SQLite

Introducing SQLite

SQLite is a compact relational database management system (rdbms) that requires no installation other than having a single executable file. It creates and stores the schema, tables and data for each database as a single file which is parsed when the database is opened. In general SQLite uses standard sql syntax but with limited alter table support.

SQL statements may extend for several lines and must be terminated with a semicolumn before the statement will be evaluated by the database engine. SQLite dot commands which are specific to this system, serve several special functions. They are entered on only one line and do not require a terminating character. These dot commands duplicate functions found in other databases. For example ".tables" is equivalent to "show tables" function found in many SQL database management systems. More information on the available dot commands can be found in the SQLite notes section.

Starting SQLite in Microsoft Windows

Open a command prompt window by either going through the Start menu and selecting the command prompt listing which can usually be found in the accessories menu, or enter "cmd.exe" in the run dialog box of the Start menu. Using the change directory command (cd), navigate to the folder where the sqlite program is located. Here is one example, although the path shown in red text will likely be different on your computer. "cd C:\Databases\sqlite\ ". At the command line enter "sqlite3 path and database name " If the database file does not currently exist then it will be created. If no path is specified then the database file will be placed in the folder in which the program "sqlite3.exe" resides.

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\My Documents\>cd C:\Databases\sqlite
C:\Databases\sqlite>sqlite3 C:\Databases\inventoryctrl.db
SQLite version 3.7.4
Enter ".help" for instructions
```

```
sqlite> .databases
seq     name            file
---     ---------------  ---------------------------------------------------------
0       main            C:\Databases\inventoryctrl.db
1       temp            C:\DOCUME~1\COLINR~1\LOCALS~1\Temp\etilqs_i12OPG64gGySrwGX
sqlite >
```

Colin Riley -- January 2011 --

SQLite

Creating a Table

```
CREATE TABLE table_name(fieldname_1 data_type, fieldname_2 data_type, fieldname_3 data_type);
```

Let us say you create the following table to track your inventory and costs for your supplies and equipment where each field name is followed by a data type.

```
sqlite> CREATE TABLE inventory(StockNumber INTEGER PRIMARY KEY,Descrip VARCHAR(50),OnHandQuan INTEGER,PackQty
INTEGER,PackCost FLOAT);
sqlite>
```

There is technically no requirement to declare data types when creating a table in SQLite with the possible exception of creating an INTEGER PRIMARY KEY field. That being said, it is still a good idea to do so if you wish to make your database portable to another database management system such as MYSQL or ORACLE, now or sometime in the future.

Putting Data into the Table

Now we fill our currently empty table with the following insert statements containing our data. Note that string values are surrounded by single quotes, numeric values are not.

```
INSERT INTO table_name(fieldname_1,fieldname_2,fieldname_3)VALUES ('value a','value b',0.000);
```

```
sqlite> INSERT INTO inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VALUES (51002,'AA Dry Cells 4 Pack',173,12,9.00);
sqlite> INSERT INTO inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VALUES (51004,'AA Dry Cells 8 Pack',5,12,16.80);
sqlite> INSERT INTO inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VALUES (43512,'10W-30 Motor Oil, Quart',36,12,18.20);
sqlite> INSERT INTO inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VALUES (51013,'D Dry Cells 8 Pack',19,12,90.20);
sqlite> INSERT INTO inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VALUES (23155,'Shovel Pointed Long
Handle',1500,1,9.82);
sqlite> INSERT INTO inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VALUES (51001,'AAA Dry Cells 4 Pack ',92,12,9.00);
sqlite> INSERT INTO inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VALUES (43111,'White Gas Gallon Can',14,4,14.75);
```

Let us insert another record for 5W-Motor oil item number 43522

```
sqlite> INSERT INTO inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VA LUES (43512,'5W-30 Motor Oil, Quart',17,12,18.20);
```

```
SQL error: PRIMARY KEY must be unique
sqlite>
```

Oops the wrong stock number was used. In the table definition above, the StockNumber column was specified to be the primary key of the table meaning that each value in that column must be unique. If the statement is reentered with the correct and unique StockNumber then the record will be added to the table. Let's do a simple select query to show that the records have been added to the table properly.

```
sqlite> select * from inventory;
```

```
StockNumber|Descrip|OnHandQuan|PackQty|PackCost
23155|Shovel Pointed Long Handle|1500|1|9.82
43111|White Gas Gallon Can|14|4|14.75
43512|10W-30 Motor Oil, Quart|36|12|18.2
43522|5W-30 Motor Oil, Quart|17|12|18.2
51001|AAA Dry Cells 4 Pack |92|12|9.0
51002|AA Dry Cells 4 Pack |173|12|9.0
```

```
51004|AA Dry Cells 8 Pack|5|12|16.8
51013|D Dry Cells 8 Pack|19|12|90.2
sqlite>
```

It is possible to use an INSERT statement that does not list column names as shown below.

```
INSERT INTO table_name VALUES ('value 1',NULL,'value 2 ,'','value 3');
```

However the values listed in the INSERT statement must be in the same order as they appear in the CREATE TABLE statement and a NULL value or empty quotes must be included for the values that are omitted in the sequence. Use with extreme caution.

Updating Records in a Table

If you look at the first record in the Select Query results you will see that the stated on hand quantity of item number 23155 is 1500 when in fact there are only 15 shovels on hand. This can be corrected by using an UPDATE statement.

```
UPDATE inventory SET OnHandQuan= 15 WHERE StockNumber = 23155;
```

```
sqlite> UPDATE inventory SET OnHandQuan= 15 WHERE StockNumber = 23155;
sqlite> SELECT * FROM inventory WHERE StockNumber = 23155;
StockNumber|Descrip|OnHandQuan|PackQty|PackCost
23155|Shovel Pointed Long Handle|15|1|9.82
sqlite>
```

Note that it is very important to include a WHERE clause specifying the correct criteria for the records you wish to update or all the records in the table will be updated.

Deleting Records From a Table

Here we have a circumstance where the Stock Number was entered incorrectly. While it is certainly possible to use an update statement to fix this, instead we will reenter the record with the correct stock number and delete the incorrect record.

```
DELETE FROM inventory WHERE StockNumber = 149;
```

```
sqlite> select * from inventory;
StockNumber|Descrip|OnHandQuan|PackQty|PackCost
149|Ball Point Pens Blue Fine tip, 12 pack|92|20|15.37
23155|Shovel Pointed Long Handle|15|1|9.82
43111|White Gas Gallon Can|14|4|14.75
43512|10W-30 Motor Oil, Quart|36|12|18.2
43522|5W-30 Motor Oil, Quart|17|12|18.2
51001|AAA Dry Cells 4 Pack |92|12|9.0
51002|AA Dry Cells 4 Pack |173|12|9.0
51004|AA Dry Cells 8 Pack|5|12|16.8
51013|D Dry Cells 8 Pack|19|12|90.2
sqlite> sqlite> INSERT INTO
inventory(StockNumber,Descrip,OnHandQuan,PackQty,PackCost)VALUES (75149,'Ball Point
Pens Blue Fine tip, 12 pack',92,20,15.37);
sqlite> DELETE FROM inventory WHERE StockNumber = 149;
sqlite>
```

Do not use a DELETE command without a "WHERE" clause unless you intend to discard all the records in a table. Also much like the UPDATE command, it is very important to specify the right criteria to be certain that you are in fact deleting the correct records. It is often worthwhile to test the WHERE clause to be used in the DELETE statement by first running a SELECT query using the same criteria.

The .import command

Records from a delimited text file can be added to a SQLite table by using the ".import" command. There are however several limitations and considerations. The table must already exist in the database. The data in each field in the source document must be arranged in the same order as the column names in the SQLite table definition and there must be the same number of fields per line in the text document as there are columns in the table definition. Text values contained in quotes will retain the quotation marks with the text.

```
.import path/filename.txt tablename
```

Remember that the default delimiter for SQLite is the pipe "|". If the text file that the data is coming from uses a different delimiter such as a comma then the ".separator" command must to be used to change it. The example below uses the tilde (~) symbol as a delimiter in the source files because commas were present in some of the values in the document as show in the excerpt below.

```
HARTSHORN, SLATS OF~AMMONIUM CARBONATE
SALT OF HARTSHORN~AMMONIUM CARBONATE
MURIATE OF AMMONIA~AMMONIUM CHLORIDE
SAL AMMONIAC~AMMONIUM CHLORIDE
```

```
sqlite> CREATE TABLE chem_name(old_name TEXT,chemical TEXT);
sqlite> .separator "~" sqlite> .import chempart1.txt chem_name
sqlite> SELECT * FROM chem_name limit 8;
```

```
old_name                         chemical
-------------------------------  -----------------------------------
HARTSHORN, SLATS                 AMMONIUM CARBONATE
OF
SALT OF HARTSHORN                AMMONIUM CARBONATE
MURIATE OF AMMONIA               AMMONIUM CHLORIDE
SAL AMMONIAC                     AMMONIUM CHLORIDE
TARTAR EMETIC                    ANTIMONY AND POTASIUM TARTRATE
BUTTER OF ANTIMONY               ANTIMONY TRICHLORIDE
ORPIMENT                         ARSENIC TRISULFIDE
BARYTA                           BARIUM OXIDE
```

```
sqlite>
sqlite> .import C:/Databases/chempart2.txt chem_name
sqlite> SELECT COUNT(old_name) FROM chem_name;
COUNT(old_name)
------------------------------
81
sqlite>
```

The separator for tab delimited files is "\t".

Sqlite Datatypes

Sqlite is a typeless database and with the exception of a field that has been declared as an INTEGER PRIMARY KEY, just about any type of a data can be placed in any column in a SQLite table. SQLite will automatically class each data item in one of the following catagories.

- NULL - where there is no value in the field, not even a zero or an empty string.
- INTEGER - The value is a positive or negative integer.
- REAL - The value is a floating point number
- TEXT - Text string
- BLOB - Data stored exactly as it was entered.

```
sqlite> CREATE TABLE datatype(linenum INTEGER PRIMARY KEY,testdata INTEGER);
sqlite> INSERT INTO datatype(linenum,testdata) VALUES(1,-34);
sqlite> INSERT INTO datatype(linenum,testdata) VALUES(2,'This a text field');
sqlite> INSERT INTO datatype(linenum,testdata) VALUES(3,3.1415);
sqlite> INSERT INTO datatype(linenum,testdata) VALUES(4,NULL);
sqlite> /*Demonstrating auto increment by placing null in primary key field.*/
sqlite> INSERT INTO datatype(linenum,testdata) VALUES(NULL,'Placing NULL in Primary Key
');
sqlite> SELECT linenum,testdata,typeof(testdata) FROM datatype;
```

```
linenum   testdata                       typeof(testdata)
----------  -----------------------------  ----------------
1         -34                            integer
2         This a text field              text
3         3.1415                         real
4                                        null
5         Placing NULL in Primary Key    text
```

In the select query the expression typeof (field_value) uses the typeof function to show how sqlite classes each data item in the column. Note the result on linenumber 5. NULL was entered as the value for the INTEGER PRIMARY KEY on the fifth insert statement and that SQLite automatically added one to the highest existing integer in the column. When a record is successfully added to a SQLite table, SQLite will auto increment an INTEGER PRIMARY KEY column if no value is provided for the column.

```
sqlite> /*Demonstrating the result of entering a non integer value to the primary key field. */
sqlite> INSERT INTO datatype(linenum,testdata) VALUES('This value does','not belong in the primary key');
```

```
SQL error: datatype mismatch
sqlite>
```

The result of this last statement is that the record was not added because an attempt was made to put a text string ("This value does") into an INTEGER PRIMARY KEY column.

Note that SQLite's PRIMARY KEY constraint does not prevent the entry of null values.
In instances where the primary key is an integer then SQLite will autoincrement to the next higher number in the key column. For non integer primary keys however, you must include the words "NOT NULL " after the column datatype in the CREATE TABLE statement for any column that is a primary key or part of Primary Key as is the case with a Composite Primary Key. This will ensure that each record in the table has a distinct identifier and help to avoid the dreaded Cartesian Product when you run queries using multiple tables.

Dates and Times

All dates and times are stored in a SQLite tables as text strings except for the Julian Date format which stores the number of days since November 24, 4714 BC as a floating point number. Dates are most often in the format of "YYYY-MM-DD". SQLite has a number of functions for manipulating and working with date strings

```
sqlite> SELECT date('now');
2011-01-21
sqlite>SELECT datetime('now');
2011-01-21 13:45:51
sqlite> SELECT datetime('now','localtime');
2011-01-21 08:47:22
sqlite>
```

Relating Records Between Tables

PRIMARY KEY

A primary key is a field or combination of fields that uniquely identifies each record in a table. It is generally a good practice declare a primary key field in a CREATE TABLE statement. Note that in the following table definition that no primary key field has been specified and that it is just a single column list of colors

```
sqlite> CREATE TABLE colorlist (color TEXT);
sqlite> INSERT INTO colorlist VALUES('red');
sqlite> INSERT INTO colorlist VALUES('yellow');
sqlite> INSERT INTO colorlist VALUES('blue');
```

In SQLite every table has a default PRIMARY KEY field called the "rowid" which is an integer that identifies the record within the table.

```
sqlite>.headers on
sqlite> SELECT rowid,color FROM colorlist;
rowid|color
1|red
2|yellow
3|blue
sqlite>
```

List mode, shown above is the default method of display in SQLite for a query result. Each field in a record is separated by a delimiter, most often a pipe " | " character.
TIP: You can change to the Column mode, which is often easier to read by entering ".mode columns" at the prompt as shown below.

```
sqlite> .mode columns
sqlite> SELECT rowid,color FROM colorlist;
 rowid       color
 ----------  ----------
 1           red
 2           yellow
 3           blue
sqlite>
```

Constraints

In the following example, two tables are used to demonstrate primary and foreign key constraints within SQLite. The "employee" table which lists employees and timecard which records the hours worked each week by each employee.

```
sqlite>CREATE TABLE employee(EmpIDparent INTEGER PRIMARY KEY,FirstName TEXT,LastName
TEXT,StartDate CHAR(10 ),EndDate CHAR(10),PayGrade CHAR(2) , PayRate Real);
sqlite>INSERT INTO employee (EmpIDparent,FirstName,LastName,StartDate,EndDate,PayGrade,PayRate)
VALUES(153,'Melvin','Roberts','2009-09-19',NULL,'L3',18.19);
sqlite>INSERT INTO employee (EmpIDparent,FirstName,LastName,StartDate,EndDate,PayGrade,PayRate)
VALUES(154,'Alan','Jones','2009-09-11',NULL,'L3',17.79);
```

```
sqlite> .mode columns
sqlite> .headers on
sqlite> SELECT * FROM employee;
```

```
 EmpIDparent  FirstName  LastName  StartDate  EndDate    PayGrade   PayRate
 -----------  ---------- ---------- ---------- ---------- ---------- ----------
 153          Melvin     Roberts   40075                 L3         18.19
 154          Alan       Jones     40067                 L3         17.79
sqlite>
```

The "timecard" table has what is commonly known as a Composite key, which is a primary key that uses two or more fields to uniquely identify each row in a table. In this particular instance there will be more than one record with the same employee id and certainly more than one record with the same week number but there should only be one record with the same combination of employee id and pay period number.

