to always be supplied (NOT NULL). Without this requirement, the itemNumber column isn't guaranteed to be unique because multiple columns could be assigned a NULL value. The price column is created as a REAL data type. The last two columns are simple: The stockDate column is stored as a TEXT (we can use application logic to transform the date/time information into and out of the appropriate YYYY-MM-DD format), and description is also stored in a TEXT field.

The myVendors table has three columns: itemNumber, vendorNumber, and vendorName. In this case, both the itemNumber and vendorNumber columns have attached column-level constraints (NOT NULL). In addition, the vendorName column is stored as a TEXT field.

Drop Table

No one is perfect. What do you do when you incorrectly create a table or a table is no longer needed? The simple answer is to delete the table from the database and, if necessary, create a replacement table. Deleting a table is easy, which means, of course, that you should exercise great care when doing so -- no dialog box pops up and asks if you're sure you want to proceed!

The full syntax for deleting-or, more formally, dropping-a table from a database in SQLite is

DROP TABLE tableName ;

The syntax is simple: You append the fully qualified name and a semicolon to the DROP TABLE SQL command, and you're finished.

In the next few code cells, we first create our schema file, before executing the SQL commands on our new database by using the `sqlite3` client tool.

In [ ]:
```
%%writefile create.sql

-- First we drop any tables if they exist
-- Ignore the no such Table error if present

DROP TABLE myVendors ;
DROP TABLE myProducts ;

-- Vendor Table: Could contain full vendor contact information.

CREATE TABLE myVendors (
    itemNumber INT NOT NULL,
    vendornumber INT NOT NULL,
    vendorName TEXT
) ;

-- Product Table: Could include additional data like quantity

CREATE TABLE myProducts (
    itemNumber INT NOT NULL,
    price REAL,
    stockDate TEXT,
    description TEXT
) ;
```

```
Writing create.sql
```

In [ ]:
```
# Now create the schema in a new test database
# The following two error lines are fine to ignore
# Error: near line 5: in prepare, no such table: myVendors (1)
# Error: near line 6: in prepare, no such table: myProducts (1)

!sqlite3 test < create.sql
```

```
Parse error near line 5: no such table: myVendors
Parse error near line 6: no such table: myProducts
```

In [ ]:
```
# We can test the results
```

```
!sqlite3 test ".schema"
```

```
CREATE TABLE myVendors (
    itemNumber INT NOT NULL,
    vendornumber INT NOT NULL,
    vendorName TEXT
);
CREATE TABLE myProducts (
    itemNumber INT NOT NULL,
    price REAL,
    stockDate TEXT,
    description TEXT
);
```

SQL scripts

As the previous code blocks demonstrated, we can write SQL commands into a script file that can be easily executed by the `sqlite3` client. While this might seem like overkill given the simplicity of our current schema, it is actually a useful technique. Often, you'll need to execute multiple, complex commands. To simplify debugging a set of complex SQL commands, it's generally easier to write them in a text file and then execute the commands in the text file all at once. By placing SQL commands in a script file, you gain the additional benefit of being able to execute the commands as many times as necessary.

A script file is just a plain text file that contains a combination of SQL commands and SQLite commands that can be run directly from the `sqlite3` tool. A script file simplifies the development and maintenance of relational databases and provides a self-documenting technique for building databases. You should store these files as ASCII text files, not as RTF files (or any other format), to prevent text-encoding errors. Some text applications may try to automatically save your file as a rich text file. Be careful to avoid doing so, or you may have problems when you try to execute your script file. For this reason, you probably will want to always create and edit your SQL scripts at the Unix command prompt.

Our previous script file, `create.sql`, includes several lines that start with two dashes (--). These lines are SQL comments; you should use them to provide a basic description of the purpose of each major component within the script file. The first actual commands ion the script file are SQL DROP statements that delete the myProducts and myVendors tables from the database. If the tables don't exist (which is the case if the database was just created), an error message is displayed; but as the preceding SQL comments indicate, you can safely ignore those messages.

You first drop the tables, if they exist, so that you can cleanly create new tables with the exact column definitions you need. The next two SQL statements do just that, creating the myProducts and myVendors tables. To run this script file, we can either have the `sqlite3` client tool read the file form *STDIN*, as shown above, or we can start `sqlite3` and use the `.read` command to read and execute SQL commands from our `create.sql` file.

What if something goes wrong?

Sometimes, no matter how hard you try, things don't work out quite right. If you can't safely execute the create.sql script, there are a number of possibilities to check:

- Be sure the SQLite client tool starts up properly. You can do this at the Unix command prompt.
- Be sure you have free disk space in which to create a new database.
- Be sure you have proper permissions (to read the script file and to create the new database) in the directory where you try to execute the script file.
- Be sure your script file is a simple ASCII text file (and not an RTF file).

If the output of the `!sqlite3 test ".schema"` matches the data in the `create.sql` file, congratulations are in order. You now have a new test database with two new tables ready to hold data.

Now we'll focus on the process of inserting data into a table by using SQLite. To follow along, you will need a SQLite database with the myProducts table available. If you haven't already done so, you should execute the create.sql script file.

Additional References

Several sites exist that allow you to try out SQL commands online.

1. W3 Schools SQL, a general SQL demo site.
2. SQLZoo, allows you to specify the Relational Database to target.

Introduction to SQL Data Manipulation Language

From here, we focus on the basic task of manipulating data in relational database management systems by using SQL DML. In this Notebook we will expand on our use of the SQLite database to build and query a fictitious database. We will cover inserting data into tables, creating and executing queries, and finishing with updating and deleting data.

INSERT

One of the most important tasks when you're building a database application is inserting data into the database. It doesn't matter how good the database software is-if you put bad data in a database, nothing else matters. There are several different ways to insert data into a database, but the rest of this lesson focuses on inserting data into a SQLite database by using the SQL INSERT statement.

Before you can insert data into a SQLite database using the SQL INSERT statement, you must know how to properly use this statement. The full syntax for the SQL INSERT statement is

```
INSERT INTO table-Name
    [ (Simple-column-Name [ , Simple-column-Name]* ) ]
         Expression
```

which should seem familiar. As discussed previously, the square brackets ([]) enclose optional parameters. The only component whose purpose isn't immediately clear is *Expression*; but how complex can that simple phrase be? Of course, appearances can be deceiving; the Expression term can expand to one of four different structures:

- a single-row VALUES list
- a multiple-row VALUES list
- a SELECT expression
- a UNION expression

Of these, the last two are beyond the scope of this lesson. The first two are similar; the only difference is that the first form inserts one row into a table, whereas the latter form inserts multiple rows into a table.

You can use the optional part of the SQL INSERT statement to specify the column order of the values being inserted into the table. By default, data is inserted into a table's columns in the same order that the columns were listed when the table was created. Sometimes you may want to change this order or perhaps only specify values for columns that have NOT NULL constraints. By explicitly listing the columns in your SQL INSERT statement, you gain more control of the operation and can more easily handle these specific use cases.

