SQLite Tutorial

Copyright (c) 2004 by Mike Chirico mchirico@users.sourceforge.net
This material may be distributed only subject to the terms and conditions set forth in the Open Publication License v1.0, 8 June 1999 or later.

This article explores the power and simplicity of sqlite3, starting with common commands and triggers. It then covers the attach statement with the union operation, introduced in a way that allows multiple tables, in separate databases, to be combined as one virtual table, without the overhead of copying or moving data. Next, I demonstrate the simple sign function and the amazingly powerful trick of using this function in SQL select statements to solve complex queries with a single pass through the data, after making a brief mathematical case for how the sign function defines the absolute value and IF conditions.

Although the sign function currently does not exist in sqlite3, it is very easy to create in the "/src/func.c" file so that this function will be permanently available to all sqlite applications. Normally, user functions are created in C, Perl, or C++, which is also documented in this article. sqlite3 has the ability to store "blob", binary data. The sample program in the download, "eatblob.c", reads a binary file of any size into memory and stores the data in a user-specified field.

This tutorial was made with sqlite3 version 3.0.8.

SQLite Tutorial

Getting Started

Common Commands

To create a database file, run the command "sqlite3", followed by the database name. For example, to create the database "test.db", run the sqlite3 command as follows:

```
$ sqlite3 test.db
SQLite version 3.0.8
Enter ".help" for instructions
sqlite> .quit
$
```

The database file test.db will be created, if it does not already exist. Running this command will leave you in the sqlite3 environment. There are three ways to safely exit this environment: .q, .quit, and .exit.

You do not have to enter the sqlite3 interactive environment. Instead, you could perform all commands at the shell prompt, which is ideal when running bash scripts and commands in an ssh string. Here is an example of how you would create a simple table from the command prompt:

```
$ sqlite3 test.db  "create table t1 (t1key INTEGER
        PRIMARY KEY,data TEXT,num double,timeEnter DATE);"
```

After table t1 has been created, data can be inserted as follows:

```
$ sqlite3 test.db  "insert into t1 (data,num) values ('This is sample data',3);"
$ sqlite3 test.db  "insert into t1 (data,num) values ('More sample data',6);"
$ sqlite3 test.db  "insert into t1 (data,num) values ('And a little more',9);"
```

As expected, doing a select returns the data in the table. Note that the primary key "t1key" auto increments; however, there are no default values for timeEnter. To populate the timeEnter field with the time, an update trigger is needed. Note that you should not use the abbreviation "INT" when working with the PRIMARY KEY. You must use "INTEGER" for the primary key to update.

```
$ sqlite3 test.db  "select * from t1 limit 2";
1|This is sample data|3|
2|More sample data|6|
```

In the statement above, the limit clause is used, and only two rows are displayed. For a quick reference of SQL syntax statements available with SQLite, see the syntax page. There is an offset option for the limit clause. For instance, the third row is equal to the following: "limit 1 offset 2".

```
$ sqlite3 test.db "select * from t1 order by t1key limit 1 offset 2";
3|And a little more|9|
```

The ".table" command shows the table names. For a more comprehensive list of tables, triggers, and indexes created in the database, query the master table "sqlite_master", as shown below.

```
$ sqlite3 test.db ".table"
```

```
t1

$ sqlite3 test.db "select * from sqlite_master"
table|t1|t1|2|CREATE TABLE t1 (t1key INTEGER
        PRIMARY KEY,data TEXT,num double,timeEnter DATE)
```

All SQL information and data inserted into a database can be extracted with the ".dump" command. Also, you might want to look for the "~/.sqlite_history" file.

```
$ sqlite3 test.db ".dump"
BEGIN TRANSACTION;
CREATE TABLE t1 (t1key INTEGER
        PRIMARY KEY,data TEXT,num double,timeEnter DATE);
INSERT INTO "t1" VALUES(1, 'This is sample data', 3, NULL);
INSERT INTO "t1" VALUES(2, 'More sample data', 6, NULL);
INSERT INTO "t1" VALUES(3, 'And a little more', 9, NULL);
COMMIT;
```