```
sqlite> CREATE TABLE timecard(PayPeriod INTEGER,Hours REAL,EmpIDchild INTEGER,PRIMARY KEY
(PayPeriod,EmpIDchild), FOREIGN KEY(EmpIDchild) REFERENCES employee(EmpIDparent));
sqlite> INSERT INTO timecard(PayPeriod,EmpIDchild,Hours) VALUES(1,153,38.5);
sqlite> INSERT INTO timecard(PayPeriod,EmpIDchild,Hours) VALUES(1,154,41.25);
```

The next record to be added has a combination of pay period and Employee id number that duplicates a record that has been previously entered.

```
sqlite> INSERT INTO timecard(PayPeriod,EmpIDchild,Hours) VALUES(1,153,34.5);
Error: columns PayPeriod, EmpIDchild are not unique
```

Now reinsert the record with the corrected pay period value of 2.

```
sqlite> INSERT INTO timecard(PayPeriod,EmpIDchild,Hours) VALUES(2,153,34.5);
sqlite> SELECT * FROM timecard;
```

```
 PayPeriod  Hours      EmpIDchild
 ---------- ---------- ----------
 1          38.5       153
 2          34.5       153
 1          41.25      154
```

Foreign Keys

A foreign key constraint specifies that for each record in the table there must be a unique record that matches the key in the linked table While most often related to the primary key of the parent table, it doesn't necessarily have to be that way. However the related column in the parent table must have a UNIQUE constraint otherwise a Cartesian Product will likely be the result.

Since version 3.6.19, SQLite has included the capability to enforce Foreign Key constraints but this functionality currently must be activated for each database session by entering PRAGMA foreign_keys = ON;.

In the following example, foreign key support has not been enabled for the database session. A record is entered in the timecard table having an employee id number which is not found in the employee table. The database engine happily accepts this orphan record, which may result in somebody not being paid for that week since their hours worked can't be related to their employee id.

```
sqlite> INSERT INTO timecard(PayPeriod,EmpIDchild,Hours) VALUES(2,150,40.5);
sqlite> SELECT * FROM timecard WHERE rowid = last_insert_rowid();
 PayPeriod   Hours       EmpIDchild
 ----------  ----------  ----------
 2           40.5        150
sqlite> DELETE FROM timecard WHERE rowid = last_insert_rowid();
```

Fortunately the error is spotted immediately and corrected by deleting the record by using the last_insert_rowid() function as criteria.

Foreign Keys is now enabled but the wrong value is still entered again.

```
sqlite> PRAGMA foreign_keys = ON;
sqlite> INSERT INTO timecard(PayPeriod,EmpIDchild,Hours) VALUES(2,150,40.5);
Error: foreign key constraint failed
```

A value for EmpIDchild is entered which corresponds to a record in the employee table.

```
sqlite> INSERT INTO timecard(PayPeriod,EmpIDchild,Hours) VALUES(2,154,40.5);
sqlite> SELECT * FROM timecard;
 PayPeriod   Hours       EmpIDchild
 ----------  ----------  ----------
 1           38.5        153
 2           34.5        153
 1           41.25       154
 2           40.5        154
sqlite>
```

Cascade

Assuming that foreign key support has been enabled for the session.
Foreign key constraints will prevent you from changing the key column in the parent table or deleting records in the parent table which have related records in the child table without first updating the child table. Appending ON DELETE CASCADE and ON UPDATE CASCADE will cause the SQLite Database Engine to make the necessary changes to the child table automatically.

ON DELETE CASCADE appended after the FOREIGN KEY definition will cause records in the Child table to be deleted when a matching parent id is deleted.
When ON UPDATE CASCADE is used, the foreign key field in the child table will be updated to match the record(s) in the parent table.

CREATE TABLE child_table_name (field_1 INTEGER PRIMARY KEY, field_2 TEXT, foreign_key_field INTEGER , FOREIGN KEY(foreign_key_field) REFERENCES parent_table_name(parent_key_field) ON DELETE CASCADE ON UPDATE CASCADE);

CHECK Constraints

A check constraint defines what is a valid value for a column.

```
field_name_1 REAL NOT NULL CHECK(field_name_1 >= 0)
field_name_2 TEXT NOT NULL CHECK(field_name_2 NOT IN ("string_1","string_2","string_3")
```

Working with SQLite TABLES

Add a column to an Existing Table

Sqlite will allow you to add a column or columns to an existing table using the "ALTER TABLE" statement, however the removal of a column from a table requires that the table be recreated and the data from the old table to be loaded into the revised table using a select query.

```
ALTER TABLE table_name ADD new_column_name data_type ;
```

```
sqlite> .schema inventorystat
CREATE TABLE inventorystat (naicscode INTEGER,year INTEGER,month
INTEGER,inventoryval DOUBLE);
sqlite>ALTER TABLE inventorystat ADD descrip VARCHAR(50);
sqlite> .schema inventorystat
CREATE TABLE inventorystat(naicscode INTEGER,year INTEGER,month
INTEGER,inventoryval DOUBLE,descrip VARCHAR(50));
sqlite>
```

In the example above we have entered ".schema inventorystat" to show the create statement for the table. The ALTER TABLE statement then adds the column "descrip" to the table specification which can be shown by entering ".schema inventorystat" once again.

Rename a Table

```
ALTER TABLE old_table_name RENAME TO new_table_name ;
```

```
sqlite> .tables
ReqDetail    ReqTotal            inventory    test1
ReqEquip     RequisitionDetail   reqdescrip   test2
sqlite> ALTER TABLE test2 RENAME TO backuplist;
sqlite> .tables
ReqDetail    ReqTotal            backuplist   reqdescrip
ReqEquip     RequisitionDetail   inventory    test1
sqlite>
```

In this example the table names "test2" is being renamed to "backuplist". The ".tables" statements show the list of the tables in the database before and after the change.

Deleting a table

Be very careful about using the "DROP TABLE" command. Once a table is gone, it is gone forever along with whatever data that it may have contained.

```
DROP TABLE table_name ;
DROP TABLE database_name.table_name ;
```

```
sqlite> .tables
CustomerBackup    Customers    asforum    catalogsales
sqlite> DROP TABLE CustomerBackup;
sqlite> .tables
Customers    asforum    catalogsales
sqlite>
```

Renaming or Dropping Columns in Table

Sqlite has only limited ALTER TABLE support. Operations involving dropping columns, renaming columns or a combination of the two require that a new table be created with the changes incorporated in it. The original table is then dropped or renamed and the new table is renamed to the original table name. One thing to be aware of is that once a table is dropped, any associated triggers will be lost. It is a good idea to run a query against the sqlite_master table for the table that you are working on and copy them into a text editor so that they can be easily be reentered by copying and pasting the CREATE TRIGGER statement(s) into the command line. Triggers may have to be edited if any column names referenced in the trigger(s) have been changed

```
SELECT sql FROM sqlite_master WHERE tbl_name = 'table_name';
```

```
sqlite> select sql from sqlite_master where tbl_name = 'ReqEquip';
sql
CREATE TABLE 'ReqEquip'(ReqNumber INTEGER PRIMARY KEY,Requestor VARCHAR(30)
NOT NULL,Auth VARCHAR(30) NOT NULL,ReqDate CHAR(10) NOT NULL)
CREATE TRIGGER ReqNumDel
BEFORE DELETE ON 'ReqEquip'
FOR EACH ROW BEGIN
DELETE from ReqDetail WHERE ReqDetail.ReqNumber = OLD.ReqNumber;
END

 sqlite>
```

Rename Columns in a Table

1. Compose a CREATE TABLE statement with a new table name that uses an "AS SELECT" clause followed by a comma separated list of the columns names in the table.
2. For those column names that you wish to change, follow the name with single space and then the new name enclosed in single quotes.
3. Run the sql statement and confirm that the new table is structured as you want it and that

the data from the old table has be loaded into the new table correctly.
4. Rename or drop the original table and rename the new table to the name of the original table.

```
CREATE TABLE temp_table_name AS SELECT old_field_name ' new_field_name',
list_other_fields FROM table_name;
ALTER TABLE table_name RENAME TO archive_old_table;
ALTER TABLE temp_table_name RENAME TO table_name;
```

```
sqlite> CREATE TABLE test1bk AS SELECT Requisition 'Req_num', Requestor,ReqDate
'Req_Date', Req_Total FROM test1; sqlite> ALTER TABLE test1 RENAME TO test1_old;
sqlite> ALTER TABLE test1bk RENAME TO test1;
sqlite> SELECT * FROM test1 limit 2;
Req_num|Requestor|Req_Date|Req_Total
1000|Carl Jones|2007/10/30|$ 24.12
1001|Peter Smith|2007/11/05|$ 13.51
sqlite>
```

Drop columns from a Table

1. Compose a CREATE TABLE statement with a new table name that uses a SELECT AS clause followed by a list of the columns or field names that you wish to retain.
2. Run the sql statement and confirm that the new table is structured as you want it.
3. Drop the original table or rename it and rename the new table to the name of the original table.

Making a copy of an existing table within the main database

From time to time you may want to duplicate an existing table in a database to test table changes without risking the original data or to make a snapshot of the data for backup purposes. Additionally, if you wish to change column names or remove one or more columns from a sqlite table then it will be necessary to create a table with the desired changes and to copy the data using a sql query before renaming the original and the new tables.

```
CREATE TABLE new_table_name AS SELECT * FROM original_table_name;
```

```
sqlite> CREATE TABLE CustomersCopy AS SELECT * FROM Customers;
sqlite> SELECT * FROM CustomersCopy;
130169|Acme Widgets|1744 Alder Road|Apt 31C|Springfield|VA|20171|Alan Allen|57551267
130208|Nike Missiles Inc|5946 Oak Drive||Springfield|VA|20171|Lucy Baker|57155762
130247|Charlies Bakery|7116 Ginko St|suite 100|Springfield|VA|20171|Susan Nordrom|5715552363
130286|Unisales Inc.|8438 Maple Ave||Springfield|VA|20171-3521|Roger Norton|57551418
130325|M.I. Sinform & Sons|1785 Elm Avenue|P.O. Box 31|Springfield|VA|20171|Mi I. Sinform|5715558760
130364|Big Dents Towing Inc.|7578 Spruce St.|Building 31 A|Springfield|VA|20175231|George Spencer|5715557855
130365|Weneverpay Inc|428 Holly Ct||Springfield|VA|20171|Peter Norton|57155543
sqlite>
```

If you wish only to copy the table structure without copying the records then add a limit of 0 to the end of the statement.

```
CREATE TABLE new_table_name AS SELECT * FROM original_table_name LIMIT 0;
```

SQLite

```
sqlite> CREATE TABLE CustomersEmpty AS SELECT * FROM Customers LIMIT 0;
sqlite> Select * from CustomersEmpty;
sqlite>
sqlite> .schema CustomersEmpty
CREATE TABLE CustomersEmpty(
AcctNumber INTEGER,
Custname VARCHAR(50),
Addr1 VARCHAR(50),
Addr2 VARCHAR(50),
City VARCHAR(30),
State CHAR(2),
Zipcode VARCHAR(10),
Contact VARCHAR(30),
Phone VARCHAR(10)
);
```

Select Queries

SELECT * FROM table_name;

SQL queries can be used for a myriad of tasks such as record searching, statistical analysis and making other complex calculations. On the previous page we used a a simple select query (shown above) to verify that the records were entered properly into the inventory table. What this query does is select all columns from the inventory table and lists every record in that table.

In SQLite- To specify that the column names are to be listed above the query results use the ".headers ON " on the command line, this only needs to be done once during the session unless the ".headers OFF " command was entered

```
sqlite> .headers ON
sqlite> SELECT * FROM inventory;
StockNumber|Descrip|OnHandQuan|PackQty|PackCost
23155|Shovel Pointed Long Handle|15|1|9.82
43111|White Gas Gallon Can|14|4|14.75
43512|10W-30 Motor Oil, Quart|36|12|18.2
43522|5W-30 Motor Oil, Quart|17|12|18.2
51001|AAA Dry Cells 4 Pack |92|12|9.0
51002|AA Dry Cells 4 Pack |173|12|9.0
51004|AA Dry Cells 8 Pack|5|12|16.8
51013|D Dry Cells 8 Pack|19|12|90.2
75149|Ball Point Pens Blue Fine tip, 12pack|92|20|15.37
sqlite>
```

First of all, we want to limit the columns in the query result to only those that we are interested in. We do this by replacing the asterisk in the previous select query with a list of fieldnames separated by commas.

```
sqlite> SELECT StockNumber,OnHandQuan,Descrip FROM inventory;
StockNumber|OnHandQuan|Descrip
23155|15|Shovel Pointed Long Handle
43111|14|White Gas Gallon Can
43512|36|10W-30 Motor Oil, Quart
43522|17|5W-30 Motor Oil, Quart
51001|92|AAA Dry Cells 4 Pack
51002|173|AA Dry Cells 4 Pack
51004|5|AA Dry Cells 8 Pack
51013|19|D Dry Cells 8 Pack
75149|92|Ball Point Pens Blue Fine tip, 12pack
sqlite>
```

Let us say that someone asks what Stock Number 75149 is. Of course in an actual business application, the number of records in a table could number in the thousands or even the millions making it time consuming and difficult to find the records that you want by visually searching the whole table. A better solution is to attach a WHERE clause to the query specifying that only records with the stock number equal to 75149 should be returned.

```
sqlite> SELECT StockNumber,OnHandQuan,Descrip FROM inventory WHERE StockNumber = 75149;
StockNumber|OnHandQuan|Descrip
75149|92|Ball Point Pens Blue Fine tip, 12pack
```

Using a LIKE Clause with Wildcards to Find Records

LIKE, NOT LIKE Conditions and Wildcards

The LIKE operator is used with one or more wildcard characters to select records based on a text string where it is not possible or it would too inconvenient to make an exact match. The wildcard characters used in SQLite are the percent sign (%), which matches zero or more characters and spaces and the underscore (_) which matches a single character or space. The asterisk used in the select query at the top of the page is also a wild card which selects all columns in a table.

In this next example we wish to find what types of motor oil that we stock but we don't know the stock numbers but we do know that the phrase "motor oil" will be in the description. Using the LIKE clause and the words "motor oil" sandwiched between percent signs and single quotes we get the following results.