The syntax for the SQL VALUES expression is fairly simple,

```
{
    VALUES ( Value {, Value }* )
        [ , ( Value {, Value }* ) ]* |
    VALUES Value [ , Value ]*
 }
```

This syntax displays the multiple-row format first, followed by the single-row format (remember that the vertical line character, |, means or and that the asterisk character, *, means one or more). The value term stands for a value that you want to insert into a specific column. To insert data into multiple columns, you must enclose the data for a row in parentheses separated by commas.

As shown below, to insert data into a table, you first need to to make sure that the table exists. If you haven't already done so, execute the table creation scripts discussed earlier.

```
INSERT INTO myProducts
    VALUES(1, 19.95, '2015-03-31', 'Hooded sweatshirt') ;

INSERT INTO myProducts(itemNumber, price, stockDate,
description)
    VALUES(2, 99.99, '2015-03-29', 'Beach umbrella') ;

INSERT INTO myProducts(itemNumber, price, stockDate)
    VALUES(3, 0.99, '2015-02-28') ;
```

This example presents three single-row inserts into the myProducts table. The first SQL INSERT statement doesn't provide a list of columns; it inserts an itemNumber, a price, a stockDate, and a description. Notice that the values inserted into both the stockDate and description columns are enclosed in single quote characters. The description column is a TEXT field, so it expects a string (which you indicate by enclosing the character data within single quotes). The stockDate column is also a TEXT field; as part of our application logic, we could pass in dates in the correct day, month, and year format. (For more guidance on the format of data types during a SQL INSERT operation, read the SQLite documentation).

The second SQL INSERT statement explicitly lists all four columns and inserts new values appropriately. The final SQL INSERT statement lists only three columns and inserts only three values. The description column is left empty, which means it will have a NULL value.

Although single-row SQL INSERT statements can be useful, when you need to insert multiple rows, it's more efficient to do so directly, as shown below:

```
INSERT INTO myProducts(itemNumber, price, stockDate,
description)
VALUES (4, 29.95, '2015-02-10', 'Male bathing suit, blue'),
       (5, 49.95, '2015-02-20', 'Female bathing suit, one
piece, aqua'),
       (6, 9.95, '2015-01-15', 'Child sand toy set'),
       (7, 24.95, '2014-12-20', 'White beach towel'),
       (8, 32.95, '2014-12-22', 'Blue-striped beach towel'),
       (9, 12.95, '2015-03-12', 'Flip-flop'),
       (10, 34.95, '2015-01-24', 'Open-toed sandal') ;
```

In this example, we insert seven rows into the database by explicitly listing all four columns and providing new values for each row. As discussed earlier, multiple-row inserts enclose the values for each new row within parentheses, and these values are separated by commas.

To actually execute these statements, we can place the SQL INSERT statements in a script file and run the script to insert the data. This approach lets you more easily fix errors or reinsert the data if necessary without recreating the requisite SQL INSERT statements. In the following code cells, we first create a SQL INSERT script file, before executing this script by using the `sqlite3` client tool. After this, we use the `.dump` command in the `sqlite3` client tool to display the full schema and contents of this SQLite database.

In [ ]:
```
%%writefile insert.sql

-- Single unnamed INSERT

INSERT INTO myProducts
VALUES(1, 19.95, '2015-03-31', 'Hooded sweatshirt') ;

-- Single named INSERT

INSERT INTO myProducts (itemNumber, price, stockDate, description)
VALUES(2, 99.99, '2015-03-29', 'Beach umbrella') ;

-- Single named INSERT with missing data

INSERT INTO myProducts (itemNumber, price, stockDate)
VALUES(3, 0.99, '2015-02-28') ;

-- Multiple named INSERT
```

```
INSERT INTO myProducts (itemNumber, price, stockDate, description)
VALUES (4, 29.95, '2015-02-10', 'Male bathing suit, blue'),
       (5, 49.95, '2015-02-20', 'Female bathing suit, one piece, aqua'),
       (6, 9.95, '2015-01-15', 'Child sand toy set'),
       (7, 24.95, '2014-12-20', 'White beach towel'),
       (8, 32.95, '2014-12-22', 'Blue-striped beach towel'),
       (9, 12.95, '2015-03-12', 'Flip-flop'),
       (10, 34.95, '2015-01-24', 'Open-toed sandal') ;

-- Insert into myVendors

INSERT INTO myVendors(itemNumber, vendorNumber, vendorName)
VALUES (1, 1, 'Luna Vista Limited'),
       (2, 1, 'Luna Vista Limited'),
       (3, 1, 'Luna Vista Limited'),
       (4, 2, 'Mikal Arroyo Incorporated'),
       (5, 2, 'Mikal Arroyo Incorporated'),
       (6, 1, 'Luna Vista Limited'),
       (7, 1, 'Luna Vista Limited'),
       (8, 1, 'Luna Vista Limited'),
       (9, 3, 'Quiet Beach Industries'),
       (10, 3, 'Quiet Beach Industries') ;
```

```
Writing insert.sql
```

In [ ]:

```
!head -10 insert.sql
```

```
-- Single unnamed INSERT

INSERT INTO myProducts
VALUES(1, 19.95, '2015-03-31', 'Hooded sweatshirt') ;

-- Single named INSERT

INSERT INTO myProducts (itemNumber, price, stockDate, description)
VALUES(2, 99.99, '2015-03-29', 'Beach umbrella') ;
```

In [ ]:

```
!sqlite3 test < insert.sql
```

In [ ]:

```
!sqlite3 test ".dump"
```

```
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE myVendors (
    itemNumber INT NOT NULL,
    vendornumber INT NOT NULL,
    vendorName TEXT
);
INSERT INTO myVendors VALUES(1,1,'Luna Vista Limited');
INSERT INTO myVendors VALUES(2,1,'Luna Vista Limited');
INSERT INTO myVendors VALUES(3,1,'Luna Vista Limited');
INSERT INTO myVendors VALUES(4,2,'Mikal Arroyo Incorporated');
INSERT INTO myVendors VALUES(5,2,'Mikal Arroyo Incorporated');
INSERT INTO myVendors VALUES(6,1,'Luna Vista Limited');
INSERT INTO myVendors VALUES(7,1,'Luna Vista Limited');
INSERT INTO myVendors VALUES(8,1,'Luna Vista Limited');
INSERT INTO myVendors VALUES(9,3,'Quiet Beach Industries');
INSERT INTO myVendors VALUES(10,3,'Quiet Beach Industries');
CREATE TABLE myProducts (
    itemNumber INT NOT NULL,
    price REAL,
    stockDate TEXT,
    description TEXT
);
INSERT INTO myProducts VALUES(1,19.94999999999999929,'2015-03-31','Hooded sw
eatshirt');
INSERT INTO myProducts VALUES(2,99.98999999999999949,'2015-03-29','Beach umbr
ella');
INSERT INTO myProducts VALUES(3,0.98999999999999992,'2015-02-28',NULL);
INSERT INTO myProducts VALUES(4,29.94999999999999929,'2015-02-10','Male bath
ing suit, blue');
INSERT INTO myProducts VALUES(5,49.95000000000000284,'2015-02-20','Female ba
thing suit, one piece, aqua');
INSERT INTO myProducts VALUES(6,9.94999999999999928,'2015-01-15','Child sand
toy set');
INSERT INTO myProducts VALUES(7,24.94999999999999929,'2014-12-20','White bea
ch towel');
INSERT INTO myProducts VALUES(8,32.95000000000000285,'2014-12-22','Blue-stri
ped beach towel');
INSERT INTO myProducts VALUES(9,12.94999999999999929,'2015-03-12','Flip-flo
p');
INSERT INTO myProducts VALUES(10,34.95000000000000285,'2015-01-24','Open-toe
d sandal');
COMMIT;
```

Transactions

If you look carefully at the output of the `.dump` command, you see that near the top of the output is a `BEGIN TRANSACTION;` statement and at the end of the output is a `COMMIT;` statement. These two statements are explicit instructions, inserted by

SQLite, to start a transaction, which is a logical unit of work, and save all operations in the transaction to the database. If a set of operations is not completed successfully, the transaction model requires that the commit does not occur, and instead a rollback is issued to return the database to the state that existed prior to the transaction commencing.