The contents of the ".dump" can be filtered and piped to another database. Below, table t1 is changed to t2 with the sed command, and it is piped into the test2.db database.

```
$ sqlite3 test.db ".dump"|sed -e s/t1/t2/|sqlite3 test2.db
```

Triggers

An insert trigger is created below in the file "trigger1". The Coordinated Universal Time (UTC) will be entered into the field "timeEnter", and this trigger will fire *after* a row has been inserted into the table t1.

```
-- ************************************************************************
--   Creating a trigger for timeEnter
--     Run as follows:
--           $ sqlite3 test.db < trigger1
-- ************************************************************************
CREATE TRIGGER insert_t1_timeEnter AFTER  INSERT ON t1
BEGIN
 UPDATE t1 SET timeEnter = DATETIME('NOW')  WHERE rowid = new.rowid;
END;
-- ************************************************************************
```

The AFTER specification in ..."insert_t1_timeEnter AFTER..." is necessary. Without the AFTER keyword, the rowid would not have been generated. This is a common source of errors with triggers, since AFTER is *not* the default, so it must be specified. If your trigger depends on newly-created data in any of the fields from the created row (which was the case in this example, since we need the rowid), the AFTER specification is needed. Otherwise, the trigger is a BEFORE trigger, and will fire before rowid or other pertinent data is entered into the field.

Comments are preceded by "--". If this script were created in the file "trigger1", you could easily execute it as follows.

```
$ sqlite3 test.db < trigger1
```

Now try entering a new record as before, and you should see the time in the field timeEnter.

```
$ sqlite3 test.db  "insert into t1 (data,num) values ('First entry with timeEnter',19);"

$ sqlite3 test.db "select * from t1";
1|This is sample data|3|
2|More sample data|6|
3|And a little more|9|
4|First entry with timeEnter|19|2004-10-02 15:12:19
```

The last value has timeEnter filled automatically with Coordinated Universal Time, or UTC. If you want localtime, use select datetime('now','localtime'). See the note at the end of this section regarding UTC and localtime.

For the examples that follow, the table "exam" and the database "examScript" will be used. The table and trigger are defined below. Just like the trigger above, UTC time will be used.

```
-- ********************************************************************
--  examScript: Script for creating exam table
--   Usage:
--       $ sqlite3 examdatabase < examScript
--
--   Note: The trigger insert_exam_timeEnter
--         updates timeEnter in exam
-- ********************************************************************
-- ********************************************************************
CREATE TABLE exam (ekey      INTEGER PRIMARY KEY,
                fn        VARCHAR(15),
                ln        VARCHAR(30),
                exam      INTEGER,
                score     DOUBLE,
                timeEnter DATE);

CREATE TRIGGER insert_exam_timeEnter AFTER  INSERT ON exam
BEGIN

UPDATE exam SET timeEnter = DATETIME('NOW')
        WHERE rowid = new.rowid;
END;
-- ********************************************************************
-- ********************************************************************
```

Here's an example usage:

```
$ sqlite3 examdatabase < examScript
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score)
      values ('Anderson','Bob',1,75)"

$ sqlite3 examdatabase "select * from exam"

1|Bob|Anderson|1|75|2004-10-02 15:25:00
```

As you can see, the PRIMARY KEY and current UTC time have been updated correctly.