```
sqlite> SELECT StockNumber,OnHandQuan,Descrip FROM inventory WHERE Descrip LIKE '%motor oil%';
StockNumber|OnHandQuan|Descrip
43512|36|10W-30 Motor Oil, Quart
43522|17|5W-30 Motor Oil, Quart
sqlite>
```

```
sqlite> SELECT brand,descrip,onhand_quan,on_order FROM product;
brand       descrip                    onhand_quan  on_order
----------  -------------------------  ------------ ----------
Shady Oak   Milk                       12           0
Cloverleaf  2 % Milk - Quart           31           12
Cloverleaf  1 % Milk - Quart           13           12
Cloverleaf  Skim Milk - Quart          42           0
Shady Oak   2 % Milk - Gallon          6            0
Acme        Soy Milk - Quart           0            48
Shady Oak   Whole Milk - Vitamin D     2            20
sqlite>
```

In this example we wish to know how many quart containers of milk that we have on hand.

```
sqlite>SELECT brand,descrip,onhand_quan,on_order FROM product WHERE descrip LIKE '%Milk - Quart';
brand         descrip               onhand_quan   on_order
------------  --------------------  ------------  ------------
Cloverleaf    2 % Milk - Quart      31            12
Cloverleaf    1 % Milk - Quart      13            12
Cloverleaf    Skim Milk - Quart     42            0
Acme          Soy Milk - Quart      0             48
sqlite>
```

If we wish to exclude Soy milk from the results then we can use an "AND" clause and a "NOT LIKE" condition.

```
sqlite> SELECT brand,descrip,onhand_quan,on_order FROM product WHERE descrip LIKE '%Milk - Quart' AND descrip NOT LIKE 'Soy%';
brand         descrip               onhand_quan   on_order
------------  --------------------  ------------  ------------
Cloverleaf    2 % Milk - Quart      31            12
Cloverleaf    1 % Milk - Quart      13            12
Cloverleaf    Skim Milk - Quart     42            0
sqlite>
```

ESCAPE characters

Here we wish to find out how many containers of 2% milk we have on hand and on order. To prevent SQLite from mistaking the percent sign in the description for a wildcard, we proceed it with an escape character which we declare at the end of the statement with the following syntax.

ESCAPE '\ '

The escape character does not necessarily have to be a backslash, it can be any single character you choose.

```
sqlite> SELECT brand,descrip,onhand_quan,on_order FROM product WHERE descrip LIKE '2 \% milk%' ESCAPE '\';
brand        descrip                     onhand_quan   on_order
----------   ------------------------    ------------  ----------
Cloverleaf   2 % Milk - Quart            31            12
Shady Oak    2 % Milk - Gallon           6             0
```

Here we have a table listing book titles and their authors.

```
sqlite> .width 20 40
sqlite> SELECT * FROM booklist;
author                title
--------------------  ----------------------------------------
John Doe              Metalurgy and Casting
John C. Doe           Probability and Error Estimation
John Charles Doe      Statistics - Tools for Decision Making
Joanna Carla Doe      Business Statistics and Applications
Mike J. Doe           Methods of Error Estimation
John Albert Doe       Methods of Analytic Chemistry
```

We want to select a list of books written by John Charles Doe. Since the Author may be listed with a full middle name or only a middle initial we use the following syntax.

LIKE 'john_c%doe'

Notice the underscore character between john and c which is another wildcard representing a single character or number.

```
sqlite> SELECT author, title FROM booklist WHERE author LIKE 'john_c%doe';
author               title
--------------------  --------------------------------------
John C. Doe          Probability and Error Estimation
John Charles         Statistics - Tools for Decision
Doe                  Making
sqlite>
```

USING "IN" to SELECT Records Based on A List of Values

Another way to use the WHERE clause is to use it with the word IN and a comma separated list of values as in the example below.

```
sqlite> SELECT InvoiceNo,ItemNo,Quan FROM catalogsales WHERE ItemNo IN(4501,4502)
ORDER BY InvoiceNo,ItemNo DESC;
InvoiceNo        ItemNo           Quan
---------------  ---------------  ---------------
21001            4502             10
21001            4501             10
21015            4502             10
21015            4501             3
21023            4502             27
21023            4501             10
21027            4501             10
sqlite>
```

Notice also the ORDER BY clause and the list of column names that follows the clause. This will define the sort order of the results.

The ORDER BY Clause

ORDER BY column name DESC - sort descending, highest to lowest
ORDER BY column name ASC - sort ascending,lowest to highest. The addition of "ASC" after the column name to sort by is generally unnecessary since the ascending sort order is the default.
Rows will be sorted in the order in which they are listed after the "ORDER BY " clause in the example above the records are first sorted by invoice number and then item number because we wish to keep the rows for each invoice together.

Queries with Calculated Fields in SQLite

Another useful thing that can be done with select queries are mathematical calculations with or without using data from your database tables. The example below could have probably have been done more easily on a calculator but it demonstrates the principle.

```
sqlite> select ((12 *12)/(17-11)+1);
((12 *12)/(17-11)+1)
25
sqlite>
```

Notice that the header above the result is the equation itself. By default the headers will be the column name or in the case of a calculated field the equation. A more descriptive term can replace that header by following the column name or equation by one or more spaces and the desired text for the header in single quotes

```
sqlite> select ((12 *12)/(17-11)+1) 'Math Result';
Math Result
25
sqlite>
```

In this example we want to calculate the dollar value of the items listed in the inventory table. In order to do this it will be necessary to multiply the price per item by the quantity on hand of that item. If you look carefully at the inventory table you will see that only the pack cost is listed not the per unit cost. To calculate the unit cost, the Pack Cost must be divided by the Pack Quantity. The value of each list item can then be calulated by multiplying the unit cost by the quantity on hand. Notice that the ROUND function is used to round the results to two decimal places. The format for this function is

ROUND(Equation or Numeric Field ,Number of decimal places)

```
sqlite> SELECT OnHandQuan 'Quantity',Descrip 'Description', ROUND(PackCost/PackQty,2)
'Unit Price',ROUND((PackCost/PackQty) * OnHandQuan,2) 'Ext' FROM inventory;
Quantity|Description|Unit Price|Ext
15|Shovel Pointed Long Handle|9.82|147.3
14|White Gas Gallon Can|3.69|51.63
36|10W-30 Motor Oil, Quart|1.52|54.6
17|5W-30 Motor Oil, Quart|1.52|25.78
92|AAA Dry Cells 4 Pack |0.75|69.0
173|AA Dry Cells 4 Pack |0.75|129.75
5|AA Dry Cells 8 Pack|1.4|7.0
19|D Dry Cells 8 Pack|7.52|142.82
92|Ball Point Pens Blue Fine tip, 12pack|0.77|70.7
sqlite>
```

The results look a little jumbled and are hard to read. The presentation of the query results can be improved by the following commands. In SQLite, query results can be produced in 8 different formats using the " .mode". The default is list mode in which each field is separated by a specified character. The default character is the pipe "|". This example uses the dot command ".mode column". The ".width" specifies the space allocated to each field, the default is 10 spaces per field.

```
.mode column
.width 10 30 10 10
```

```
sqlite>.mode column
sqlite>.width 10 30 10 10
sqlite> SELECT OnHandQuan 'Quantity',Descrip 'Description', ROUND(PackCost/PackQty,2)
'Unit Price',ROUND((PackCost/PackQty) * OnHandQuan,2) 'Ext' FROM inventory;
Quantity    Description                   Unit Price  Ext
----------  --------------------------    ----------  ----------
15          Shovel Pointed Long           9.82        147.3
            Handle
14          White Gas Gallon Can          3.69        51.63
36          10W-30 Motor Oil, Quart       1.52        54.6
17          5W-30 Motor Oil, Quart        1.52        25.78
92          AAA Dry Cells 4 Pack          0.75        69.0
173         AA Dry Cells 4 Pack           0.75        129.75
5           AA Dry Cells 8 Pack           1.4         7.0
19          D Dry Cells 8 Pack            7.52        142.82
92          Ball Point Pens Blue Fine     0.77        70.7
```

What if we want to know the total value of the inventory? We could of course add up the values in the Ext column but that could take awhile particularly if there were hundreds of different items in the inventory. A better way would be to use the SUM aggregate function to add all those values for us.

```
sqlite> SELECT SUM((PackCost/PackQty) * OnHandQuan) 'Total Value of Inventory' FROM
inventory;
Total Value of Inventory
------------------------
698.577
sqlite>
```

SQLite does not have a currency format for output but we can simulate it by using pipes "||" to append a dollar sign to the result after it has been rounded to two decimal places.

```
sqlite> SELECT '$ ' || ROUND(SUM((PackCost/PackQty) * OnHandQuan),2) 'Total Value of
Inventory' FROM inventory;
Total Value of Inventory
------------------------
$ 698.58
sqlite>
```

The Use of CAST for the Division of INTEGER Values

It should be mentioned that caution should be used when dividing one integer value with another in SQLite. In the example below we are dividing 5 by 3 which is about 1.6666666666667 however if SQLite sees two integer numbers it will round down to the nearest integer.

```
sqlite> SELECT 5/3;
1
sqlite>
```

A work around for this is to either add a decimal point to one of the numbers in the calculation or to use the CAST(number or column name AS REAL) function to change the type to a floating point number.

```
sqlite> SELECT 5.0/3;
1.66666666666667
sqlite> SELECT CAST(5 AS REAL)/3;
1.66666666666667
sqlite>
```

In the next demonstration we have a table called "tbl_int" with two columns. One is declared to have INTEGER values (MyInt) and the other REAL values(MyDec).

```
CREATE TABLE tbl_int(MyInt INTEGER,MyDec REAL);
INSERT INTO tbl_int VALUES(1,1.5);
INSERT INTO tbl_int VALUES(3,3.7);
INSERT INTO tbl_int VALUES(499,499);
INSERT INTO tbl_int VALUES(43,42.999);
INSERT INTO tbl_int VALUES(17,17);
```

If we execute a SELECT query in which the values in each column are divided by two and then multiplied by two we can see that the result of the "INT" column is one less compared to the original INTEGER value (MyInt).

```
sqlite> SELECT MyInt, (MyInt/2)*2 AS 'INT', MyDec, (MyDec/2)*2 AS 'DEC' FROM tbl_int;

MyInt       INT         MyDec       DEC
----------  ----------  ----------  ----------
1           0           1.5         1.5
3           2           3.7         3.7
499         498         499.0       499.0
43          42          42.999      42.999
17          16          17.0        17.0
```

Queries using Multiple Tables

Most useful queries in a relational database require the use of two or more tables. For the purposes of this next exercise we are going to create two tables to use in conjunction with the inventory table. The Requisition table (RecEquip) and the Requisition Detail table (ReqDetail) which is a child table for the ReqEquip table to store the details for requisition request. For every record in ReqEquip there will be one or more records in the ReqDetail table. The ReqNumber column will be the common key for the two tables.

```
sqlite> CREATE TABLE requisition(ReqNumber INTEGER PRIMARY KEY,Requestor VARCHAR(30) NOT
NULL,Auth VARCHAR(30) NOT NULL,ReqDate CHAR(10) NOT NULL);
sqlite> INSERT INTO requisition(ReqNumber,Requestor,Auth,ReqDate) VALUES (1000,'Carl Jones','A. Robinson
Mgr','2007/10/30');
sqlite> INSERT INTO requisition(ReqNumber,Requestor,Auth,ReqDate) VALUES (1001,'Peter Smith','A. Robinson
Mgr','2007/11/05');
sqlite> INSERT INTO requisition(ReqNumber,Requestor,Auth,ReqDate) VALUES (1002,'Carl Jones','A. Robinson
Mgr','2007/11/06');
sqlite>
```

Renaming Tables

Using the ALTER TABLE command, let's rename the table to "ReqEquip" and verify it with the dot command ".tables".

```
sqlite> ALTER TABLE requisition RENAME TO ReqEquip;
sqlite> .tables
ReqEquip inventory
sqlite>
```

```
sqlite> CREATE TABLE ReqDetail(ReqNumber INTEGER,StockNumber INTEGER,Quantity
INTEGER,ItemCost REAL);
sqlite> INSERT INTO ReqDetail(ReqNumber,StockNumber,Quantity,ItemCost)
VALUES(1000,51013,2,7.52);
sqlite> INSERT INTO ReqDetail(ReqNumber,StockNumber,Quantity,ItemCost)
VALUES(1000,51002,4,.75);
sqlite> INSERT INTO ReqDetail(ReqNumber,StockNumber,Quantity,ItemCost)
VALUES(1000,43512,4,1.52);
sqlite> INSERT INTO ReqDetail(ReqNumber,StockNumber,Quantity,ItemCost)
VALUES(1001,23155,1,9.82);
sqlite> INSERT INTO ReqDetail(ReqNumber,StockNumber,Quantity,ItemCost)
VALUES(1001,43111,1,3.69);
sqlite> INSERT INTO ReqDetail(ReqNumber,StockNumber,Quantity,ItemCost)
VALUES(1002,51001,1,.75);
sqlite> INSERT INTO ReqDetail(ReqNumber,StockNumber,Quantity,ItemCost)
VALUES(1002,23155,1,9.82);
sqlite>
```

Dot Notation for Table-Column Names.

Since the primary key of ReqEquip and the foreign key of ReqDetail are both called "ReqNumber" it will be necessary to be explicit regarding the table from which the column is coming from. This can be done by using the notation "table name.column name"

```
sqlite> SELECT
ReqEquip.ReqNumber,ReqEquip.Requestor,ReqDetail.Quantity,ReqDetail.StockNumber FROM
ReqEquip,ReqDetail WHERE ReqEquip.ReqNumber = ReqDetail.ReqNumber;
ReqNumbe                         StockNumbe
r          Requestor    Quantity r
---------- ------------ -------- ------------
1000       Carl Jones   2        51013
1000       Carl Jones   4        51002
1000       Carl Jones   4        43512
           Peter
1001       Smith        1        23155
           Peter
1001       Smith        1        43111
1002       Carl Jones   1        51001
1002       Carl Jones   1        23155
sqlite>
```

As it happens we are only interested in items on Requisition number 1000. Using an AND clause we can exclude the other records.