SELECT

In the SQL programming language, the task of performing a query falls to the SELECT statement. To provide all the query functionality required by database applications, the SELECT statement's capabilities are extensive. Before looking at example SELECT statements, lets first look at the formal syntax of SELECT, which, as shown below is actually simple. The basic format is `SELECT ... FROM ... WHERE;`, you select the columns of interest from rows in a table or tables where certain conditions are satisfied. Of course, things can become considerably more complex. This article covers the basic features of SELECT and defers the more advanced issues to subsequent articles.

```
SELECT [ DISTINCT | ALL ] SelectItem [ , SelectItem ]*
FROM clause
[ WHERE clause ]
[ GROUP BY clause ]
[ HAVING clause ]
```

From this you can see that a basic SELECT statement requires only a SELECT and a FROM; you must specify what data to select and indicate the location of the data of interest. Everything else is optional (as indicated by the square brackets). The DISTINCT and ALL keywords are optional qualifiers to indicate that either rows with unique values or all rows should be selected, respectively. By default, ALL is implicitly assumed, and you can use only one DISTINCT qualifier per SELECT statement.

A SELECT statement can have multiple columns listed following the SELECT keyword. Multiple elements (or, more generally, column names) are separated by commas. For example, `SELECT a, b, c` selects the three columns a, b, and c. To select all columns from a table, you can use the asterisk character (*) as a shorthand for all columns. An important point to remember is that the result of any SELECT statement is a transient SQLite table, and you can use it in many of the same ways you use a more permanent table.

The FROM component of a SELECT statement indicates from which table (or multiple tables) the data will be extracted. For now, we will focus on selecting data from a single table; latter we will cover table joins and selecting data from multiple tables. In this case, the fully qualified name of the table to query must follow the FROM keyword.

The rest of the SELECT statement is optional. Before you build your first query, however, lets review the order in which the SELECT statement components are evaluated:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause

When you break down the process SQLite follows when processing a query, this order is intuitive. First you must locate the data to be analyzed, after which you filter out the rows of interest. The next steps are to group related rows and, finally, to select the actual columns of interest.

SQLite

To demonstrate a SELECT statement, we can extract all the columns from the myProducts table by using the `sqlite3` tool and passing the SQL statement in as a command line argument.

```
In [ ]: !sqlite3 test "SELECT * FROM myProducts ;"
```

```
1|19.95|2015-03-31|Hooded sweatshirt
2|99.99|2015-03-29|Beach umbrella
3|0.99|2015-02-28|
4|29.95|2015-02-10|Male bathing suit, blue
5|49.95|2015-02-20|Female bathing suit, one piece, aqua
6|9.95|2015-01-15|Child sand toy set
7|24.95|2014-12-20|White beach towel
8|32.95|2014-12-22|Blue-striped beach towel
9|12.95|2015-03-12|Flip-flop
10|34.95|2015-01-24|Open-toed sandal
```

In the previous code cell, we used the asterisk character to select all columns from the myProducts table without listing them explicitly. This can be a useful shortcut, especially when you're developing database applications, but it isn't a recommended practice. By using the shortcut, you don't explicitly specify the database column names or their order. In a database application, if you always assume that the column names and their order in a table are fixed, you may end up with subtle bugs if someone else modifies the database tables on which your application depends. You should always explicitly name the database columns in your SELECT statements and list the order you require.

As a result, lets look at explicitly listing the columns to extract. This is a recommended practice that also allows us to control the order in which the columns are listed in the query output.

In [ ]:
```
!sqlite3 test "SELECT price, itemNumber, description FROM myProducts ;"
```

```
19.95|1|Hooded sweatshirt
99.99|2|Beach umbrella
0.99|3|
29.95|4|Male bathing suit, blue
49.95|5|Female bathing suit, one piece, aqua
9.95|6|Child sand toy set
24.95|7|White beach towel
32.95|8|Blue-striped beach towel
12.95|9|Flip-flop
34.95|10|Open-toed sandal
```

WHERE clause

Up to this point, you have only selected columns for all rows in a single table. This can be expensive in terms of query performance, especially if you only want a subset of the rows from a large table. A more efficient technique is to filter database rows by placing conditions in the WHERE clause, which is evaluated immediately after the tables are specified within the FROM clause. The rest of this section discusses some of the basic features that are enabled by using the WHERE clause, including the ability to select rows that satisfy Boolean conditions as well as join multiple tables to perform more complex queries.

The simplest and most common use of the WHERE clause is to filter the rows from a table before selecting any columns, as shown in the next two code cells.

In [ ]:
```
!sqlite3 test "SELECT p.itemNumber, p.price FROM myProducts AS p WHERE p.pri
```

```
2|99.99
5|49.95
8|32.95
10|34.95
```

In [ ]:
```
!sqlite3 test "SELECT * FROM myProducts WHERE price > 30.00 AND stockDate <
```

```
8|32.95|2014-12-22|Blue-striped beach towel
```

The first query shown in this example selects the `itemNumber` and `price` columns from the myProducts table for all rows where the price column has a value greater than

`$30.00`. The second query extends this same query to select only those columns whose price column has a value more than `$30.00` and whose stockDate column has a value less than January 1, 2006. These two query restrictions are combined in this query by using the Boolean AND operator.

You can perform a number of different Boolean operations within a WHERE clause. The following table lists and provides examples of the basic SQL Boolean operations that you can use in a query.