Logging All Inserts, Updates, and Deletes

The script below creates the table examlog and three triggers (update_examlog, insert_examlog, and delete_examlog) to record updates, inserts, and deletes made to the exam table. In other words, whenever a change is made to the exam table, the changes will be recorded in the examlog table, including the old value and the new value. If you are familiar with MySQL, the functionality of this log table is similar to MySQL's binlog. See Tips 2, 24, and 25 if you would like more information on MySQL's log file.

```
-- ********************************************************************
--  examLog: Script for creating log table and related triggers
--   Usage:
--       $ sqlite3 examdatabase < examLOG
--
--
-- ********************************************************************
-- ********************************************************************
CREATE TABLE examlog (lkey INTEGER PRIMARY KEY,
                ekey INTEGER,
                ekeyOLD INTEGER,
                fnNEW   VARCHAR(15),
                fnOLD   VARCHAR(15),
                lnNEW   VARCHAR(30),
                lnOLD   VARCHAR(30),
                examNEW INTEGER,
                examOLD INTEGER,
                scoreNEW DOUBLE,
                scoreOLD DOUBLE,
                sqlAction VARCHAR(15),
                examtimeEnter    DATE,
                examtimeUpdate   DATE,
                timeEnter        DATE);

--  Create an update trigger
CREATE TRIGGER update_examlog AFTER UPDATE  ON exam
BEGIN

  INSERT INTO examlog  (ekey,ekeyOLD,fnOLD,fnNEW,lnOLD,
                    lnNEW,examOLD,examNEW,scoreOLD,
                    scoreNEW,sqlAction,examtimeEnter,
                    examtimeUpdate,timeEnter)

         values (new.ekey,old.ekey,old.fn,new.fn,old.ln,
                new.ln,old.exam, new.exam,old.score,
                new.score, 'UPDATE',old.timeEnter,
                DATETIME('NOW'),DATETIME('NOW') );

END;
--
--  Also create an insert trigger
--    NOTE  AFTER keyword ------v
CREATE TRIGGER insert_examlog AFTER INSERT ON exam
BEGIN
INSERT INTO examlog  (ekey,fnNEW,lnNEW,examNEW,scoreNEW,
                  sqlAction,examtimeEnter,timeEnter)
```

```
        values (new.ekey,new.fn,new.ln,new.exam,new.score,
              'INSERT',new.timeEnter,DATETIME('NOW') );

END;

--  Also create a DELETE trigger
CREATE TRIGGER delete_examlog DELETE ON exam
BEGIN

INSERT INTO examlog  (ekey,fnOLD,lnNEW,examOLD,scoreOLD,
                    sqlAction,timeEnter)

        values (old.ekey,old.fn,old.ln,old.exam,old.score,
              'DELETE',DATETIME('NOW') );

END;
-- ********************************************************************
-- ********************************************************************
```

Since the script above has been created in the file examLOG, you can execute the commands in sqlite3 as shown below. Also shown below is a record insert, and an update to test these newly-created triggers.

```
 $ sqlite3 examdatabase < examLOG

 $ sqlite3 examdatabase "insert into exam
                          (ln,fn,exam,score)
                        values
                          ('Anderson','Bob',2,80)"

 $ sqlite3 examdatabase "update exam set score=82
                           where
                         ln='Anderson' and fn='Bob' and exam=2"
```

Now, by doing the select statement below, you will see that examlog contains an entry for the insert statement, plus two updates. Although we only did one update on the commandline, the trigger "insert_exam_timeEnter" performed an update for the field timeEnter; this was the trigger defined in "examScript". In the second update, we can see that the score has been changed. The trigger is working. Any change made to the table, whether by user interaction or another trigger, is recorded in the examlog.

```
 $ sqlite3 examdatabase "select * from examlog"

 1|2||Bob||Anderson||2||80||INSERT|||2004-10-02 15:33:16
 2|2|2|Bob|Bob|Anderson|Anderson|2|2|80|80|UPDATE||2004-10-02 15:33:16|2004-10-02 15:33:16
 3|2|2|Bob|Bob|Anderson|Anderson|2|2|82|80|UPDATE|2004-10-02 15:33:16|2004-10-02 15:33:26|2004-10-02
15:33:26
```

Again, pay particular attention to the AFTER keyword. Remember that by default, triggers are BEFORE, so you must specify AFTER to insure that all new values will be available if your trigger needs to work with any new values.