```
sqlite> SELECT
ReqEquip.ReqNumber,ReqEquip.Requestor,ReqDetail.Quantity,ReqDetail.StockNumber FROM
ReqEquip,ReqDetail WHERE ReqEquip.ReqNumber = ReqDetail.ReqNumber AND
ReqEquip.ReqNumber = 1000;
ReqNumber  Requestor    Quantity StockNumber
---------- ------------ -------- ------------
1000       Carl Jones   2        51013
1000       Carl Jones   4        51002
1000       Carl Jones   4        43512
```

Adjust the width of the fields to make a clearer presentation. By adding the inventory table we are able to obtain the description of the items requisitioned.

```
sqlite> .width 10 12 8 8 20 10
sqlite> SELECT ReqEquip.ReqNumber, ReqEquip.Requestor, ReqDetail.StockNumber,
ReqDetail.Quantity ,inventory.Descrip ,ReqDetail.ItemCost FROM ReqEquip ,ReqDetail,
inventory WHERE ReqEquip.ReqNumber = ReqDetail.ReqNumber AND ReqEquip.ReqNumber
= 1000 AND inventory.StockNumber = ReqDetail.StockNumber;
ReqNumber  Requestor    StockNum Quantity Descrip              ItemCost
```

```
----------   ----------   ------------   --------   ----------------------   ----------
1000         Carl Jones   51013          2          D Dry Cells 8 Pack       7.52
1000         Carl Jones   51002          4          AA Dry Cells 4 Pack      0.75
1000         Carl Jones   43512          4          10W-30 Motor Oil, Qu     1.52
sqlite>
```

Using an alias for a table name in a Query

When writing complex queries with dot notation it is sometimes helpful to reference each table with a short alias particularly if the table names are long and similar in spelling.

SELECT column_1,column_2 FROM table_name_1 AS alias_1, table_name_2 AS alias_2 ;

```
sqlite> SELECT a.ReqNumber, a.Requestor, b.StockNumber, b.Quantity ,c.Descrip , b.ItemCost
FROM ReqEquip AS a ,ReqDetail AS b , inventory AS c WHERE a.ReqNumber = b.ReqNumber
AND a.ReqNumber = 1000 AND c.StockNumber = b.StockNumber;
ReqNumber    Requestor    StockNum       Quantity   Descrip                  ItemCost
----------   ----------   ------------   --------   ----------------------   ----------
1000         Carl Jones   51013          2          D Dry Cells 8 Pack       7.52
1000         Carl Jones   51002          4          AA Dry Cells 4 Pack      0.75
1000         Carl Jones   43512          4          10W-30 Motor Oil, Qu     1.52
sqlite>
```

Cartesian Products

Cartesian Product - What happens if tables in a select query are not related properly.
The following example has two tables. One, the "managerlist" table listing store managers, their assigned store (store_assn) along with other information about the employee and the "storelist" table listing information about the store's location with the primary key, "store_number" which relates to the to "store_assn"" column in the "managerlist". The objective is to produce a list of Managers and the City and State where their assigned store is located.

```
sqlite> SELECT employee_id,last_name,first_name,start_date,store_assn FROM managerlist;
employee_id   last_name          first_name         start_date   store_assn
------------  ----------------   ----------------   ----------   ----------
456           Walters            Joanna             2006-06-14   23
532           Niels              Matthew            2007-02-09   35
637           Simpson            Robert             2008-01-21   27
sqlite> SELECT store_number,city,state FROM storelist;
store_number   city               state
------------   ----------------   ----------------
23             Bowie              MD
27             Scranton           PA
35             Allentown          PA
sqlite>
```

When dealing with two or more tables it is very important to join or relate the tables properly, if you fail to do so you will likely create what is known as a cartesian product as shown below. Notice that the records in each table have been the matched against each other resulting in nine rows. Two thirds of the result list an incorrect location for specified store number.

```
sqlite> .width 20 10 12 20 sqlite> SELECT (last_name||', '||first_name) AS 'Manager', employee_id AS 'Manager ID', store_assn AS 'Store Number', city||', '||state AS 'Location' FROM managerlist, storelist;
```

```
Manager              Manager ID  Store Number  Location
--------------------  ----------  ------------  --------------------
Walters, Joanna      456         23            Bowie, MD
Walters, Joanna      456         23            Scranton, PA
Walters, Joanna      456         23            Allentown, PA
Niels, Matthew       532         35            Bowie, MD
Niels, Matthew       532         35            Scranton, PA
Niels, Matthew       532         35            Allentown, PA
Simpson, Robert      637         27            Bowie, MD
Simpson, Robert      637         27            Scranton, PA
Simpson, Robert      637         27            Allentown, PA
sqlite>
```

The following is the correct query with a proper WHERE clause.

```
sqlite> SELECT (last_name||', '||first_name) AS 'Manager', employee_id AS 'Manager ID', store_assn AS 'Store Number', city||', '||state AS 'Location' FROM managerlist, storelist WHERE store_assn = store_number;
```

```
Manager              Manager ID  Store Number  Location
--------------------  ----------  ------------  --------------------
Walters, Joanna      456         23            Bowie, MD
Niels, Matthew       532         35            Allentown, PA
Simpson, Robert      637         27            Scranton, PA
```

INNER and OUTER JOIN QUERIES

Here we have created three tables, one a list of customers and their Account numbers called oddly enough, "Customers" A table called "cust_invoice" listing invoices and using the customer account number as a foreign key with the "Customers" table. "catalogsales" which is a detail table for "cust_invoice" listing the individual items ordered by our customers on each invoice using the invoice number "InvoiceNo" as a foreign key.

Let us say that we want a list of customers that have ordered from us in the past. To find this out we can relate the table listing the customer accounts with the table listing the invoices by matching the primary key field "AcctNumber" in the Customers table with the foreign key field "AcctNumber" in the cust_invoice table

```
sqlite> SELECT Customers.AcctNumber, Customers.Custname FROM Customers, cust_invoice
WHERE Customers.AcctNumber = cust_invoice.AcctNumber;

AcctNumber    Custname
------------  -------------------------
130208        Nike Missiles Inc
130286        Unisales Inc.
130247        Charlies Bakery
130286        Unisales Inc.
sqlite>
```

SELECT DISTINCT

Notice that since Unisales Inc. has two invoice numbers associated with the same account number in the "cust_invoice" table that they are listed in the results twice. If they had 50 invoice numbers under the same account number then they would be listed 50 times. However we are only interested in just a list of customers that have ordered from us. Using the "SELECT DISTINCT" clause as shown below, enables us to eliminate the duplicate rows.

```
sqlite> SELECT DISTINCT Customers.AcctNumber, Customers.Custname FROM Customers,
cust_invoice WHERE Customers.AcctNumber = cust_invoice.AcctNumber;

AcctNumber    Custname
------------  -------------------------
130208        Nike Missiles Inc
130247        Charlies Bakery
130286        Unisales Inc.
sqlite>
```

Inner Join

The above query by the way is known as in Inner Join query where the only rows returned are ones in which both tables have fields that match the stated criteria. The following code will achieve the same result.

SELECT DISTINCT Customers.AcctNumber, Customers. Custname FROM Customers INNER JOIN cust_invoice ON Customers.AcctNumber = cust_invoice.AcctNumber;

NATURAL JOIN

A NATURAL JOIN will also work in the above example since the primary key and the foreign key in the two tables have the same name. Be very careful in using Natural Join queries in the absence of properly matched columns, a cartesian product will be produced.

SELECT DISTINCT Customers .AcctNumber , Customers .Custname FROM Customers NATURAL JOIN cust_invoice ;

Using Aggregate Functions

sqlite> SELECT Customers.AcctNumber,Customers.Custname ,catalogsales.InvoiceNo,ItemNo,Price,Quan ,(Price*Quan) 'EXT' FROM Customers ,catalogsales WHERE Customers.AcctNumber= catalogsales.AcctNumber ;

```
AcctNumber  Custname            InvoiceNo   ItemNo    Price     Quan    EXT
----------  ------------------  ----------  --------  --------  ------  --------
130208      Nike Missiles Inc   21001       4501      13.53     10      135.3
130208      Nike Missiles Inc   21001       5700      24.95     12      299.4
130208      Nike Missiles Inc   21001       4437      6.53      4       26.12
130208      Nike Missiles Inc   21001       4551      13.53     10      135.3
130208      Nike Missiles Inc   21001       4502      17.95     10      179.5
130286      Unisales Inc.       21027       4501      13.53     10      135.3
130286      Unisales Inc.       21027       5700      24.95     17      424.15
130286      Unisales Inc.       21027       4437      6.53      25      163.25
130286      Unisales Inc.       21027       3570      291.32    2       582.64
130286      Unisales Inc.       21015       4501      13.53     3       40.59
130286      Unisales Inc.       21015       4502      17.95     10      179.5
130286      Unisales Inc.       21015       5390      1499.99   1       1499.99
130247      Charlies Bakery     21023       4502      17.95     27      484.65
130247      Charlies Bakery     21023       4501      13.53     10      135.3
130247      Charlies Bakery     21023       5700      24.95     7       174.65
130247      Charlies Bakery     21023       4437      6.53      15      97.95
```

GROUP BY

Let us say that we want to know what each customer spent. Well we could try to use the SUM aggregate function as we did to calculate the total value of the inventory.

sqlite> SELECT Customers.AcctNumber,Customers.Custname ,catalogsales.InvoiceNo , SUM(Price*Quan) 'TOTAL' FROM Customers ,catalogsales WHERE Customers.AcctNumber= catalogsales.AcctNumber ;

```
Customers.AcctN    Customers.Custname    catalogsales.In    TOTAL
---------------    --------------------  ---------------    ----------
130247             Charlies Bakery       21023              4693.59
sqlite>
```

This doesn't look right does it. All the line items were added together and incorrectly attributed to one customer.

If you want a query with an aggregate function to deliver multiple rows such as by invoice number or account number then you must use the GROUP BY clause and the appropriate column to group on. In the example below we are grouping by invoice number (InvoiceNo), since there are 4 distinct invoice numbers there are four line items in the query result

```
sqlite> SELECT Customers.AcctNumber,Customers.Custname ,catalogsales.InvoiceNo
,SUM(Price*Quan) 'TOTAL' FROM Customers ,catalogsales WHERE Customers.AcctNumber=
catalogsales.AcctNumber GROUP BY InvoiceNo;
```

```
Customers.  Customers.Custname              catalogsales.InvoiceNo    TOTAL
----------  ------------------------------  ----------------------    ----------
130208      Nike Missiles Inc               21001                     775.62
130286      Unisales Inc.                   21015                     1720.08
130247      Charlies Bakery                 21023                     892.55
130286      Unisales Inc.                   21027                     1305.34
sqlite>
```

If we group by Account Number then we get a slightly different result. The invoice charges for each account will be added together yielding three rows since there were three customers that were invoiced.

```
sqlite> SELECT
Customers.AcctNumber,Customers.Custname,catalogsales.InvoiceNo,SUM(Price*Quan)
'TOTAL' FROM Customers,catalogsales WHERE
Customers.AcctNumber=catalogsales.AcctNumber GROUP BY catalogsales.AcctNumber;
```

```
Customers.  Customers.Custname              catalogsal  TOTAL
----------  ------------------------------  ----------  ----------
130208      Nike Missiles Inc               21001       775.62
130247      Charlies Bakery                 21023       892.55
130286      Unisales Inc.                   21015       3025.42
```

Left Outer Join Select Query

A left outer join returns all the records from the table on the left side of the JOIN clause and only those records from the table on the right that match the specified criteria.

The objective of the following example is to view a list of all customers regardless of whether or not they have ordered from us and if they have, then return the dollar value of those orders. To do this it will be necessary to use an Outer Join Query. The Query on the left is "Customers" and all records will be listed with records from catalogsales being listed only if there is a match.

```
sqlite> SELECT
Customers.AcctNumber,Customers.Custname,catalogsales.InvoiceNo,Price*Quan 'TOTAL'
FROM Customers LEFT OUTER JOIN catalogsales ON
Customers.Acctnumber=catalogsales.AcctNumber;
AcctNumber  Custname                        InvoiceNo   TOTAL
----------  ------------------------------  ----------  ----------
130169      Acme Widgets
130208      Nike Missiles Inc               21001       135.3
130208      Nike Missiles Inc               21001       299.4
130208      Nike Missiles Inc               21001       26.12
130208      Nike Missiles Inc               21001       135.3
130208      Nike Missiles Inc               21001       179.5
130247      Charlies Bakery                 21023       484.65
130247      Charlies Bakery                 21023       135.3
130247      Charlies Bakery                 21023       174.65
130247      Charlies Bakery                 21023       97.95
130286      Unisales Inc.                   21027       135.3
130286      Unisales Inc.                   21027       424.15
130286      Unisales Inc.                   21027       163.25
130286      Unisales Inc.                   21027       582.64
130286      Unisales Inc.                   21015       40.59
130286      Unisales Inc.                   21015       179.5
130286      Unisales Inc.                   21015       1499.99
130325      M.I. Sinform & Sons
130364      Big Dents Towing Inc.
130365      Weneverpay Inc
sqlite>
```

We are not quite there yet but by using the SUM aggregate function and grouping by AcctNumber, we can achieve the desired result.

```
sqlite> .width 12 25 12
sqlite> SELECT Customers.AcctNumber AS 'Acct Number', Customers.Custname AS
'Company', SUM(Price*Quan) AS 'Invoice Amt' FROM Customers LEFT OUTER JOIN
catalogsales ON Customers.Acctnumber = catalogsales.AcctNumber GROUP BY
Customers.AcctNumber ORDER BY SUM(Price*Quan)DESC,Customers.Custname;
Acct Number   Company                    Invoice Amt
------------  -------------------------  ------------
130286        Unisales Inc.              3025.42
130247        Charlies Bakery            892.55
130208        Nike Missiles Inc          775.62
130169        Acme Widgets
130364        Big Dents Towing Inc.
130325        M.I. Sinform & Sons
```

```
 130365           Weneverpay Inc
sqlite>
```

Full Outer Join

A full outer join returns all the records from the tables being joined and matches them where it can based on the specified column(s).

There is currently no provision for the use of FULL OUTER JOIN in SQLite, however we can achieve the same functionality by using a UNION clause to tie together two LEFT OUTER JOIN queries that mirror each other.