Operator	Example	Description
=	`p.price = 29.95`	Test if any built-in type is equal to a specified value.
<	`p.price < 29.95`	Test if any built-in type is less than a specified value.
>	`p.price > 29.95`	Test if any built-in type is greater than a specified value.
<=	`p.price <= 29.95`	Test if any built-in type is less than or equal to a specified value.
>=	`p.price >= 29.95`	Test if any built-in type is greater than or equal to a specified value.
<>	`p.price <> 29.95`	Test if any built-in type is not equal to a specified value.
IS NULL	`p.description IS NULL`	Test if an expression or value is null.
IS NOT NULL	`p.description IS NOT NULL`	Test if an expression or value is not null.

| AND | `(p.price > 29.92) AND (p.itemNumber > 5)` | Test if two expressions are both true or evaluate as nonzero. | OR | `(p.price > 29.92) OR (p.itemNumber > 5)` | Test if one or both of two expressions are true or evaluate as nonzero. | NOT | `NOT v.vendorNumber = 1` | Test if an expression is false or evaluates as zero. | BETWEEN | `p.price BETWEEN 29.95 AND 39.95` | Test if a value lies inclusively between two other values (example is equivalent to `29.95 <= p.price <= 39.95`). | LIKE | `v.vendorName LIKE 'Lun%'` | Test if a character expression matches a pattern, with the percent character (%) matching zero or more arbitrary characters and the underscore character (_) matching exactly one arbitrary character.

The first query above also introduces the AS clause, which you can use to create a table synonym. In these examples, you define a synonym p for the fully qualified table name myProducts. By defining a synonym, you can refer to table quantities by using a shorter notation. This may not seem important when only one table is being referenced in a query, but the next section shows how to join multiple tables together within a query; in that case, providing table synonyms is very useful. You can also use an AS clause to name the selected columns in a query. Doing so lets you control how the results are displayed, which is demonstrated in the next section.

Joins

The second major function performed by a WHERE clause is to join multiple tables together into a single table that can be queried more easily. Joining multiple tables is a powerful technique, and it can be complex when you're dealing with several large tables. Tables can be joined either explicitly, by using the JOIN keyword, or implicitly, by using a WHERE clause.

You join two tables by using an inner join or an outer join. An inner join is essentially the intersection of two tables, where the tables are matched by comparing the values of a key column, such as itemNumber. The resulting table is composed of only rows that were matched between the two tables. An outer join is more like a union of two tables, where the tables are matched by comparing the values of a key column, but non-matching rows are still included in the resulting table and filled with NULL values as appropriate. Writing SQL queries that use these more advanced table joins will be addressed in future articles.

In the current simple scheme, the process is simple; In the next code cell, we perform an implicit inner join of the myProducts table and the myVendors table.

In [ ]:
```
%%writefile select.sql

SELECT p.price, p.description AS 'Item', v.vendorName AS 'Vendor'
FROM myProducts AS p, myVendors AS v
  WHERE p.itemNumber = v.itemNumber ;
```

```
Writing select.sql
```

In [ ]:
```
# Execute SQL Script

!sqlite3 test < select.sql
```

```
19.95|Hooded sweatshirt|Luna Vista Limited
99.99|Beach umbrella|Luna Vista Limited
0.99||Luna Vista Limited
29.95|Male bathing suit, blue|Mikal Arroyo Incorporated
49.95|Female bathing suit, one piece, aqua|Mikal Arroyo Incorporated
9.95|Child sand toy set|Luna Vista Limited
24.95|White beach towel|Luna Vista Limited
32.95|Blue-striped beach towel|Luna Vista Limited
12.95|Flip-flop|Quiet Beach Industries
34.95|Open-toed sandal|Quiet Beach Industries
```

This query may seem complex, due primarily to its length. But by breaking it down line by line you can easily follow what's happening. First, you select two columns from the myProducts table and one column from the myVendors table and use an AS clause to name these columns for subsequent usage (in this case they are displayed by the `sqlite3` client tool. Because the query joins these two tables (by using an implicit inner join), you can select columns from both tables. In the FROM clause, you list both tables and provide aliases for them to simplify the full SQL statement. In the WHERE clause, you provide the logic for joining the two tables, by explicitly instructing the SQLite database to only select rows from the two tables that have matching values in their respective itemNumber columns. In processing this query, SQLite first pulls all rows out of the first (left) table in the query (myProducts) and finds the row with a matching value in the itemNumber column in the second (right) table in the query (myVendors).

DISTINCT

By default, when data is selected by using an SQL query, all rows that satisfy the WHERE clause are extracted from the database. In some cases, this may result in rows that have identical column values being returned. If you need to restrict your query so that only unique row values are returned, you can use the DISTINCT qualifier, as shown in the following two code cells.

In [ ]:
```
!sqlite3 test "SELECT DISTINCT vendorNumber AS 'Vendor #' FROM myVendors ;"
```

```
1
2
3
```

In [ ]:
```
!sqlite3 test "SELECT DISTINCT vendorNumber AS 'Vendor #', itemNumber as 'It
```

```
1|6
1|7
1|8
3|9
3|10
```

If you want to use the DISTINCT qualifier, it must be the first item listed in the SELECT clause, as shown in Listing 1, and you can have only one DISTINCT qualifier per SELECT clause. If the selected rows contain a column with NULL values, multiple NULL values are considered duplicates when identifying unique rows.

The first query in this listing uses the DISTINCT qualifier to restrict the output of the query to only distinct, or unique, values of the vendorNumber column, which is the only column listed in the SELECT clause. In the example schema that these articles use, there are only three vendors (with vendorNumber being restricted to 1, 2, or 3). Thus, when the DISTINCT qualifier is used in the query, only three rows are selected.

The DISTINCT qualifier, however, applies to the entire list of selected columns, so if multiple columns are listed following a DISTINCT keyword, only unique combinations of all the columns are selected. This is demonstrated in the second example, where both vendorNumber and itemNumber are listed in the SELECT clause. Because every item has a unique itemNumber, every combination of these two columns is unique, and all rows that satisfy the WHERE clause are selected; in other words, the DISTINCT qualifier has no effect on the results.

One remaining point that you may have noticed from the two previous examples is that the selected rows were not in the same order. If the order of selected rows is important, you can easily control it by using an ORDER BY clause in your query.

ORDER BY

In general, you can't assume that SQLite, or any database, will return rows from a query in a specific order. If the order is important, you can use the ORDER BY clause to have SQLite order the data that are returned by your query in a particular manner. Generally, you do so by specifying a column that should be used to provide the ordinal values for comparison as shown in the next two code cells.

In [ ]:
```
%%writefile orderby.sql

SELECT v.vendorNumber AS "Vendor #", vendorName as "Vendor",
    p.price AS "Price", p.itemNumber AS "Item #"
    FROM myProducts AS p, myVendors AS v
    WHERE p.itemNumber = v.itemNumber AND p.price > 20.0
    ORDER by v.vendorNumber ;

SELECT v.vendorNumber AS "Vendor #", vendorName as "Vendor",
    p.price AS "Price", p.itemNumber AS "Item #"
    FROM myProducts AS p, myVendors AS v
    WHERE p.itemNumber = v.itemNumber AND p.price > 20.0
    ORDER BY v.vendorNumber ASC, p.price DESC ;
```

```
Writing orderby.sql
```

In [ ]:
```
!sqlite3 test < orderby.sql
```

```
1|Luna Vista Limited|99.99|2
1|Luna Vista Limited|24.95|7
1|Luna Vista Limited|32.95|8
2|Mikal Arroyo Incorporated|29.95|4
2|Mikal Arroyo Incorporated|49.95|5
3|Quiet Beach Industries|34.95|10
1|Luna Vista Limited|99.99|2
1|Luna Vista Limited|32.95|8
1|Luna Vista Limited|24.95|7
2|Mikal Arroyo Incorporated|49.95|5
2|Mikal Arroyo Incorporated|29.95|4
3|Quiet Beach Industries|34.95|10
```

In the previous example, the first query uses the ORDER BY clause to list a subset of all the rows in the table that results from joining the myVendors table to the myProducts table. The rows are ordered by vendorNumber (the subset is constructed by applying the WHERE clause). An ORDER BY clause can take either a column name, as in this example, or a column number, which is taken from the order in which the columns are listed after the SELECT keyword.