UTC and Localtime

Note that select DATETIME('NOW') returns UTC or Coordinated Universal Time, but select datetime('now','localtime') returns the current time.

```
sqlite> select datetime('now');
2004-10-18 23:32:34

sqlite> select datetime('now','localtime');
2004-10-18 19:32:46
```

There is an advantage to inserting UTC time like we did with the triggers above, since UTC can easily be converted to localtime after UTC has been entered in the table. See the command below. By inserting UTC, you avoid problems when working with multiple databases that may not share the same timezone and/or dst settings. By starting with UTC, you can always obtain the localtime. (Reference: Working with Time)

```
CONVERTING TO LOCALTIME:

sqlite> select datetime(timeEnter,'localtime') from exam;
```

Other Date and Time Commands

If you look in the sqlite3 source file "./src/date.c", you will see that datetime takes other options. For example, to get the localtime, plus 3.5 seconds, plus 10 minutes, you would execute the following command:

```
sqlite> select datetime('now','localtime','+3.5 seconds','+10 minutes');
2004-11-07 15:42:26
```

It is also possible to get the weekday where 0 = Sunday, 1 = Monday, 2 = Tuesday ... 6 = Saturday.

```
sqlite> select datetime('now','localtime','+3.5 seconds','weekday 2');
2004-11-09 15:36:51
```

The complete list of options, or modifiers as they are called in this file, are as follows:

```
NNN days
NNN hours
NNN minutes
NNN.NNNN seconds
NNN months
NNN years
start of month
start of year
start of week
start of day
weekday N
unixepoch
localtime
utc
```

In addition, there is the "strftime" function, which will take a timestring, and convert it to the specified format, with the modifications. Here is the format for this function:

```
**    strftime( FORMAT, TIMESTRING, MOD, MOD, ...)
**
** Return a string described by FORMAT.  Conversions as follows:
**
**   %d  day of month
**   %f  ** fractional seconds  SS.SSS
**   %H  hour 00-24
**   %j  day of year 000-366
**   %J  ** Julian day number
**   %m  month 01-12
**   %M  minute 00-59
**   %s  seconds since 1970-01-01
**   %S  seconds 00-59
**   %w  day of week 0-6  sunday==0
**   %W  week of year 00-53
**   %Y  year 0000-9999
```

Below is an example.

```
sqlite> select strftime("%m-%d-%Y %H:%M:%S %s %w %W",'now','localtime');
11-07-2004 16:23:15 1099844595 0 44
```

The ATTACH Command: Build a Virtual Table that Spans Multiple Tables on Separate Databases

This is a very powerful concept. As you have seen, sqlite3 works with a local database file. Within this local database, multiple tables can be created. This section will examine a technique to combine multiple tables with the same field layout that exist in separate database files into a single virtual table. On this single virtual table, you will see how selects can be performed. There is no overhead in copying or moving data. No data gets copied or moved, period. This is the ideal situation when working with very large tables. Suppose the computers on your network record port scans from snort to a local sqlite3 file. Provided you have access to the individual database files, via NFS mount or samba mount, you could virtually combine the tables from all your computers into one virtual table to perform database queries in an effort to identify global patterns of attack against your network.

This example will be done with the examdatabase, since we still have the scripts that were used for the exam table. We can easily create a new database "examdatabase2", along with a new exam table, by executing the following script from the bash shell:

```
$ sqlite3 examdatabase2 < examScript
$ sqlite3 examdatabase2 < examLOG
$ sqlite3 examdatabase2 "insert into exam (ln,fn,exam,score) values ('Carter','Sue',1,89);
                insert into exam (ln,fn,exam,score) values ('Carter','Sue',2,100);"

$ sqlite3 examdatabase2 "select * from exam"
1|Sue|Carter|1|89|2004-10-02 16:04:12
2|Sue|Carter|2|100|2004-10-02 16:04:12
```

To combine the two database files, use the attach command. The alias for examdatabase will be e1, and the alias for examdatabase2 will be e2. The shorter names will come in handy when the tables are joined with the union clause (a standard SQL command).