```
SELECT * FROM table_name_1 LEFT OUTER JOIN table_name_2 ON id_1 = id_2 UNION
SELECT * FROM table_name_2 LEFT OUTER JOIN table_name_1 ON id_1 = id_2 ;
```

```
sqlite> SELECT id_2,field_2,id_1,field_1 FROM tbl_2 LEFT OUTER JOIN tbl_1 ON id_2=id_1
...> UNION
...> SELECT id_2,field_2,id_1,field_1 FROM tbl_1 LEFT OUTER JOIN tbl_2 ON id_1=id_2;
 id_2         field_2     id_1        field_1
 ----------   ----------  ----------  ----------
                          99
 100          alpha       100
 101          bravo       101
 102          charlie     102
 103          delta
```

UNION and UNION ALL

The use of the UNION clause allows the result sets of two or more SELECT queries to be combined. Used by itself UNION will eliminate duplicate rows and sort ascending based on first column values unless an ORDER BY statement is added to the last SELECT statement. UNION ALL will list each row returned by each SELECT statement.

```
sqlite> SELECT id_2,field_2,id_1,field_1 FROM tbl_2 LEFT OUTER JOIN tbl_1 ON id_2=id_1
...>UNION ALL
...>SELECT id_2,field_2,id_1,field_1 FROM tbl_1 LEFT OUTER JOIN tbl_2 ON id_1=id_2;
 id_2         field_2     id_1        field_1
 ----------   ----------  ----------  ----------
 100          alpha       100
 101          bravo       101
 102          charlie     102
 103          delta
                          99
 100          alpha       100
 101          bravo       101
 102          charlie     102
sqlite>
```

Sorting UNION Queries

```
sqlite> SELECT id_2,field_2,id_1,field_1 FROM tbl_2 LEFT OUTER JOIN tbl_1 ON id_2=id_1
...>UNION ALL
...> SELECT id_2,field_2,id_1,field_1 FROM tbl_1 LEFT OUTER JOIN tbl_2 ON id_1=id_2
...>ORDER BY id_1 DESC, field_2;
id_2        field_2     id_1        field_1
----------  ----------  ----------  ----------
102         charlie     102
102         charlie     102
101         bravo       101
101         bravo       101
100         alpha       100
100         alpha       100
                        99
103         delta
sqlite>
```

More about UNION ALL

UNIONS can also be used to compile data from multiple tables and format the output. In the following example an invoice is created using six SELECT statements each generating 4 columns all joined together by a UNION ALL clause.

```
sqlite> SELECT 'Requisition #: ' ,'Requestor: ','Authorization:','Req Date : ' UNION ALL
...>SELECT ReqNumber, Requestor, Auth,ReqDate FROM ReqEquip WHERE ReqNumber=
1004 UNION ALL
...>SELECT ' ',' ', ' ',' ' UNION ALL
...>SELECT 'Stock Number' , 'Quantity', 'Cost', 'Ext' UNION ALL
...>SELECT StockNumber, Quantity,ItemCost,Quantity * ItemCost AS 'EXT' FROM ReqDetail
...>WHERE ReqNumber= 1004 UNION ALL
...>SELECT ' ',' ','Total Amount : ' ,SUM(Quantity * ItemCost) AS 'EXT' FROM ReqDetail
...> WHERE ReqNumber= 1004;
Requisition #:   Requestor:    Authorization:   Req Date :
1004             Steve North   R. Perry Mgr     2007/12/02

Stock Number     Quantity      Cost             Ext
75150            1             0.75             0.75
51002            12            0.75             9
43111            2             3.7              7.4
51001            3             0.75             2.25
                               Total Amount :   19.4
sqlite>
```

Inserting Records from another table using a select query

In the following simple example we have a table called "contact1" which lists names and email addresses. To avoid sending duplicate emails to the same individual we have made the Email address the primary key.

```
CREATE TABLE contactlist1 (FirstName TEXT,LastName TEXT,Email TEXT NOT NULL
PRIMARY KEY);
```

```
sqlite> SELECT * FROM contactlist1;

FirstName   LastName    Email
----------  ----------  ----------------------
Peter       Nelson      pnelson@oldmail.fake
Alan        Reed        aj.reed@oldmail.fake
```

Right now there are only two records in the table but we also have a table called newcontacts from which we can add to our list.

```
sqlite> SELECT * FROM newcontacts;

FirstName   LastName    Email
----------  ----------  ----------------------
James       Doe         james.doe@mymail.fake
Roberta     Allen       r.allen@mymail.fake
George                  gpmillford@mymail.fake
Kim         Simpson     ka.simpson@mymail.fake
```

Rather than copying the data from the newcontacts table and handkeying it into contactlist1 table we can use a single insert statement with a select query.

```
INSERT INTO target_table(field1 ,field2 ,field3 ) SELECT field_a, field_b, field_c FROM
source_table;
```

```
sqlite> INSERT INTO contactlist1 (FirstName,LastName,Email) SELECT
FirstName,LastName,Email FROM newcontacts;
sqlite> SELECT * FROM contactlist1;

FirstName   LastName    Email
----------  ----------  ---------------------
Peter       Nelson      pnelson@oldmail.fake
Alan        Reed        aj.reed@oldmail.fake
James       Doe         james.doe@mymail.fake
Roberta     Allen       r.allen@mymail.fake
George                  gpmillford@mymail.fake
Kim         Simpson     ka.simpson@mymail.fake
```

You can see that 4 new records were added to the table above. Of course it is not always that easy. The following table specification for contactlist2 is identical to contactlist1 except that LastName and FirstName field may not be null.

```
CREATE TABLE contactlist2 (FirstName TEXT NOT NULL,LastName TEXT NOT NULL,Email
TEXT NOT NULL PRIMARY KEY);
```

Notice that the third record in the newcontacts table has a NULL value for the LastName field.

```
sqlite> SELECT * FROM contactlist2;
FirstName   LastName    Email
----------  ----------  ----------------------
Peter       Nelson      pnelson@oldmail.fake
Alan        Reed        aj.reed@oldmail.fake
sqlite> INSERT INTO contactlist2 (FirstName,LastName,Email)SELECT
FirstName,LastName,Email FROM newcontacts;
Error: contactlist2.LastName may not be NULL
sqlite> SELECT * FROM contactlist2;
FirstName   LastName    Email
----------  ----------  ----------------------
Peter       Nelson      pnelson@oldmail.fake
Alan        Reed        aj.reed@oldmail.fake
```

We can correct the record with the NULL value in the newcontacts table and rerun the INSERT statement.

```
sqlite> UPDATE newcontacts SET LastName="Millford" WHERE
Email="gpmillford@mymail.fake";
sqlite> INSERT INTO contactlist2 (FirstName,LastName,Email)SELECT
FirstName,LastName,Email FROM newcontacts;
sqlite> SELECT * FROM contactlist2;
FirstName   LastName    Email
----------  ----------  ----------------------
Peter       Nelson      pnelson@oldmail.fake
Alan        Reed        aj.reed@oldmail.fake
James       Doe         james.doe@mymail.fake
Roberta     Allen       r.allen@mymail.fake
George      Millford    gpmillford@mymail.fake
Kim         Simpson     ka.simpson@mymail.fake
sqlite>
```

Using multiple tables in an UPDATE Statement

Here we have two tables (tbl_1 and tbl_2) and we wish to update field_1 of tbl_1 with values from field_2 of tbl_2 where the key columns id_1 and id_2 match.

```
sqlite> SELECT * FROM tbl_1;
id_1        field_1
----------  ----------
99
100
```

```
101
102
sqlite> SELECT * FROM tbl_2;
id_2        field_2
----------  ----------
100         alpha
101         bravo
102         charlie
103         delta
sqlite> UPDATE tbl_1 SET field_1=(SELECT field_2 FROM tbl_2 WHERE id_2 = id_1) ; sqlite>
SELECT * FROM tbl_1;
id_1        field_1
----------  ----------
99
100         alpha
101         bravo
102         charlie
sqlite>
```

UPDATE multiple fields using Values from another table

```
sqlite> SELECT * FROM tbl_3;
idnum       city         state
----------  -----------  ----------
100         Springfield  MA
101         Washingto    DC
            n
102         Mobile       AL
sqlite> SELECT * FROM tbl_4;
idnum       city        state
----------  ----------  ----------
100
101
102
sqlite> UPDATE tbl_4 SET city=(SELECT tbl_3.city FROM tbl_3 WHERE tbl_4.idnum
=tbl_3.idnum), state = (SELECT tbl_3.state FROM tbl_3 WHERE tbl_4.idnum = tbl_3.idnum);
sqlite> SELECT * FROM tbl_4;
idnum       city         state
----------  -----------  ----------
```

```
100         Springfield  MA
101         Washington   DC
102         Mobile       AL
sqlite>
```

Working with strings in SQLite

SQLite has a number of functions for manipulating text strings. In this next example the ".import" command was used to load values into the "fish" table from a comma separated values file (*.csv).

```
sqlite>CREATE TABLE fish(common_name TEXT,latin_name TEXT);
sqlite>.separator ","
sqlite>.import fishlist.csv fish
sqlite>.separator "|"
sqlite> select common_name,latin_name from fish;
common_name|latin_name
'Brown Trout' |'Salmo trutta'
'American Shad' |'Alosa sapidissima'
'Black Bullhead' |'Ictalurus melas'
'Chain Pickerel' |'Esox niger'
'Muskellunge' |'Esox masquinongy'
'Walleye'|'Stizostedion vitreum'
```

trim(field_name) removes white space characters from both ends of the string.
replace(field_name,'old_string','new_string')

Unfortunately the values were enclosed in single quotes and the ".import" command included them as well as some unwanted white space as part of each string. We can fix that by using some of the string manipulation functions in SQLite. First, it is advisible to try the changes out with a select query to test the result, before doing any permanent changes to the data.
In the example below, the replace function which is used to remove the quotation marks is nested inside a trim function in order to remove the excess whitespace from outside of the quotation marks.

```
sqlite> SELECT trim(replace(common_name,'''','')) 'Common Name'
,trim(replace(latin_name,'''','')) 'Scientific Name' from fish;
Common Name|Scientific Name
Brown Trout|Salmo trutta
American Shad|Alosa sapidissima
Black Bullhead|Ictalurus melas
Chain Pickerel|Esox niger
Muskellunge|Esox masquinongy
Walleye|Stizostedion vitreum
```

Generally single quotes only are used to enclose string values but the single quote is the character that we wish to replace so this example requires that the single quote to be escaped with 4 single quotes together.

```
sqlite> UPDATE fish SET common_name=trim(replace(common_name,'''','')),latin_name
=trim(replace(latin_name,'''',''));
sqlite> select * from fish;
common_name|latin_name
```

```
Brown Trout|Salmo trutta
American Shad|Alosa sapidissima
Black Bullhead|Ictalurus melas
Chain Pickerel|Esox niger
Muskellunge|Esox masquinongy
Walleye|Stizostedion vitreum
sqlite>
```

Using the SUBSTR Function to return parts of a string.

```
SUBSTR(field_name,start_location)
SUBSTR(field_name,start_location,substring_length )
```

If the start location is a positive integer then the substring will begin x number of characters from the left of the string. If the start location is a negative integer then the substring will begin x number of characters from the right.

```
sqlite> SELECT SUBSTR('String Manipulation in SQLite',8,12);
Manipulation
sqlite> SELECT SUBSTR('String Manipulation in SQLite',-9,9);
in SQLite
sqlite> SELECT SUBSTR('String Manipulation in SQLite',-9,2);
in
sqlite> SELECT SUBSTR('String Manipulation in SQLite',8);
Manipulation in SQLite
sqlite> SELECT SUBSTR('String Manipulation in SQLite',-9);
in SQLite
sqlite>
```

Given a date string in the format of yyyy-mm-dd we can split it into month, day and year with the SUBSTR function

```
sqlite> SELECT ReqNumber,ReqDate FROM ReqEquip Limit 1;
ReqNumber   ReqDate
----------  ----------
1000        2007-10-30
sqlite>
```

```
sqlite> SELECT ReqNumber,SUBSTR(ReqDate,6,2) 'MONTH', SUBSTR(ReqDate,9,2)
'DAY',SUBSTR(ReqDate,1,4) 'YEAR' FROM ReqEquip;
ReqNumber   MONTH  DAY    YEAR
----------  -----  -----  -----
1000        10     30     2007
1001        11     5      2007
1002        11     6      2007
1003        12     1      2007
sqlite>
```

Concatenate Strings using the "||"

```
sqlite> SELECT ReqNumber, SUBSTR(ReqDate,6,2)||'-'||SUBSTR(ReqDate,9,2)||'-'||
SUBSTR(ReqDate,1,4) 'Requisition Date' FROM ReqEquip;
```

```
ReqNumber   Requisition Date
----------  --------------------
1000        10-30-2007
1001        11-05-2007
1002        11-06-2007
1003        12-01-2007
```

Date and Time Functions

Looking for a date range

```
SELECT field_1,field_2,date_field WHERE julianday(date_field) BETWEEN julianday('1998-01-
01') and julianday('2008-01-01');
```

Find the number of days between two dates.

```
sqlite> SELECT julianday('2008-07-03')- julianday('2008-06-20');
13.0
```

Find Calendar date at a specified interval of time

Plus or minus "days","months", "years"

```
sqlite> SELECT date('2008-07-03', '+90days');
2008-10-01
```

Reformat output of Dates with strftime function

%m - Month, %d - Day, %Y - Year

```
sqlite> SELECT strftime('%m/%d/%Y', '2008-02-02');
02/02/2008
```

Conditional Clauses Using SELECT CASE

```
CASE WHEN first conditional expression THEN column value
WHEN second conditional expression THEN column value
WHEN third conditional expression THEN column value
END

CASE WHEN conditional expression THEN column value
ELSE default column value
END
```

```
sqlite> SELECT state,city|| ', ' || CASE
...> WHEN state ='AL' THEN 'Alabama'
...> WHEN state='DC' THEN 'District of Columbia'
...> WHEN state ='DE' THEN 'Delaware'
...> WHEN state='VA' THEN 'Virginia'
...> WHEN state = 'MA' THEN 'Massachusett'
...> END AS 'full name' FROM tbl_3;
state|full name
VA|Fairfax, Virginia
MA | Springfield, Massachusett
DC | Washington, District of Columbia
AL | Mobile, Alabama
sqlite>
```

In the following example we want to list the number of hours worked each week by the employees and to calculate any overtime that they may have worked. The CASE expression uses the condition that if the number of hours worked in a week by a given employee is greater than 40 then subtract 40 from the hours worked to calculate the amount of overtime worked, otherwise list zero in the overtime column.