You can also specify multiple columns to use during the sorting process and even specify ASC for ascending order, which is the default, or DESC for descending order. For example, if you used the ORDER BY 1 DESC, 4 DESC clause in the first query, the query would return the same rows, but they would be ordered by using the vendorNumber column as the primary sort column in descending order followed by the itemNumber column as the secondary sort column in descending order.

Although using column numbers may seem like a handy shortcut, it generally isn't a good idea. To see why, consider what happens if you modify the columns listed in a SELECT clause or just modify their order. If you forget to modify the numbers used in the ORDER BY clause, the query will break-or worse, return bad data. In general, it's a *best practice* to always be explicit and specify the column names directly, even if doing so means more typing.

Query Math

Selecting columns from a database provides a number of useful benefits, but being able to compute and select quantities based on data in a table opens up even more possibilities. SQLite,as does any SQL database, provides several mathematical operators, the most common of which are listed in the following table, that you can use in either a SELECT clause or a WHERE clause.

Operator	Example	Description
unary +	+1.0	A noop, or no operation, as +4 = 4
unary -	-p.price	Changes the sign of the value to which it's applied
+	p.itemNumber + 10	Adds the second value to the first value
-	p.itemNumber - 10	Subtracts the second value from the first value
*	p.price * 1.0825	Multiplies the first value by the second value
/	p.price / 100.0	Divides the first value by the second value

Using these operators is straightforward because they generally behave exactly as you expect. For example, if the sales tax is 8.25%, you can return the price for an item both before and after sales tax has been applied by using `SELECT price, price * 1.0825 FROM myProducts ;`. As another example, if you have a column called `numberItems` that tracks the number of items purchased and another column called `price` that contains the price at which they're purchased, you can return the total amount paid for those items at a given price by using `numberItems * price`. Several of the queries shown in this IPython Notebook provide additional examples of how to use these operators.

The only concern when using these operators arises from complications that result from using different data types, such as integer or floating-point, in a mathematical operation. If both operands are the same data type, the result type will be the same. If you're performing division, this can result in truncation (for example, if you're using two integer values), which might cause unexpected problems. On the other hand, if the two operands are different data types, the result type is promoted to the more complex type.

SQL Functions

SQL is a powerful and expressive language that can be used to perform a wide range of actions. Part of the SQL language's power comes from its ability to directly interact with a variety of data types. Some of the greatest power of a relational database arises from the inherent functions that it provides and from the extensibility enabled by allowing users to create new functions. SQLite provides functions (see the following SQLite tutorial for more information) in three different categories:

1. Core functions
2. Aggregate functions
3. Date and Time functions

Of these built-in functions, we will most likely use the aggregate functions, which operate on multiple rows. Aggregate functions-also known as set functions in SQL-92 or, more informally, as column functions-return a computed quantity from a column over a number of rows. SQLite supports the following five aggregate functions (a sixth function, `group_concat`, is not listed).

Function	Example	Description
AVG	AVG(p.price)	Returns the average value of a column from all rows that satisfy an expression. Can only be used with built-in numeric data types. The precision of the returned value is defined by the precision of the column being evaluated.
COUNT	COUNT(p.price)	Returns the number of rows that satisfy an expression, such as a query. Can be used with any data type.
MAX	MAX(p.price)	Returns the maximum value of a column from all rows that satisfy an expression. Can only be used with built-in data types.
MIN	MIN(p.price)	Returns the minimum value of a column from all rows that satisfy an expression. Can only be used with built-in data types.
SUM	SUM(p.price)	Returns the sum of a column over all rows that satisfy an expression. Can only be used with built-in numeric data types.

These aggregate functions can often be used to quickly find useful information that might otherwise be difficult to identify, as shown in Listing 4.

```
SELECT COUNT(p.itemNumber) AS Number,
        AVG(p.price) AS Average,
        MIN(p.stockDate) AS "First Date", MAX(p.stockDate) AS
"Last Date"
    FROM myProducts AS p ;
```

Listing 4 uses four of the five aggregate functions to get summary information about the data in the myProducts table. The COUNT function indicates that the table includes ten rows (because the query didn't use a WHERE clause to restrict the rows selected from the table). The AVG function calculates the average price of all items in the myProducts table. Finally, the MIN and MAX functions extract the minimum and maximum dates from the myProducts table.

DELETE

To delete data in a SQLite database, you use the SQL DELETE statement, which can delete either all rows in a table or a specific subset of rows. The formal syntax for the SQL DELETE statement is remarkably simple:

```
DELETE FROM tableName
    [WHERE clause]
```

The DELETE statement deletes all rows from the specified table that satisfy an optional WHERE clause. If no WHERE clause is included, all rows in the table are deleted. To demonstrate this use of the DELETE statement, we can create a temporary table, insert several rows, and delete them all.

In [ ]:
```
%%writefile delete.sql

-- First create the temporary table
CREATE TABLE temp (aValue INT) ;

-- Insert fake data
INSERT INTO temp VALUES(0), (1), (2), (3) ;

-- Count rows in the table
SELECT COUNT(*) AS COUNT FROM temp ;

-- Delete all rows
DELETE FROM temp ;

-- Count all rows in the table
SELECT COUNT(*) AS COUNT FROM temp ;

-- Now drop the temporary table

DROP TABLE temp ;
```

```
Overwriting delete.sql
```

In [ ]:
```
!sqlite3 test < delete.sql
```

```
4
0
```

The previous example created a single-column temporary table to hold a single integer value. Next we inserted four rows into the database and issued a SELECT statement to verify that the new table contained four rows. By using an unconstrained DELETE statement, we delete all four rows from the temporary table, which is verified by the second SELECT statement, which indicates that the temporary table contains zero rows. Finally, the DROP TABLE statement deletes the empty table from the schema.

In general, however, you don't want to delete all rows from a table; instead, you'll selectively delete rows. To do this, you create an appropriate WHERE clause that identifies all rows of interest. The syntax for the WHERE clause that you can use with a DELETE statement is identical to that discussed previously when we presented the full SQL SELECT statement syntax. The basic building blocks for constructing a Boolean expression within a WHERE clause were presented in an earlier table. The following example demonstrates using a WHERE clause in a DELETE statement, where we delete all rows that satisfy at least one of two conditions.