After the "attach" database command is performed, the ".database" command can be used to show the location of the individual database files. The location follows the alias. See the example below.

```
$ sqlite3
SQLite version 3.0.8
Enter ".help" for instructions
sqlite> attach database 'examdatabase' as e1;
sqlite> attach database 'examdatabase2' as e2;
sqlite> .database
seq  name            file
---  --------------  ----------------------------------------------------------
0    main
2    e1              /work/cpearls/src/posted_on_sf/sqlite_examples/sqlite_exam
3    e2              /work/cpearls/src/posted_on_sf/sqlite_examples/sqlite_exam
sqlite>
```

To select all data from both tables, perform the union of two select statements as demonstrated below. Note that by adding 'e1' and 'e2' to the respective selects, it is possible to identify which database the returned records are coming from.

```
sqlite> select 'e1',* from e1.exam union select 'e2',* from e2.exam;

e1|1|Bob|Anderson|1|75|2004-10-02 15:25:00
e1|2|Bob|Anderson|2|82|2004-10-02 15:33:16
e2|1|Sue|Carter|1|89|2004-10-02 16:04:12
e2|2|Sue|Carter|2|100|2004-10-02 16:04:12
```

To summarize: A query was performed on two tables that resided in separate databases. This union created the virtual table. The select syntax is as follows: SELECT <expression> FROM < TABLE>. For the table option, we have used the complete string "(select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)", which is our virtual table.

Here is a query example performed on this virtual table. Suppose you wanted the maximum score by exam across databases.

```
sqlite> select  exam,max(score) from
      (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
        group by exam;

1|89
2|100
```

No problem. You got the maximum score for each exam, but who does it below to? Find the ln and fn, but be careful; if you add "ln" and "fn" to the first part of the select, you will get the *wrong* answer.

```
sqlite> select  exam,max(score),ln,fn from
      (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
        group by exam;

   ** THIS IS INCORRECT; it should be Carter|Sue. **

1|89|Anderson|Bob
2|100|Anderson|Bob
```

"Anderson", "Bob" happens to be the name that dropped down in this select statement. It is not the correct answer. If, by chance, you got the correct answer by doing this query, it is because you entered the names in a different order. If that is the case, perform the query below, which takes the min(score) and gets an error on one of these examples.

Here, the min(score) is queried. By chance, because of the order in which data was entered into this table, the correct answer is displayed.

```
sqlite> select  exam,min(score),ln,fn from
      (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
        group by exam;

   ** correct answer -- just chance **

1|75|Anderson|Bob
2|82|Anderson|Bob
```

Clearly, there needs to be a better way of finding out who got the maximum and minimum scores for each exam. Here is the correct SQL statement which will always give the correct

answer:

```
sqlite> select db,ln,fn,exam,score from
      (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
         where
            (
              score=(
              select max(score) from
          (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
               where exam=1
                  )
              and exam = 1
              )
              OR
              (
              score=(
               select max(score) from
           (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
               where exam=2
                  )
              and exam = 2
               )  ;

  e2|Carter|Sue|1|89
  e2|Carter|Sue|2|100
```

Or it can be done as two independent select statements as follows:

```
sqlite> select db,ln,fn,exam,score from
      (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
       where exam=1 order by score desc limit 1;

 e2|Carter|Sue|1|89

 sqlite> select db,ln,fn,exam,score from
       (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
       where exam=2 order by score desc limit 1;

 e2|Carter|Sue|2|100
```

A Pivot Table

What if you wanted a pivot table in which the scores are listed across the top as exam1,exam2,..examN for each person? For example:

```
fn  ln         exam1  exam2
--- --------   -----  -----
Bob Anderson   75     82
Sue Carter     89     100
```

Also, is there a way to display the deltas between exams, to have a fifth column that would show 7 points (82-75) or the delta between exam1 and exam2 and similar data for Sue Carter?

Such power select statements can be done with the sign function. And unlike the case statement, the sign function can be placed in the GROUP BY and HAVING expressions of a SELECT statement. For example, taking a look at the general syntax of the SELECT statement, the sign function can be used anywhere you see an expression or expression-list.

```
SELECT [ALL | DISTINCT] result [FROM table-list]
[WHERE expr]
[GROUP BY expr-list]
[HAVING expr]
[compound-op select]*
[ORDER BY sort-expr-list]
[LIMIT integer [( OFFSET | , ) integer]]
```

The sign function does not exist in sqlite, but that is not a problem, since we can easily create it.