```
sqlite> SELECT tbl_hours.week_number 'Week #', tbl_employee.employee_id 'Id Number',
tbl_employee.first_name||' ' ||tbl_employee.last_name 'Name', hours,CASE WHEN (hours>40)
THEN hours-40 ELSE 0 END AS overtime FROM tbl_employee, tbl_Hours WHERE
tbl_employee.employee_id = tbl_hours.employee_id ORDER BY
tbl_hours.week_number,tbl_employee.employee_id;
```

```
WEEK #      ID NUMBER   NAME               HOURS       OVERTIME
----------  ----------  -----------------  ----------  ----------
1           50          JONATHAN SMITH     37.5        0
1           60          GERALD MARSHAL     40.25       0.25
2           50          JONATHAN SMITH     28.5        0
2           60          GERALD MARSHAL     40.25       0.25
3           50          JONATHAN SMITH     41.5        1.5
3           60          GERALD MARSHAL     41.5        1.5
```

```
4          50             JONATHAN SMITH  40      0
4          60             GERALD MARSHAL  47.75   7.75
5          50             JONATHAN SMITH  40      0
5          60             GERALD MARSHAL  40      0
```

The next example uses the CASE clause in an aggregate query to list the total overtime incurred by all employees for each week.

```
sqlite> SELECT tbl_hours.week_number 'Week #',SUM(hours) AS 'Payroll Hours' ,SUM(CASE
WHEN (hours>40) THEN hours-40
...> ELSE 0
...> END) AS 'Overtime Hours' FROM tbl_employee, tbl_Hours WHERE
tbl_employee.employee_id = tbl_hours.employee_id GROUP BY tbl_hours.week_number;
WEEK #     PAYROLL HOURS OVERTIME HOURS
---------- ------------- ------------
1          77.75         0.25
2          68.75         0.25
3          83            3
4          87.75         7.75
5          80            0
sqlite>
```

Reformat Date Strings from mm/dd/yyyy to the Standard SQLite Datestring Format of yyyy-mm-dd Using SELECT CASE

Here we have a table listing dates in the conventional American format of Month, Day, Year separated by "/". Our objective is to change it to "yyyy-mm-dd" format.

```
sqlite> .headers on
sqlite> .mode column
sqlite> .width 10 10 16 8
sqlite> SELECT order_num,order_date,order_stat,cust_id FROM order_list;
order_num  order_date  order_stat        cust_id
---------- ----------  ----------------  --------
101281     2/7/2008    Completed         650
101288     2/25/2008   Completed         453
101313     03/09/2008  Cancelled         219
101301     3/01/2008   Billing Pending   243
101316     06/9/2008   In Progress       650
101419     12/17/2008  Cancelled         219
sqlite>
```

Ideally the dates in the column would have two digits for the month, two for the day and 4 digits for the year, in which case it would be a simple matter of using the SUBSTR function to extract the elements and rearrange them as shown below.

```
SUBSTR(date_field,7,4)||'-'||SUBSTR(date_field,1,2)||'-'||SUBSTR(date_field ,4,2)
```

Unfortunately in the example above, some of the values in the date column have only one digit for the month and or day. Using SELECT CASE and the CAST(substring AS INTEGER) function makes it possible to extract the value of the month and the day.

```
(SELECT SUBSTR(date_field,-4,4)) ||'-'||(SELECT CASE WHEN CAST(SUBSTR(date_field,1,2)
AS INTEGER)>=10 THEN SUBSTR(date_field,1,2) ELSE '0'|| CAST(SUBSTR (date_field,1,2)
AS INTEGER) END ) ||'-'|| (SELECT CASE WHEN CAST(SUBSTR(date_field,-7,3) AS
INTEGER)>=10 THEN SUBSTR(date_field,-7,2) ELSE '0'|| CAST(SUBSTR(date_field,-6,2) AS
INTEGER) END) AS 'date_field'
```

```
sqlite> SELECT order_num,(SELECT SUBSTR(order_date,-4,4)) ||'-'||(SELECT CASE WHEN
CAST(SUBSTR(order_date,1,2) AS INTEGER)>=10 THEN SUBSTR(order_date,1,2) ELSE '0'||
CAST(SUBSTR (order_date,1,2) AS INTEGER) END ) ||'-'|| (SELECT CASE WHEN
CAST(SUBSTR(order_date,-7,3) AS INTEGER)>=10 THEN SUBSTR(order_date,-7,2) ELSE
'0'|| CAST(SUBSTR(order_date,-6,2) AS INTEGER) END) AS 'order_date', order_stat,cust_id
FROM order_list;
order_num   order_date  order_stat        cust_id
----------  ----------  ----------------  --------
101281      2008-02-07  Completed         650
101288      2008-02-25  Completed         453
101313      2008-03-09  Cancelled         219
101301      2008-03-01  Billing Pending   243
101316      2008-06-09  In Progress       650
101419      2008-12-17  Cancelled         219
sqlite>
```

```
UPDATE table_name SET date_field = (select substr(date_field,-4,4)) ||'-'|| (SELECT CASE
WHEN CAST(SUBSTR(date_field,1,2) AS INTEGER) >= 10 THEN SUBSTR(date_field,1,2)
ELSE '0'|| CAST(SUBSTR (date_field,1,2) AS INTEGER) END ) ||'-'|| (SELECT CASE WHEN
CAST(SUBSTR(date_field,-7,3) AS INTEGER)>=10 THEN SUBSTR(date_field,-7,2) ELSE '0'||
CAST(SUBSTR(date_field,-6,2) AS INTEGER) END);
```

NOTE: When doing a mass update in a table it is often a good idea to make a back up of the table before doing your changes.

```
sqlite> UPDATE order_list SET order_date = (SELECT SUBSTR(order_date,-4,4)) ||'-'||
(SELECT CASE WHEN CAST(SUBSTR(order_date,1,2) AS INTEGER)>=10 THEN
SUBSTR(order_date,1,2) ELSE '0'||CAST(SUBSTR (order_date,1,2) AS INTEGER) END )||'-'||
(SELECT CASE WHEN CAST(SUBSTR(order_date,-7,3) AS INTEGER)>=10 THEN
SUBSTR(order_date,-7,2) ELSE '0'|| CAST(SUBSTR(order_date,-6,2) AS INTEGER) END);
sqlite> SELECT order_num,order_date,order_stat,cust_id FROM order_list;;


order_num   order_date  order_stat        cust_id
----------  ----------  ----------------  --------
```

```
101281    2008-02-07   Completed        650
101288    2008-02-25   Completed        453
101313    2008-03-09   Cancelled        219
101301    2008-03-01   Billing Pending  243
101316    2008-06-09   In Progress      650
101419    2008-12-17   Cancelled        219
sqlite>
```

Cross Tab Query

It is sometimes useful to organize data by column in a cross tab query in order to compare subsets of data. In following example we have a table with 12 rows called "Sales2008" listing monthly sales and expenses. The record for each month is identified by an integer field called "period" with a value between 1 (January) and 12 (December). The objective here is to compare sales by quarter. This can be done by creating a CASE statement for each column which represents a three month period or quarter.

```
sqlite>.HEADERS ON
sqlite> .MODE COLUMN
sqlite> SELECT SUM (CASE WHEN period BETWEEN 1 AND 3 THEN sales_amount ELSE 0 END) AS '1st Qtr',
...> SUM (CASE WHEN period BETWEEN 4 AND 6 THEN sales_amount ELSE 0 END) AS '2nd Qtr',
...> SUM (CASE WHEN period BETWEEN 7 AND 9 THEN sales_amount ELSE 0 END) AS '3rd Qtr',
...> SUM (CASE WHEN period BETWEEN 10 AND 12 THEN sales_amount ELSE 0 END) AS '4th Qtr',
...>SUM(sales_amount) AS 'Totals for 2008' FROM Sales2008;

1st Qtr     2nd Qtr     3rd Qtr     4th Qtr     Totals for 2008
----------  ----------  ----------  ----------  ---------------
42410.46    36290.93    29841.67    25719.7     134262.76
```

Note that the result set is one row and that the chosen format is ".MODE COLUMN" with ".HEADERS ON"

The next example utilizes the structure of the above query joined by UNION ALL clauses to show quarterly sales, expenses and net profit. A row header has been added to the beginning of each SELECT statement

```
SELECT 'Gross Sales' AS ' ' , SUM (CASE WHEN period BETWEEN 1 AND 3 THEN sales_amount ELSE 0 END)
AS '1st Qtr',
SUM (CASE WHEN period BETWEEN 4 AND 6 THEN sales_amount ELSE 0 END) AS '2nd Qtr',
SUM (CASE WHEN period BETWEEN 7 AND 9 THEN sales_amount ELSE 0 END) AS '3rd Qtr',
SUM (CASE WHEN period BETWEEN 10 AND 12 THEN sales_amount ELSE 0 END) AS '4th Qtr',
SUM(sales_amount) AS 'Totals for 2008' FROM Sales2008
UNION ALL
SELECT 'Expenses' AS ' ',
SUM (CASE WHEN period BETWEEN 1 AND 3 THEN expenses ELSE 0 END) AS '1st Qtr',
SUM (CASE WHEN period BETWEEN 4 AND 6 THEN expenses ELSE 0 END) AS '2nd Qtr',
SUM (CASE WHEN period BETWEEN 7 AND 9 THEN expenses ELSE 0 END) AS '3rd Qtr',
SUM (CASE WHEN period BETWEEN 10 AND 12 THEN expenses ELSE 0 END) AS '4th Qtr',
SUM (expenses ) AS 'Totals for 2008' FROM Sales2008
UNION ALL
SELECT 'Net Profit' AS ' ',
SUM (CASE WHEN period BETWEEN 1 AND 3 THEN (sales_amount - expenses) ELSE 0 END) AS '1st Qtr',
SUM (CASE WHEN period BETWEEN 4 AND 6 THEN (sales_amount - expenses) ELSE 0 END) AS '2nd Qtr',
SUM (CASE WHEN period BETWEEN 7 AND 9 THEN (sales_amount - expenses) ELSE 0 END) AS '3rd Qtr',
SUM (CASE WHEN period BETWEEN 10 AND 12 THEN (sales_amount - expenses) ELSE 0 END)
AS '4th Qtr',
SUM(sales_amount- expenses) AS 'Totals for 2008' FROM Sales2008;
```

```
             1st Qtr     2nd Qtr     3rd Qtr     4th Qtr     Totals for 2008
-----------  ----------  ----------  ----------  ----------  ---------------
Gross Sales  42410.46    36290.93    29841.67    25719.7     134262.76
Expenses     36105.89    35809.12    33382.55    28798.77    134096.33
Net Profit   6304.57     481.81      -3540.88    -3079.07    166.42999999999
```

Doing something with a query result

Demonstrating different output methods.

SQLite is capable of formating it's output in 8 different ways using the .mode command. In HTML mode the rows returned are formatted as HTML table rows. The only thing lacking in the table markup is the <TABLE> tag at the start and the </TABLE> at the end.

```
sqlite> .mode html
sqlite> .headers ON
sqlite> select * from Customers limit 1;
<TR> <TH>AcctNumber</TH> <TH>Custname</TH> <TH>Addr1</TH> <TH>Addr2</TH>
<TH>City</TH> <TH>State</TH>
<TH>Zipcode</TH><TH>Contact</TH><TH>Phone</TH></TR>
<TR><TD>130169</TD> <TD>Acme Widgets</TD> <TD>1744 Alder Road</TD><TD>Apt
31C</TD> <TD>Springfield</TD><TD>VA</TD><TD>20171</TD><TD>Alan
Allen</TD><TD>5715551267</TD></TR>
sqlite>
```

In list mode values for each field in the record are delimited by pipes "| ", unless the delimiter is changed.

```
sqlite> .mode list
sqlite> select * from Customers limit 1;
AcctNumber|Custname|Addr1|Addr2|City|State|Zipcode|Contact|Phone
130169|Acme Widgets|1744 Alder Road|Apt 31C|Springfield|VA|20171|Alan Allen|5715 551267
```

using the .separator command you can change the delimiter to whatever character or characters that you desire.

```
sqlite> .separator ~
sqlite> select * from Customers limit 1;
AcctNumber~Custname~Addr1~Addr2~City~State~Zipcode~Contact~Phone
130169~Acme Widgets~1744 Alder Road~Apt 31C~Springfield~VA~20171~Alan Allen~5715
551267
sqlite> .separator _-_
sqlite> select * from Customers limit 1;
130169_-_Acme Widgets_-_1744 Alder Road_-_Apt 31C_-_Springfield_-_VA_-_20171_-_Alan
Allen_-_5715551267
sqlite>
```

In the Comma separated value or CSV mode, all non numeric values are enclosed in quotation marks and values are delimited by commas. This format is recognized by a number of database and spreadsheet programs.

```
sqlite>.mode csv
sqlite> select * from Customers limit 1;
AcctNumber,Custname,Addr1,Addr2,City,State,Zipcode,Contact,Phone
```

```
130169,"Acme Widgets","1744 Alder Road","Apt 31C",Springfield,VA,20171,"Alan All
en",5715551267
sqlite>
```

The column format is often the neatest in terms of display, but it often requires trial and error tweeking of the column width with the ".width" command to get it right.

```
sqlite> .mode column
sqlite> .width 12 20 12 5
sqlite> select AcctNumber,Custname,City,State from Customers limit 1;
AcctNumber    Custname              City          State
------------  --------------------  ------------  -----
130169        Acme Widgets          Springfield   VA
```

Output Query results to a text file in column format

Using the .output command, query results can be redirected into a separate text file instead of the screen. The .output stdout command needs to be used to output to the screen after you are finished.

```
sqlite> .headers ON
sqlite> .mode columns
sqlite> .width 12 30 10 12
sqlite> .output C:/Databases/Report_1.txt
sqlite> SELECT 'Page 1' '' ,'List of Customers and Invoices' 'Report 1 Test 2';
sqlite> SELECT Customers.AcctNumber 'Acct Number',Customers.Custname 'Company'
,catalogsales.InvoiceNo 'Invoice #' ,SUM(Price*Quan) 'Invoice Amt' FROM Customers LEFT
OUTER JOIN catalogsales ON Customers.AcctNumber = catalogsales.AcctNumber GROUP BY
Customers.AcctNumber ,catalogsales.InvoiceNo;
sqlite> .output stdout
```

If no path is specified with the file name , then the output file will be placed in the same directory as the SQLite program.

```
              Report 1 Test 2
------------  ------------------------------
Page 1        List of Customers and Invoices
Acct Number   Company                         Invoice #   Invoice Amt
------------  ------------------------------  ----------  ------------
130169        Acme Widgets
130208        Nike Missiles Inc               21001       775.62
130247        Charlies Bakery                 21023       892.55
130286        Unisales Inc.                   21015       1720.08
130286        Unisales Inc.                   21027       1305.34
130325        M.I. Sinform & Sons
130364        Big Dents Towing Inc.
130365        Weneverpay Inc
```

Note the statement which acts as a header for the report, SELECT 'Page 1' '' ,'List of Customers and Invoices' 'Report 1 Test 2'; .
In column mode the resulting rows are printed below the header. In this particular case the resulting row is
Page 1   List of Customers and Invoices
and the headers are the aliases that follow each field, empty quotes to create a blank space and the title "Report1 Test 2"

Aligning numeric values in a column

Concatenate a string of ten spaces in length with the rounded value of the column and use SUBSTR to to align the values in the column.