In [ ]:
```
%%writefile delete2.sql

-- First display data
SELECT itemNumber, description FROM myProducts ;

-- Selectively delete rows
DELETE FROM myProducts
    WHERE description LIKE '%towel%' OR itemNumber <= 3 ;

-- Confirm the proper deletion
SELECT itemNumber, description FROM myProducts ;
```

```
Writing delete2.sql
```

In [ ]:
```
!sqlite3 test < delete2.sql
```

```
1|Hooded sweatshirt
2|Beach umbrella
3|
4|Male bathing suit, blue
5|Female bathing suit, one piece, aqua
6|Child sand toy set
7|White beach towel
8|Blue-striped beach towel
9|Flip-flop
10|Open-toed sandal
4|Male bathing suit, blue
5|Female bathing suit, one piece, aqua
6|Child sand toy set
9|Flip-flop
10|Open-toed sandal
```

In this example, the DELETE statement includes a WHERE clause that identifies five rows. The WHERE clause contains two expressions that are joined by the OR operator, which means that if either expression evaluates as TRUE for a specific row, that row will be deleted.

The first expression finds all rows that contain the word "towel" in the product description. If you recall, there are two towels in the myProducts table, with itemNumber column values of 7 and 8. The other expression selects all rows with an itemNumber column value less than or equal to 3. The contents of the myProducts table are finally displayed with a simple SELECT statement, demonstrating that only five of the original ten rows remain in the table.

Although this example doesn't explicitly demonstrate their use, you can also include the SQL functions to gain more control over the selection of rows for deletion. These same functions and other operators that can be used in the WHERE clause of the DELETE statement also can be used with the UPDATE statement to selectively modify the values of rows in a table, as described in the next section.

UPDATE

The last SQL task for dealing with data that you need to address is updating specific column values for selected rows in a table. At some level, the SQL UPDATE statement is the union of the SQL INSERT and DELETE statements, because you must select rows to modify as well as specify how to modify them. Formally, the UPDATE statement syntax is straightforward, because you must specify the new column values for the set of rows to be updated:

```
UPDATE tableName
    SET columnName = Value
    [ , columnName = Value} ]*
    [WHERE clause]
```

As shown in this SQL syntax, an SQL UPDATE statement must have, at a minimum, one SET component to update one column, along with one or more SET components and a WHERE clause, both of which are optional. If the WHERE clause isn't included, the UPDATE statement modifies the indicated columns for all rows in the table.

Issuing an UPDATE statement is fairly easy, as shown in the following code example, where we modify two columns of a single row.

In [ ]:
```
%%writefile update.sql

-- Extract the test row
SELECT itemNumber, price, stockDate FROM myProducts WHERE itemNumber = 6 ;
```

```
-- Update the row
UPDATE myProducts SET price = price * 1.25, stockDate = date('now')  WHERE i

-- Show the new result
SELECT itemNumber, price, stockDate FROM myProducts WHERE itemNumber = 6 ;
```

```
Writing update.sql
```

In [ ]:
```
!sqlite3 test < update.sql
```

```
6|9.95|2015-01-15
6|12.4375|2024-10-12
```

This example wraps a single UPDATE statement with SELECT statements to demonstrate the change to the target row. The SELECT statements both select three columns from the myProducts table for a single row (the row with the value 6 in the itemNumber column). The UPDATE statement modifies both the price and the stockDate columns for this specific row. The value in the price column is increased by 25% (for example, perhaps due to the item's popularity), and the stockDate column is modified to hold the current date, which can be obtained easily with SQLite by using the built-in `date` function with an argument of `now` in an SQL query.

The previous example demonstrated how to modify multiple column values for a specific row in a single table. However, sometimes the logic to select rows to update is more complex. For example, suppose you need to modify the price of all objects in the myProducts table that you obtain from Quiet Beach Industries, which has a value of 3 in the vendorNumber column in the myVendors table. To do this, you need to use an embedded query:

In [ ]:
```
%%writefile update2.sql

-- Update the table
UPDATE myProducts
    SET price = price * 1.10, description = 'NEW: ' || description
    WHERE itemNumber IN
        ( SELECT v.itemNumber
          FROM myProducts as p, myVendors as v
          WHERE p.itemNumber = v.itemNumber AND v.vendorNumber = 3 ) ;

-- Show new results
SELECT * FROM myProducts ;
```

```
Writing update2.sql
```

In [ ]:
```
!sqlite3 test < update2.sql
```

```
4|29.95|2015-02-10|Male bathing suit, blue
5|49.95|2015-02-20|Female bathing suit, one piece, aqua
6|12.4375|2024-10-12|Child sand toy set
9|14.245|2015-03-12|NEW: Flip-flop
10|38.445|2015-01-24|NEW: Open-toed sandal
```

In this example, the UPDATE statement modifies the price and description columns for all products that are obtained from the vendor with a value of 3 in the vendorNumber column in the myVendors table. Because you can't do a simple join within an UPDATE statement, you must include a subquery in the WHERE clause to extract the itemNumber rows that correspond to products from Quiet Beach Industries. The WHERE clause in the UPDATE statement uses the IN operator to select those rows that have an itemNumber column in the set of values selected by the embedded query.

Two types of queries can be used in the WHERE clause of an UPDATE statement: a scalar subquery and a table subquery. A scalar subquery is an embedded query that returns a single row that contains a single column-essentially, a single value, which is known as a scalar. You can use a scalar subquery to select a specific value that will be used in the expression of the WHERE clause. For example, `itemNumber = (Scalar Subquery)` updates any rows that have a value in the itemNumber column that matches the result of the scalar subquery.

A table subquery, on the other hand, can return multiple rows that generally only have one column. In certain instances, a table subquery can contain multiple columns. To use a table subquery, you need to use an SQL operator to combine the embedded query with a Boolean expression. For example, this was shown in the previous code listing, where the IN operator selected all rows from the myProducts table that were produced by Quiet Beach Industries.

ALTER TABLE

The previous section discussed modifying the data that already exists in a table. The other possibility is modifying the structure, or schema, of a database table. This can take the form of adding a column, changing the data type for a column, adding a constraint, or even deleting a column. This process isn't easy, which is one reason to be careful when you initially design your schema. If you do need to modify the structure of a table, you should use a temporary table.

In [ ]:
```
%%writefile rename.sql

-- Create New table with extra column
```

```
CREATE TABLE newProducts (
    itemNumber INT NOT NULL,
    price REAL,
    stockDate TEXT,
    count INT NOT NULL DEFAULT 0,
    description TEXT
) ;

-- New copy old table into new table

INSERT INTO newProducts(itemNumber, price, stockDate, description)
    SELECT itemNumber, price, stockDate, description FROM myProducts ;

-- Drop old table
DROP TABLE myProducts ;

-- Now Rename new table to old table name
ALTER TABLE newProducts RENAME TO myProducts ;

-- Show the results
SELECT * FROM myProducts ;
```

```
Writing rename.sql
```

In [ ]:
```
# Execute SQL Script

!sqlite3 test < rename.sql
```

```
4|29.95|2015-02-10|0|Male bathing suit, blue
5|49.95|2015-02-20|0|Female bathing suit, one piece, aqua
6|12.4375|2024-10-12|0|Child sand toy set
9|14.245|2015-03-12|0|NEW: Flip-flop
10|38.445|2015-01-24|0|NEW: Open-toed sandal
```

As this example shows, to modify a table-in this case, to add a new `count` column to the `myProducts` table-you first create a table that has the exact schema you require. This example requires that it always have a valid value by including the column constraint NOT NULL and assigns a default value of 0 to the count column by using the column constraint DEFAULT 0. Notice how you can combine multiple column constraints by listing them sequentially.