As a side note, you may wonder why you should create the sign function. Instead, why not create an IF or IIF function? The main reason is that the IF statement is not standard on all databases, and, on some databases where it is standard (MySQL), it was created incorrectly. Yes, if you are a MySQL user, take a look at the following LONGWINDED TIP 1 for an example of MySQL's incorrect IF statement and how the sign function solves this problem.

The Power of the Sign Function -- A Mathematical Explanation

It may come as a shock, but the problems in the last section, and much more, can be solved using the sign function. This is just the simple function in which sign(-200)=-1,.. sign(-1)=-1, sign(0)=0, sign(1)=1,.. sign(300)=1. So if the number is > 0 a 1 is returned. Zero is the only number that returns zero. All negative numbers return -1. Again, this simple function does not exist in sqlite, but you can easily create it, permanently. The next section will focus on the creation of this function, but here, the mathematical properties are explained.

The sign function can define the absolute value function abs() as the value of a number times its sign, or sign(x)*(x), abbreviated sign(x)(x). Here is a more detailed look at this function:

sign(x)(x) = abs(x)

Example, assume x=3

sign(3)(3) = abs(3)
1*3 = 3

Example, assume x=-3

sign(-3)(-3) = abs(-3)
-1*-3 = 3

Example, assume x=0

sign(0)(0) = abs(0)
0*0 = 0

Comparisons can be made with the sign function between two variables x and y. For instance, if sign(x-y) is 1, then, x is greater than y.

sign(x-y) is equal to 1 if x > y

sign(x-y) is equal to 0 if x = y

sign(x-y) is equal to -1 if x < y

Now look closely at the three statements below. The sign function starts to resemble an IF statement; a 1 is returned if and only if x = y. Thoroughly understanding the statements below is important, as the rest of the discussion quickly builds from these examples.

IF (X==Y) return 1; ELSE return 0;

can be expressed as follows:

1 - abs(sign(x-y)) is equal to 0 if x > y

1 - abs(sign(x-y)) is equal to 1 if x = y

1 - abs(sign(x-y)) is equal to 0 if x < y

It is possible to return a 1 if and only if x < y, otherwise return a zero.

```
IF ( X < Y ) return 1;  ELSE return 0;
```

can be expressed as follows:

```
1-sign(1+sign(x-y)) is equal to 0 if  x > y

1-sign(1+sign(x-y)) is equal to 0 if  x = y

1-sign(1+sign(x-y)) is equal to 1 if  x < y
```

The last example is known as the delta for x<y, or Delta[x<y]. This Delta notation will be used instead of writing it out in long form or using the IF statement. Therefore, the following is a summarized table of all the Delta functions or comparison operators.

```
Delta[x=y]   =   1 - abs(sign(x-y))

Delta[x!=y]  =   abs(sign(x-y))

Delta[x<y]   =   1-sign(1+sign(x-y))

Delta[x<=y]  =   sign(1-sign(x-y))

Delta[x>y]   =   1-sign(1-sign(x-y))

Delta[x>=y]  =   sign(1+sign(x-y))


Delta[z=x AND z=y] = sign( Delta[z=x]*Delta[z=y] )

Delta[z=x OR  z=y] = sign( Delta[z=x]+Delta[z=y] )

Delta[z>x AND  z<y] = sign( Delta[z>x]*Delta[z<y] )

... more can be defined ... but you get the idea
```

To summarize the following if statement, note the introduction of a third variable, z:

```
if( x==y )
      return z;
   else
     return 0;
```

The above expression, in Delta notation, is the following:

```
z*Delta[x=y]
```

Here is an interesting example:

```
create table logic (value int);

insert into logic (value) values (1);
insert into logic (value) values (0);
insert into logic (value) values (-1);
```

First, take the Cartesian product to show all possible combinations of x and y.

```
sqlite> .header on
sqlite> .mode column
sqlite> select x.value,y.value from logic x, logic y;

x.value     y.value
----------  ----------
1           1
1           0
1           -1
0           1
0           0
0           -1
-1          1
-1          0
-1          -1
```