```
sqlite> SELECT amount AS 'raw value', (ROUND(amount,2)) AS 'RND value' , CASE WHEN
(LENGTH(ROUND(amount,2))) - (LENGTH(CAST(amount AS INTEGER)) ) =2 THEN
SUBSTR('          '||(ROUND(amount,2))||'0', -10,10) ELSE SUBSTR('          '||
(ROUND(amount,2 )),-10,10) END AS 'result' FROM CurrencyTest;
raw value   RND value   result
----------  ----------  ----------
1.0         1.0               1.00
1.1         1.1               1.10
1.021       1.02              1.02
1.01        1.01              1.01
100.2       100.2           100.20
25.257      25.26            25.26
0.586       0.59              0.59
299.9999    300             300.00
53.0        53.0             53.00
35000.12    35000.12      35000.12
sqlite>
```

Triggers

What is a trigger? In SQL, a trigger is a sql statement or series of sql statements that are executed automatically in response to a specified event such as the update of or creation or deletion of a table or record. Each trigger must have a name that is unique to the database. Triggers are deleted when the table that they are associated with is dropped or they can be deleted with a DROP TRIGGER statement. Once created, triggers cannot be modified, to make changes the trigger must be dropped and then recreated.

```
DROP TRIGGER trigger_name ;
```

CREATE TRIGGER

This trigger automatically updates the inventory table by subtracting the Quantity of items requisitioned from the OnHandQuan value when an insert statement adds a record to the ReqDetail table.

```
CREATE TRIGGER inventoryupdate AFTER INSERT ON ReqDetail BEGIN
UPDATE inventory SET OnHandQuan = (OnHandQuan - NEW.Quantity) WHERE
inventory.StockNumber = NEW.StockNumber;
END;
```

```
sqlite> select StockNumber,OnHandQuan,Descrip from inventory where StockNumber = 75149;

StockNumber|OnHandQuan|Descrip
75149|92|Ball Point Pens Blue Fine tip, 12pack
sqlite> CREATE TRIGGER inventoryupdate AFTER INSERT ON ReqDetail BEGIN
   ...> UPDATE inventory SET OnHandQuan = (OnHandQuan- NEW.Quantity)
   ...> WHERE inventory.StockNumber = NEW.StockNumber;
   ...> END;
sqlite> INSERT INTO ReqDetail(ReqNumber,StockNumber,Quantity,ItemCost)
VALUES(1003,75149,3,0.77);
sqlite> select StockNumber,OnHandQuan,Descrip from inventory where StockNumber =75149;
StockNumber|OnHandQuan|Descrip
75149|89|Ball Point Pens Blue Fine tip, 12pack
sqlite>
```

Using Triggers to Enforce Referential Integrity

```
CREATE TRIGGER trigger_name BEFORE INSERT ON child_table BEGIN
SELECT CASE
WHEN ((SELECT parent_table . primary_key FROM parent_table WHERE parent_table .
primary_key = NEW. foreign_key ) ISNULL)
THEN RAISE(ABORT, 'Error Message')
END;
END;
```

```
sqlite> CREATE TRIGGER ReqNumIn BEFORE INSERT ON ReqDetail BEGIN
   ...> SELECT CASE
   ...> WHEN ((SELECT ReqEquip.ReqNumber FROM ReqEquip WHERE
ReqEquip.ReqNumber= NEW.ReqNumber) ISNULL)
   ...> THEN RAISE(ABORT, 'This Requisition number does not exist in the ReqEquip table.')
   ...> END;
   ...> END;
sqlite> insert into ReqDetail(ReqNumber,StockNumber,Quantity)values(2000,51001,15);
SQL error: This Requisition number does not exist in the ReqEquip table.
sqlite>
```

```
CREATE TRIGGER trigger_name BEFORE UPDATE ON child_table FOR EACH ROW BEGIN
SELECT CASE
WHEN ((SELECT parent_table . primary_key FROM parent_table WHERE parent_table .
primary_key = NEW.foreign_key ) ISNULL)
THEN RAISE(ABORT, 'Error Message')
END;
END;
```

```
sqlite>CREATE TRIGGER ReqNumUp BEFORE UPDATE ON ReqDetail FOR EACH ROW
BEGIN
   ...>SELECT CASE
   ...> WHEN ((SELECT ReqEquip.ReqNumber FROM ReqEquip
   ...>WHERE ReqEquip.ReqNumber= NEW.ReqNumber) ISNULL)
   ...> THEN RAISE(ABORT, 'update on table ReqDetail violates foreign key')
   ...> END;
   ...>END;
sqlite>
```

Cascading Delete

Delete records from a child table when a record from the parent table is deleted

```
CREATE TRIGGER trigger_name
BEFORE DELETE ON parent_table
FOR EACH ROW BEGIN
DELETE FROM child_table WHERE child_table.foreign_key = OLD. primary_key ;
END;
```

```
sqlite> CREATE TRIGGER ReqNumDel
   ...> BEFORE DELETE ON ReqEquip
   ...> FOR EACH ROW BEGIN
   ...> DELETE from ReqDetail WHERE ReqDetail.ReqNumber = OLD.ReqNumber;
   ...> END;
sqlite>
```

CREATE VIEW

A VIEW is a saved SELECT statement that can be used in much the same way as a table. However in SQLite a view can not be used to add, update or delete the records in the underlying tables.

```
CREATE VIEW view_name AS select_statement;
CREATE TEMPORARY VIEW database_name.view_name AS select_statement;
```

```
sqlite> CREATE VIEW 'ReqTotal' AS SELECT ReqEquip.ReqNumber 'Requisition',
   ...> ReqEquip.Requestor 'Requestor',ReqDate,
   ...>'$ ' || (ROUND(SUM(Quantity*ItemCost),2)) 'Req Total'
   ...> FROM ReqEquip,ReqDetail
   ...> WHERE ReqEquip.ReqNumber=ReqDetail.ReqNumber GROUP BY
ReqDetail.ReqNumber;
sqlite>
```

```
sqlite> .headers on
sqlite> .mode column
sqlite> .width 10 14 10 10
sqlite> select * from ReqTotal;
```

```
Requisition  Requestor       ReqDate     Req Total
-----------  --------------  ----------  ----------
1000         Carl Jones      2007/10/30  $ 24.12
1001         Peter Smith     2007/11/05  $ 13.51
1002         Carl Jones      2007/11/06  $ 10.57
1003         Mike Smith      2007/12/01  $ 2.31
1004         Steve North     2007/12/02  $ 19.4
1005         Harold Allen    2007/12/04  $ 54.91
```

However a trigger assigned to a view can be used to insert,update or delete records in underlying tables of the view.

Trigger to insert records into the underlying tables of a view

```
CREATE TRIGGER insert_view
INSTEAD OF INSERT ON view_name
FOR EACH ROW BEGIN
INSERT INTO table_one(field_1,field_2,field_3)
VALUES(NEW.field_1,NEW.field_2,NEW.field_3);
INSERT INTO table_two(field_4,field_5,field_6)
VALUES(NEW.field_4,field_5,field_6);
END;
```

Trigger to update records in the underlying tables of a view

```
CREATE TRIGGER update_view
INSTEAD OF UPDATE ON view_name FOR EACH ROW BEGIN UPDATE table_one
SET field_1= new.field_1
field_2 = new.field_2
field_3 = new.field_3
WHERE key_id = OLD.key_id ;
UPDATE table_two
SET field_4 = new.field_4
field_5 = new.field_5
field_6 = new.field_6
WHERE key_id = OLD.key_id;
END;
```

Working With Attached Databases in SQLite

SQLite allows the use of as many as ten attached databases in addition to the database loaded as "main". Tables from different databases can be included in the same query. Changes may be made to the schema and data contained in the attached databases. Attached databases may have table names that are identical to the database loaded as "main" or to other attached databases but a table cannot be created that duplicates a currently loaded table.

```
sqlite> .databases
seq   name             file
---   ---------------  ----------------------------------------------------------
0     main             C:\Databases\sqlite\sales2007.sqlite
1     temp             C:\DOCUME~1\COLINR~1\LOCALS~1\Temp\etilqs_Fr4rkeXfxA2wWVBo
```

```
ATTACH ' drive/path/database_name' AS database_alias;
```

```
sqlite> ATTACH 'sales2006.sqlite' AS lastyear;
sqlite> .databases
seq   name             file
---   ---------------  ----------------------------------------------------------
0     main             C:\Databases\sqlite\sales2007.sqlite
1     temp             C:\DOCUME~1\COLINR~1\LOCALS~1\Temp\etilqs_Fr4rkeXfxA2wWVB
                       o
2     lastyear         C:\Databases\sqlite\sales2006.sqlite
```

An unfortunate problem is that dot commands such as ".tables ", ".dump ", ".schema " will only list objects of the database loaded as main not those of the attached databases. Consequently you have to know the names of the attached database tables in order to refer to them.

While there is no command to list the names of the tables in an attached database, if you do know which tables are there, you can get the table structure with the SQLite specific PRAMGMA table_info statement.

```
PRAGMA table_info(table_name); /* Will work if the table name is not duplicated in another
database */
PRAGMA database_alias.table_info(table_name);
```

```
sqlite> PRAGMA lastyear.table_info(gross_sales);
cid|name|type|notnull|dflt_value|pk
0|year|INTEGER|0|'2006'|0
1|month|TEXT|0||0
2|monthlygross|REAL|0||0
```

```
3|sortcol|INTEGER|0||0
sqlite>
```

In the following example we wish to compare this year gross sales by month for a business against those of last year. As it happens the monthly sales figures for 2007 are in a table called "gross_sales" in the database loaded as "main" and the figures for 2006 are in a table also called "gross_sales" which can be found in the attached database "lastyear". In addition both "gross_sales" tables use duplicate column names. This requires the use of dot notation in the query in order to specify the database, table and field being referred to as shown below.

```
main. table_name . field_name
database_alias. table_name . field_name
```

```
sqlite> .mode columns
sqlite> ..width 10 15 15
sqlite> sqlite> SELECT main.gross_sales.month,main.gross_sales.monthlygross ,
lastyear.gross_sales.monthlygross FROM main.gross_sales,lastyear.gross_sales WHERE
main.gross_sales.sortcol = lastyear.gross_sales.sortcol ;
month       monthlygross     monthlygross
----------  ---------------  ---------------
January     27580.56         34000.12
February    29321.58         32453
March       47412.42         45002.02
April       49300.43         47210.33
May         48990.52         56778.43
June        51014.23         48944.5
July        42530.89         45300
August      37899.25         53020.12
September   38596.56         51012.23
October     33015.55         34500
November    23564.56         27802.09
December    17825.13         19330.94
sqlite>
```

The "main.gross_sales.sortcol" and "lastyear.gross_sales.sortcol" columns which are used in the where clause are a numeric representation of the month so that only sales from the same month are compared with each other. Since the field names for "gross_sales" tables for 2007 and 2006 are identical there are two columns called "monthlygross" in the result. In this next example we will give columns for each year an alias and we will also calculate the percentage change in sales.

```
SELECT main.gross_sales.month 'Month',main.gross_sales.monthlygross '2007' ,
lastyear.gross_sales.monthlygross '2006' , ROUND((((main.gross_sales.monthlygross
-lastyear.gross_sales.monthlygross) / lastyear.gross_sales.monthlygross )*100),2 ) 'Percent
Change' FROM main.gross_sales,lastyear.gross_sales WHERE main.gross_sales.sortcol =
lastyear.gross_sales.sortcol;
```

```
Month       2007       2006       Percent Change
----------  ---------- ---------- --------------
January     27580.56   34000.12   -18.88
February    29321.58   32453      -9.65
March       47412.42   45002.02   5.36
April       49300.43   47210.33   4.43
May         48990.52   56778.43   -13.72
June        51014.23   48944.5    4.23
July        42530.89   45300      -6.11
August      37899.25   53020.12   -28.52
September   38596.56   51012.23   -24.34
October     33015.55   34500      -4.3
November    23564.56   27802.09   -15.24
December    17825.13   19330.94   -7.79
```

Detaching a Database

To detach a database use the following command.

```
DETACH database_alias;
```

Opening SQLite from Windows Explorer

When SQLite is used in the standard manner by using the command "sqlite3 database_name " at the command prompt then a file is created if it does not already exist to store database changes as they occur.

If on the other hand, you open SQLite in MS Windows by clicking on sqlite3.exe in Windows Explorer then the command prompt window will be opened without being attached to a database. What this means is that any database objects created or data entered will exist only in the RAM memory and will disappear once the command prompt window is closed unless they are copied to an attached database. This page explains how to do that.

That the database exists only in the temporary memory can be shown in this particular instance by entering ".databases" at the command prompt and pressing enter. The result should look like the following.

```
SQLite version 3.5.2
Enter ".help" for instructions

sqlite> .databases
seq  name             file
---  ---------------  ----------------------------------------------------------
0    main
sqlite >
```

Note that the listing to the right of "main" and under the file column is blank, which means that there is no file designated to save the database objects to. In addition if you now enter ".tables" at the prompt and press enter. The result will be an empty prompt as no tables yet exist in the workspace.

```
sqlite> sqlite> .tables
sqlite>
```

Let's create a table using the following statement.

```
CREATE TABLE Customers(Acctnumber INTEGER PRIMARY KEY,Custname
VARCHAR(50),Addr1 VARCHAR(50),Addr2 VARCHAR(50),City VARCHAR(30),State
CHAR(2),Zipcode VARCHAR(10),Contact VARCHAR(30),Phone VARCHAR(10));
```

```
sqlite> CREATE TABLE Customers(Acctnumber INTEGER PRIMARY KEY,Custname
VARCHAR(5 0),Addr1 VARCHAR(50),Addr2 VARCHAR(50),City VARCHAR(30),State
CHAR(2),Zipcode VA RCHAR(10),Contact VARCHAR(30),Phone VARCHAR(10));
sqlite> .tables
Customers
```

```
sqlite> .databases
seq  name             file
---  ---------------  ----------------------------------------------------------
0    main
1    temp             C:\DOCUME~1\COLINR~1\LOCALS~1\Temp\etilqs_c6rO5UpzkcPd9jEr
sqlite >
```

The temp database is automatically created when something is created in the main database. It will hold your work during the session but which will disappear when you quit the program. If the command ".tables" is again entered at the prompt, we can see that the table " Customers"now exists. However when the simple select query "select * from Customers;" is entered the result is a blank prompt because the table is empty.