The next step is to copy the existing data from the original table to the new table. You can do so by using an SQL INSERT statement that uses a subquery to get the values to insert. This is a powerful technique that lets you easily copy all or part of an existing table into a second table.

After you've created the new table and copied the appropriate data, you drop the old table by using an SQL DROP TABLE statement and rename the new table to the original name by using an SQL RENAME TABLE statement. The rename operation is straightforward: Rename the oldTableName to the newTableName, but don't supply a schema name for the new table name because the RENAME operation can't move a table between different database schemas. This example concludes by issuing a SELECT statement to display the schema and contents of the new myProducts table. As you can see, the new table has five columns, and the count column is always zero. At this point, a real application would modify the count column appropriately by issuing the necessary SQL UPDATE statements.

CREATE INDEX

Relational databases support the concept of an index to speed up queries. An index is often used when a particular column (or columns) is frequently involved in either a WHERE clause or a table join. The syntax for creating an index is rather simple:

```
CREATE [UNIQUE] INDEX idx_name ON table_name(column
[column]*)
[WHERE]
```

In this format, we create an index on one or more columns in a given table. If UNIQUE is present in the index creation, only non-duplicate entries are allowed in the index. The columns can also be followed by either ASC or DESC to indicate the column sort order for the index. Finally, if a WHERE clause is included the index is only a partial index since it does not cover the entire table.

As an example, we can create an index on the myProducts table by using the itemNumber column:

```
CREATE INDEX itn ON myProducts(itemNumber) ;
```

Advanced Features

Of course we have only scratched the surface of SQL, despite the length of this IPython Notebook. We have not discussed views, which effectively turns a SQL query into a new, read-only table, or triggers, which are database operations that are automatically performed when a specific event occurs.

Another useful feature is the LIMIT clause. We can use this to restrict the number of rows returned by a particular query. For example, `SELECT * FROM myProducts LIMIT 5 ;` will only return five rows.

Additional References

Several sites exist that allow you to try out SQL commands online.

1. W3 Schools SQL, a general SQL demo site.
2. SQLZoo, allows you to specify the Relational Database to target.

Exercises

The following 5 lab exercises will work through the same flight-based use case.

You should understand the next couple of cells, which prepare the database used in the exercises to come.

```
In [ ]: CSV_PATH = "2001.csv"
```

In the following code cell, we use the writefile magic function to write a schema that imports 2001.csv and creates a new table named flights.

```
In [ ]: %%writefile airport.sql

        DROP TABLE IF EXISTS flights;

        CREATE TABLE flights (
            year INT,
            month INT,
            dayOfMonth INT,
            dayOfWeek INT,
            actualDepartureTime INT,
            scheduledDepartureTime INT,
            arrivalArrivalTime INT,
            scheduledArrivalTime INT,
            uniqueCarrierCode TEXT,
            flightNumber INT,
            tailNumber TEXT,
            actualElapsedTime INT,
            scheduledElapsedTime INT,
            airTime INT,
            arrivalDelay INT,
            departureDelay INT,
            originCode TEXT,
            destinationCode TEXT,
            distance INT,
            taxiIn INT,
            taxiOut INT,
            cancelled INT,
            cancellationCode TEXT,
            diverted INT,
            carrierDelay INT,
            weatherDelay INT,
            nasDelay INT,
            securityDelay INT,
            lateAircraftDelay INT
        );

        .separator ,
        .import 2001.csv flights

        -- Our file has a header. The following line deletes the header.
        DELETE FROM flights WHERE Year='Year';
```

```
Writing airport.sql
```

We will name our database assignment.db and use IPython's ! magic to redirect the airport.sql code to sqlite3.

In [ ]:
```
!sqlite3 assignment.db < airport.sql
```

Exercise 1: Flight Count

- In the following code cell, write an SQL statement that counts the number of rows in the flights table.

Note that comments in SQL begin with -- (not #).

In [ ]:
```
%%writefile count_lines_flights.sql

-- Counts the numbers of rows in the flights table

SELECT COUNT(*) AS nlines_flights FROM flights;
```

```
Overwriting count_lines_flights.sql
```

Run the following code cell to check the output of count_lines_flights.sql. Make sure that this cell passes.

In [ ]:
```
nlines_flights = !sqlite3 assignment.db < count_lines_flights.sql
print(nlines_flights.s)
assert nlines_flights.s == "49"
```

```
49
```

Exercise 2: Creating Another Table

We will use another CSV file iata.csv to create a new table.

- In the following code cell, write your own schema and SQL script to import iata.csv and create a new table named iata. The names of the columns should be

  airportID, name, city, country, iata, icao, latitude, longitude, altitude, timeZone, dst, tzDatabaseTimeZone

Also make sure that your data types are all correct. Use head or otherwise to check the CSV file. If a value is enclosed by quotation marks, it should be a TEXT. If a field has a decimal point, it should be a REAL. If a field is a number with no decimal point, it should be an INT.

```
In [ ]: %%writefile import_iata.sql

-- Creating iata table
DROP TABLE IF exists iata;

CREATE TABLE iata (
airportID INT,
name TEXT,
city TEXT,
country TEXT,
iata TEXT,
icao TEXT,
latitude REAL,
longitude REAL,
altitude INT,
timeZone INT,
dst TEXT,
tzDatabaseTimeZone TEXT
);
.separator ,
.import iata.csv iata
```

```
Overwriting import_iata.sql
```

The following code cell should create a new table named iata in the database assignment.db.

```
In [ ]: !sqlite3 assignment.db < import_iata.sql
```

I think it's a good idea to print out a few lines and see if the table looks okay.

In [ ]:
```
!sqlite3 assignment.db "SELECT * FROM iata LIMIT 10"
```

```
1|Goroka|Goroka|Papua New Guinea|GKA|AYGA|-6.081689|145.391881|5282|10|U|Pac
ific/Port_Moresby
2|Madang|Madang|Papua New Guinea|MAG|AYMD|-5.207083|145.7887|20|10|U|Pacifi
c/Port_Moresby
3|Mount Hagen|Mount Hagen|Papua New Guinea|HGU|AYMH|-5.826789|144.295861|538
8|10|U|Pacific/Port_Moresby
4|Nadzab|Nadzab|Papua New Guinea|LAE|AYNZ|-6.569828|146.726242|239|10|U|Paci
fic/Port_Moresby
5|Port Moresby Jacksons Intl|Port Moresby|Papua New Guinea|POM|AYPY|-9.44338
3|147.22005|146|10|U|Pacific/Port_Moresby
6|Wewak Intl|Wewak|Papua New Guinea|WWK|AYWK|-3.583828|143.669186|19|10|U|Pa
cific/Port_Moresby
7|Narsarsuaq|Narssarssuaq|Greenland|UAK|BGBW|61.160517|-45.425978|112|-3|E|A
merica/Godthab
8|Nuuk|Godthaab|Greenland|GOH|BGGH|64.190922|-51.678064|283|-3|E|America/God
thab
9|Sondre Stromfjord|Sondrestrom|Greenland|SFJ|BGSF|67.016969|-50.689325|165|
-3|E|America/Godthab
10|Thule Air Base|Thule|Greenland|THU|BGTL|76.531203|-68.703161|251|-4|E|Ame
rica/Thule
```

Make sure that the following code cell doesn't raise any errors. You don't have to understand what the SQL statement means. It simply checks if there exists a table named iata in the database and prints out iata if it exists.