After the sign function is created (which we will do in the next section), using the above table, we could examine Delta[x!=y] as follows;

```
sqlite> .header on
sqlite> .mode column
sqlite> select x.value,y.value,abs(sign(x.value-y.value)) from logic x, logic y;


x.value     y.value     abs(sign(x.value-y.value))
----------  ----------  --------------------------
1           1           0
1           0           1
1           -1          1
0           1           1
0           0           0
0           -1          1
-1          1           1
-1          0           1
-1          -1          0
```

Note that every time x is not equal to y, abs(sign(x.value-y.value)) returns a 1. After the sign function is created, these example will run. This is extremely powerful. To show that we have created a condition statement without using the where or group by statements, consider the following example. z.value will only be displayed in the right hand column when x.value != y.value.

```
sqlite> select x.value,y.value,z.value,
            z.value*abs(sign(x.value-y.value))
            from logic x, logic y, logic z;

x.value     y.value     z.value     z.value*abs(sign(x.value-y.value))
----------  ----------  ----------  ----------------------------------
1           1           1           0
1           1           0           0
1           1           -1          0
1           0           1           1
1           0           0           0
```

1	0	-1	-1
1	-1	1	1
1	-1	0	0
1	-1	-1	-1
0	1	1	1
0	1	0	0
0	1	-1	-1
0	0	1	0
0	0	0	0
0	0	-1	0
0	-1	1	1
0	-1	0	0
0	-1	-1	-1
-1	1	1	1
-1	1	0	0
-1	1	-1	-1
-1	0	1	1
-1	0	0	0
-1	0	-1	-1
-1	-1	1	0
-1	-1	0	0
-1	-1	-1	0

Modifying the Source: Creating a Permanent Sign Function

Sqlite functions are defined in "./src/func.c". In this file, the name of this function will be "signFunc". The user will call this function in sqlite as sign(n). It will hold only a single variable.

It is helpful to model the sign function after the abs function "absFunc", since they are very similar. If fact, I would highly recommend looking at the abs function any time a new version of sqlite is released.

You will want to follow these steps: First, copy the abs function "absFunc" and make the following changes:

1. Change the function name from absFunc to signFunc.
2. Change the variable iVal. It should equal -1 if sqlite3_value_type(argv[0]) is less than zero. Note that this value is an integer. Otherwise, if this integer is zero, return zero. Or if this integer is greater than zero, return 1. All of this can be expressed simply as follows:

   ```
   iVal = ( iVal > 0) ? 1: ( iVal < 0 ) ? -1: 0;
   ```

3. Perform the same steps above for rVal, which is the real value, as opposed to the integer value above.

   ```
   rVal = ( rVal > 0) ? 1: ( rVal < 0 ) ? -1: 0;
   ```

4. Add the following entry in aFuncs[]:

   ```
   { "sign", 1, 0, SQLITE_UTF8, 0, signFunc },
   ```

5. Recompile sqlite from the main directory and install.

   ```
   $ ./configure
   $ make && make install
   ```

For a closer look, below is the section that changed. Look here for the complete file: func.c.

From ./src/func.c:

```
... cut ...
/*
** Implementation of the sign() function
*/
static void signFunc(sqlite3_context *context, int argc, sqlite3_value **argv){
  assert( argc==1 );
  switch( sqlite3_value_type(argv[0]) ){
    case SQLITE_INTEGER: {
      i64 iVal = sqlite3_value_int64(argv[0]);
 /* 1st change below. Line below was:  if( iVal<0 ) iVal = iVal * -1;     */

      iVal = ( iVal > 0) ? 1: ( iVal < 0 ) ? -1: 0;
      sqlite3_result_int64(context, iVal);
```

```
      break;
    }
    case SQLITE_NULL: {
      sqlite3_result_null(context);
      break;
    }
    default: {
 /* 2nd change below. Line for abs was: if( rVal<0 ) rVal = rVal * -1.0;  */

      double rVal = sqlite3_value_double(argv[0]);
      rVal = ( rVal > 0) ? 1: ( rVal < 0 ) ? -1: 0;
      sqlite3_result_double(context, rVal);
      break;
    }
  }
}
... cut ...

  } aFuncs[] = {
    { "min",               -1, 0, SQLITE_UTF8,    1, minmaxFunc },
    { "min",                0, 0, SQLITE_UTF8,    1, 0          },
    { "max",               -1, 2, SQLITE_UTF8,    1, minmaxFunc },
    { "max",                0, 2, SQLITE_UTF8,    1, 0          },
    { "typeof",             1, 0, SQLITE_UTF8,    0, typeofFunc },
    { "length",             1, 0, SQLITE_UTF8,    0, lengthFunc },
    { "substr",             3, 0, SQLITE_UTF8,    0, substrFunc },
    { "substr",             3, 0, SQLITE_UTF16LE, 0, sqlite3utf16Substr },
    { "abs",               1, 0, SQLITE_UTF8,    0, absFunc    },
    /*  Added here */
    { "sign",              1, 0, SQLITE_UTF8,    0, signFunc   },
    { "round",              1, 0, SQLITE_UTF8,    0, roundFunc  },
    { "round",              2, 0, SQLITE_UTF8,    0, roundFunc  },
... cut ...
```