```
sqlite> .tables
Customers
sqlite>
sqlite> select * from Customers;
sqlite>
```

Now if we paste the following statements at the prompt, the data will be loaded into the table.

```
INSERT INTO Customers(Acctnumber, Custname,Addr1,Addr2,City,State,Zipcode
,Contact,Phone) VALUES( 130169,'Acme Widgets','1744 Alder Road','Apt
31C','Springfield','VA','20171','Alan Allen','5715551267');
INSERT INTO Customers(Acctnumber, Custname,Addr1,Addr2,City,State,Zipcode
,Contact,Phone) VALUES( 130208,'Nike Missiles Inc','5946 Oak
Drive','','Springfield','VA','20171','Lucy Baker','5715558762');
INSERT INTO Customers(Acctnumber, Custname,Addr1,Addr2,City,State,Zipcode
,Contact,Phone) VALUES( 130247,'Charlies Bakery','7116 Ginko St','suite
100','Springfield','VA','20171','Susan Nordstrom','5715552363');
INSERT INTO Customers(Acctnumber, Custname,Addr1,Addr2,City,State,Zipcode
,Contact,Phone) VALUES( 130286,'Unisales Inc.','8438 Maple Ave',' ','Springfield','VA','20171-
3521','Roger Norton','5715551418');
INSERT INTO Customers(Acctnumber, Custname,Addr1,Addr2,City,State,Zipcode
,Contact,Phone) VALUES( 130325,'M.I. Sinform & Sons','1785 Elm Avenue','P.O. Box
31','Springfield','VA','20171','Mike I. Sinform','5715558760');
INSERT INTO Customers(Acctnumber, Custname,Addr1,Addr2,City,State,Zipcode
,Contact,Phone) VALUES( 130364,'Big Dents Towing Inc.','7578 Spruce St.','Building 31
A','Springfield','VA','20171-5231','George Spencer','5715557855');
```

When the "select * from Customers;" is reentered the we can see that the data has been entered correctly. Note the " .headers ON " command that preceeded the select query which tells SQLite that I want the field names to be listed as well as the data. This has only to be entered once during the session.

```
sqlite> .headers ON
sqlite> select * from Customers;
Acctnumber|Custname|Addr1|Addr2|City|State|Zipcode|Contact|Phone
130169|Acme Widgets|1744 Alder Road|Apt 31C|Springfield|VA|20171|Alan Allen|5715551267
130208|Nike Missiles Inc|5946 Oak Drive||Springfield|VA|20171|Lucy Baker|5715558762
130247|Charlies Bakery|7116 Ginko St|suite 100|Springfield|VA|20171|Susan Nordstrom|5715552363
130286|Unisales Inc.|8438 Maple Ave||Springfield|VA|20171-3521|Roger Norton|5715551418
130325|M.I. Sinform & Sons|1785 Elm Avenue|P.O. Box 31|Springfield|VA|20171|Mike I. Sinform|5715558760
130364|Big Dents Towing Inc.|7578 Spruce St.|Building 31 A|Springfield|VA|20171-5231|George Spencer|
5715557855
sqlite>
```

Using the ATTACH statement we can create a persistant database file or open it if it already exists. Note that the file name is in quotes. It is not necessary for the file name to have a suffix but it is not a bad idea in order distinguish sqlite databases from other types of files. If we once again query the database list with the ".databases" command we can see that the file has the alias "newdb". Currently the table we created exists only in the memory.

```
sqlite>ATTACH 'ServiceCtr.db' AS newdb;
```

```
sqlite> .databases
seq  name             file
---  ---------------  ----------------------------------------------------------
0    main
1    temp             C:\DOCUME~1\COLINR~1\LOCALS~1\Temp\etilqs_c6rO5UpzkcPd9jEr
2    newdb            C:\Databases\sqlite\ServiceCtr.db
sqlite >
```

The easiest way of saving the table to the attached database is to use the statement "CREATE TABLE attached_database.tablename AS SELECT * FROM main.tablename" which selects all the columns and all the records from the table in "main" as shown below.

```
CREATE TABLE attached_database.tablename AS SELECT * FROM main.tablename;
CREATE TABLE attached_database.tablename AS SELECT field1,field2,field3 FROM
main.tablename;
```

```
CREATE TABLE newdb.Customers AS SELECT Acctnumber, Custname ,Addr1 ,Addr2 ,City
,State ,Zipcode ,Contact ,Phone FROM main.Customers ;
sqlite>
sqlite> select * from newdb.Customers;
Acctnumber|Custname|Addr1|Addr2|City|State|Zipcode|Contact|Phone
```

```
130169|Acme Widgets|1744 Alder Road|Apt 31C|Springfield|VA|20171|Alan Allen|5715551267
130208|Nike Missiles Inc|5946 Oak Drive||Springfield|VA|20171|Lucy Baker|5715558762
130247|Charlies Bakery|7116 Ginko St|suite 100|Springfield|VA|20171|Susan Nordstrom|5715552363
130286|Unisales Inc.|8438 Maple Ave||Springfield|VA|20171-3521|Roger Norton|5715551418
130325|M.I. Sinform & Sons|1785 Elm Avenue|P.O. Box 31|Springfield|VA|20171|Mike I. Sinform|5715558760
130364|Big Dents Towing Inc.|7578 Spruce St.|Building 31 A|Springfield|VA|20171-5231|George Spencer|
5715557855
sqlite>
```

Now the data will be stored in the ServiceCenter.db file. Once again a simple query will verify that the records have been transferred.

SQL Quick Reference

CREATE TABLE

CREATE TABLE table_name(fieldname_1 data_type, fieldname_2 data_type, fieldname_3 data_type);

Create Tables with Foreign Keys

One to Many

CREATE TABLE child_table_name (field_1 INTEGER PRIMARY KEY, field_2 TEXT, foreign_key_field INTEGER , FOREIGN KEY(foreign_key_field) REFERENCES parent_table_name(parent_key_field));

One to One relationship

CREATE TABLE child_table_name (field_1 INTEGER PRIMARY KEY, field_2 TEXT, foreign_key_field INTEGER UNIQUE , FOREIGN KEY(foreign_key_field) REFERENCES parent_table_name (parent_key_field));

FOREIGN KEY PHRASE with CASCADE

FOREIGN KEY(foreign_key_field) REFERENCES parent_table_name(parent_key_field) ON DELETE CASCADE ON UPDATE CASCADE);

CREATE TABLE from SQL Query Results

CREATE TABLE new_table_name AS SELECT select statement

Delete Table

DROP TABLE table_name ;
DROP TABLE main.table_name ;
DROP TABLE attached_database_name.table_name ;

Add Records to a Table

INSERT INTO table_name (field_1, field_2, field_3) VALUES ("value a", "value b",0.00)

UPDATE Record in TABLE

UPDATE table_name SET column_name= value WHERE criteria;

SQLite

SELECT Statements

					Syntax or Purpose
---------	---------	--------------	----------	------------	-------------
SELECT					
	DISTINCT				Exclude duplicate records for fields selected.
		CASE WHEN expression			Conditional expression
		THEN expression			
		ELSE expression END			
	FROM	table_name			Multiple table names are separated by commas.
	WHERE				Row level filtering
		expression	AND	expression	
		expression	OR	expression	
		IN			Comma delimited list enclosed in parenthesis
		NOT IN			Comma delimited list enclosed in parenthesis
		BETWEEN			Select records within the specified numeric range
		NOT BETWEEN			Select records outside of the specified numeric range
		LIKE			String with wildcard (%) enclosed in parenthesis
		NOT LIKE			String with wildcard (%) enclosed in parenthesis
	GROUP BY				
	HAVING				
		ORDER BY			Sorting of the output using a comma delimited list of column names
		LIMIT			Limit the number of rows returned

SQLite

Expressions

Symbol	Meaning	Example
*****	Multiply	field_name * 0.05
+	Add	field_name + 1
-	Subtract	field_name - 0.5
/	Divide	field_1 / field_2
=	equal to	field_1 = field_2
==	equal to	field_1 = 1
<	less than	field_b < 100
<=	less than or equal to	field_a <= 99
>	Greater than	field_name > 2
>=	Greater than or equal to	field_name >= 15.142
<>	Not equal to	field_name <> 0
!=	Not equal to	field_a != field_b

SELECT Statement Examples

SELECT * FROM table_name;

SELECT delivery_addr,invoice_number FROM customer_info,invoice_picked WHERE customer_info.customer_id = invoice_picked.customer_id;

SELECT acct_number , customer_name FROM sales_2007 WHERE purchase_total BETWEEN 1200 AND 3300;
SELECT acct_number , customer_name FROM sales_2007 WHERE city IN ('Chicago','New York','Cleveland');

SELECT acct_number , customer_name FROM sales_2007 WHERE purchase_total NOT BETWEEN 1700 AND 2200;

SELECT DISTINCT customer_name , acct_number FROM orders WHERE invoice_no > 20000 ;

SELECT employee_id, hours, CASE WHEN (hours>40 THEN hours-40 ELSE 0 END AS overtime FROM tbl_Hours;

String Manipulation

Function	Explanation	Syntax
LENGTH( )	returns the number of characters in the string	length(fieldname or expression)
LTRIM( )	Trims listed characters from the beginning of a string, if the only argument that is provided is a field name or expression then the function will trim only white space.	LTRIM(fieldname or expression,' characters')
RTRIM( )	Trims listed characters from the end of a string, if the only argument that is provided is a field name or expression then the function will trim only white space.	RTRIM(fieldname or expression,' characters')
TRIM ( )	Trims listed characters from both ends of a string, if the only argument that is provided is a field name or expression then the function will trim only white space.	TRIM(fieldname or expression ,' characters')
QUOTE( )	returns field enclosed in single quotes	QUOTE(fieldname or expression)
||	Concatenate strings	string_one || string_two
SUBSTR( )	Extracts part of a string.	SUBSTR(field_name,start_location,substring_length)
REPLACE( )	Searches column or field for string specified in the second argument and replaces it with the string in the third argument.	REPLACE(field_name,'old_string','new_string')

SQLite

Aggregate Functions

Function	Explanation	Syntax
AVG( )	Averages the value of the column or grouping	AVG(fieldname or expression)
COUNT( )	Returns the number of rows in the column or grouping	COUNT(fieldname or expression)
GROUP_CONCAT( )	Generates a string of non null values in a column separated by commas or some other specified delimiter	GROUP_CONCAT(column name)
MAX( )	Returns the highest value found in the column or grouping	MAX(fieldname or expression)
MIN( )	Returns the lowest value found in the column or grouping	MIN(fieldname or expression)
SUM( )	Total of the values in the column or grouping added together	SUM(fieldname or expression)
TOTAL( )	Specific to SQLITE SQL, always returns the floating point sum of the values a column or grouping	TOTAL(fieldname or expression)

Date and Time

Function	Explanation
CURRENT_DATE	Returns UTC Date in YYYY-MM-DD Format
CURRENT_TIME	Returns UTC Time in HH:MM:SS Format
CURRENT_TIMESTAMP	Returns Current UTC Date and time
datetime("now","localtime")	Returns current local date and time as YYYY-MM-DD HH:MM:SS
date("now","localtime")	Returns local date as YYYY-MM-DD
time("now","localtime")	Returns local time as HH:MM:SS

Other Keywords and Functions

sqlite_master	table recording schema of the database. Automatically updated Read only to user.
sqlite_temp_master	table recording schema of temporary objects during a database session.
last_insert_rowid()	Row id of last record inserted during a session. Caution this value is not table specific.

SQLite

Identify The Version of SQLite in use

```
sqlite> select sqlite_version();
3.7.5
sqlite>
```

```
vacuum;
sqlite>
```

Comments in SQLite

It is often useful to include comments within CREATE TABLE or CREATE VIEW statements for later reference.

```
-- Single line comments are preceeded by two minus signs.
/* Multiline comments are preceeded by a forward slash and asterisk and
are terminated by an asterisk followed by a forward slash */
```

SQLITE Dot Commands

.dump
: list SQL statements used to create the database including INSERT statements. If the dump command is followed by a table,view name will only list SQL statements associated with that element

.schema
: list all tables and indices in database main. If followed by table, view name will list the CREATE statement for that element

.explain
: Changes output mode to column and creates apropriate widths for data in each displayed column.

.tables
: list all tables in database main. Equivalent to SHOW TABLES in other SQL versions.

```
SQLite version 3.7.5
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

```
sqlite> .help
.backup ?DB? FILE        Backup DB (default "main") to FILE
.bail ON|OFF             Stop after hitting an error. Default OFF
.databases               List names and files of attached databases
.dump ?TABLE? ...        Dump the database in an SQL text format
                           If TABLE specified, only dump tables
                         matching
                           LIKE pattern TABLE.
.echo ON|OFF             Turn command echo on or off
```

.exit	Exit this program
.explain ?ON|OFF?	Turn output mode suitable for EXPLAIN on or off.
	With no args, it turns EXPLAIN on.
.header(s) ON|OFF	Turn display of headers on or off
.help	Show this message
.import FILE TABLE	Import data from FILE into TABLE
.indices ?TABLE?	Show names of all indices
	If TABLE specified, only show indices for tables
	matching LIKE pattern TABLE.
.load FILE ?ENTRY?	Load an extension library
.log FILE|off	Turn logging on or off. FILE can be stderr/stdout
.mode MODE ?TABLE?	Set output mode where MODE is one of:
	csv Comma-separated values
	column Left-aligned columns. (See .width)
	html HTML <table> code
	insert SQL insert statements for TABLE
	line One value per line
	list Values delimited by .separator string
	tabs Tab-separated values
	tcl TCL list elements
.nullvalue STRING	Print STRING in place of NULL values
.output FILENAME	Send output to FILENAME
.output stdout	Send output to the screen
.prompt MAIN CONTINUE	Replace the standard prompts
.quit	Exit this program
.read FILENAME	Execute SQL in FILENAME
.restore ?DB? FILE	Restore content of DB (default "main") from FILE
.schema ?TABLE?	Show the CREATE statements
	If TABLE specified, only show tables matching
	LIKE pattern TABLE.
.separator STRING	Change separator used by output mode and .import
.show	Show the current values for various settings
.stats ON|OFF	Turn stats on or off

SQLite

```
.tables ?TABLE?          List names of tables
                           If TABLE specified, only list tables matching
                           LIKE pattern TABLE.
.timeout MS              Try opening locked tables for MS milliseconds
.width NUM1 NUM2 ...     Set column widths for "column" mode
.timer ON|OFF            Turn the CPU timer measurement on or off
```