In [ ]:
```
iata_exists = !sqlite3 assignment.db "SELECT name FROM sqlite_master WHERE t
assert iata_exists.s == "iata"
```

You don't have to understand the code in the following tests, but make sure that your SQL script passes the tests.

In [ ]:
```
iata_info = !sqlite3 assignment.db "PRAGMA table_info(iata)"
iata_names = [i.split("|")[1] for i in iata_info]
iata_names_answer = [
    "airportID",
    "name",
    "city",
    "country",
    "iata",
    "icao",
    "latitude",
    "longitude",
    "altitude",
    "timeZone",
    "dst",
    "tzDatabaseTimeZone"
]
assert len(iata_names) == len(iata_names_answer)
assert set(iata_names) == set(iata_names_answer)
```

Exercise 3: Joining Tables

-Join flights and iata tables by matching the IATA codes of the destinationCode column in flights to the IATA codes of the iata column in iata. Combine them into a new table named myTable, which should have the following columns: month, dayOfMonth, uniqueCarrierCode, flightNumber, scheduledDepartureTime, diverted, city

All columns excpet city come from the flights table. The city column comes from the iata table, and it's the full city name of the airport that corresponds to destinationCode.

In other words, if we did

```
SELECT
month,
dayOfMonth,
uniqueCarrierCode,
flightNumber,
scheduledDepartureTime,
diverted,
destinationCode
FROM flights
LIMIT 10;
```

we would have

```
1|17|US|375|1810|0|CLT
1|18|US|375|1810|0|CLT
1|19|US|375|1810|0|CLT
1|20|US|375|1810|0|CLT
1|21|US|375|1810|0|CLT
1|22|US|375|1810|0|CLT
1|23|US|375|1810|0|CLT
1|24|US|375|1810|0|CLT
1|25|US|375|1810|0|CLT
1|26|US|375|1810|0|CLT
```

Translate all the IATA codes (CLTs in this example) to actual city names so that when you do

```
SELECT * FROM myTable LIMIT 10;
```

you get

```
1|17|US|375|1810|0|Charlotte
1|18|US|375|1810|0|Charlotte
1|19|US|375|1810|0|Charlotte
1|20|US|375|1810|0|Charlotte
1|21|US|375|1810|0|Charlotte
1|22|US|375|1810|0|Charlotte
1|23|US|375|1810|0|Charlotte
1|24|US|375|1810|0|Charlotte
1|25|US|375|1810|0|Charlotte
1|26|US|375|1810|0|Charlotte
```

In [ ]:
```
%%writefile join.sql

-- Joining flights table and iata table
DROP TABLE IF exists myTable;
CREATE TABLE myTable AS
SELECT f.month, f.dayOfMonth, f.uniqueCarrierCode, f.flightNumber, f.schedul
FROM flights f, iata i
WHERE f.destinationCode = i.iata
```

```
Overwriting join.sql
```

In [ ]:
```
!sqlite3 assignment.db < join.sql
```

In [ ]:
```
!sqlite3 assignment.db "SELECT * FROM myTable LIMIT 10;"
```

```
1|17|US|375|1810|0|Charlotte
1|18|US|375|1810|0|Charlotte
1|19|US|375|1810|0|Charlotte
1|20|US|375|1810|0|Charlotte
1|21|US|375|1810|0|Charlotte
1|22|US|375|1810|0|Charlotte
1|23|US|375|1810|0|Charlotte
1|24|US|375|1810|0|Charlotte
1|25|US|375|1810|0|Charlotte
1|26|US|375|1810|0|Charlotte
```

Exercise 4: Inserting

Insert a new row into myTable. This flight

- took place on September 9, 2001,
- its uniqueCarrierCode was INFO,
- its flightNumber was 490,
- its scheduledDepartureTime was 0800,

- was diverted (i.e. diverted == 1), and
- left from San Francisco.

In [ ]:
```
%%writefile insert.sql

-- Inserting a new row into myTable
INSERT INTO myTable (month, dayOfMonth, uniqueCarrierCode, flightNumber, sch
VALUES(9, 9, 'INFO', 490, 0800, 1, 'San Francisco')
```

```
Overwriting insert.sql
```

In [ ]:
```
%%writefile delete3.sql
-- Deleting an accidently duplicated row
DELETE FROM myTable
WHERE flightNumber = 490
```

```
Overwriting delete3.sql
```

In [ ]:
```
!sqlite3 assignment.db < delete3.sql
```

In [ ]:
```
!sqlite3 assignment.db < insert.sql
```

In [ ]:
```
info_month = !sqlite3 assignment.db "SELECT month FROM myTable WHERE uniqueC
info_day = !sqlite3 assignment.db "SELECT dayOfMonth FROM myTable WHERE uniq
info_flight_no = !sqlite3 assignment.db "SELECT flightNumber FROM myTable WH
info_crs_dep = !sqlite3 assignment.db "SELECT scheduledDepartureTime FROM my
info_diverted = !sqlite3 assignment.db "SELECT diverted FROM myTable WHERE u
info_dest = !sqlite3 assignment.db "SELECT city FROM myTable WHERE uniqueCar

print('''
UniqueCarrierCode: {0}
Month: {1}
Day: {2}
Flight Number: {3}
Scheduled Departure Time: {4}
Diverted: {5}
Origin City: {6}
'''.format(
    "INFO",
    info_month.s,
    info_day.s,
    info_flight_no.s,
    info_crs_dep.s,
    info_diverted.s,
    info_dest.s
    )
)
```

```
UniqueCarrierCode: INFO
Month: 9
Day: 9
Flight Number: 490
Scheduled Departure Time: 800
Diverted: 1
Origin City: San Francisco
```

In [ ]:
```
assert "9" == info_month.s
assert "9" == info_day.s
assert "490" == info_flight_no.s
assert "800" == info_crs_dep.s
assert "1" == info_diverted.s
assert "San Francisco" == info_dest.s
```

Exercise 5: Query Maximum

- Compute the maximum of the departureDelay column in the flights table.

In [ ]:
```
%%writefile get_maximum_depdelay.sql

-- Computing the maximum departure delay
SELECT MAX(departureDelay) AS maximum_depdelay FROM flights WHERE departureD
```

```
Overwriting get_maximum_depdelay.sql
```

In [ ]:
```
maximum_depdelay = !sqlite3 assignment.db < get_maximum_depdelay.sql
print(maximum_depdelay)
assert maximum_depdelay.s == '100'
```

```
['100']
```