Using the New Sign Function

Now, back to the problem of creating a pivot table for displaying exam scores in a spreadsheet-like format. First, more data is is needed. By the way, if have not added any data, the following script, enterExamdata, will create the necessary tables and insert the data.

```
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score) values ('Anderson','Bob',3,92)"
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score) values ('Anderson','Bob',4,95)"
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score) values ('Stoppard','Tom',1,88)"
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score) values ('Stoppard','Tom',2,90)"
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score) values ('Stoppard','Tom',3,92)"
$ sqlite3 examdatabase "insert into exam (ln,fn,exam,score) values ('Stoppard','Tom',4,95)"
$ sqlite3 examdatabase2 "insert into exam (ln,fn,exam,score) values ('Carter','Sue',3,99)"
$ sqlite3 examdatabase2 "insert into exam (ln,fn,exam,score) values ('Carter','Sue',4,95)"
```

Below is the select statement for generating a pivot table for four exams on the table exams.

```
  select ln,fn,
    sum(score*(1-abs(sign(exam-1)))) as exam1,
```

```
    sum(score*(1-abs(sign(exam-2)))) as exam2,
    sum(score*(1-abs(sign(exam-3)))) as exam3,
    sum(score*(1-abs(sign(exam-4)))) as exam4
    from exam group by ln,fn;
```

Below is the select statement, like the statement above. However, it works on the virtual table, or the combined exam tables from the databases examdatabase and examdatabase2.

```
$ sqlite3
SQLite version 3.0.8
Enter ".help" for instructions
sqlite> attach database examdatabase as e1;
sqlite> attach database examdatabase2 as e2;
sqlite> .database
seq  name             file
---  ---------------  ----------------------------------------------------------
0    main
2    e1               /work/cpearls/src/posted_on_sf/sqlite_examples/sqlite_exam
3    e2               /work/cpearls/src/posted_on_sf/sqlite_examples/sqlite_exam
sqlite> .header on
sqlite> .mode column
sqlite> select ln,fn,sum(score*(1-abs(sign(exam-1)))) as exam1,
                        sum(score*(1-abs(sign(exam-2)))) as exam2,
                        sum(score*(1-abs(sign(exam-3)))) as exam3,
                        sum(score*(1-abs(sign(exam-4)))) as exam4
        from (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
                        group by ln,fn;

ln          fn          exam1       exam2       exam3       exam4
----------  ----------  ----------  ----------  ----------  ----------
Anderson    Bob         75          82          92          95
Carter      Sue         89          100         99          95
Stoppard    Tom         88          90          92          95

sqlite>
```

Taking a closer look at the results, it's very easy to see that Anderson, Bob got 75 on the first exam, 82 on the second, 92 on the third, and 95 on the forth. Likewise, Stoppard received 88, 90, 92, and 95, respectively.

```
ln          fn          exam1       exam2       exam3       exam4
----------  ----------  ----------  ----------  ----------  ----------
Anderson    Bob         75          82          92          95
Carter      Sue         89          100         99          95
Stoppard    Tom         88          90          92          95
```

Now back to the question of finding the top scores for each exam in one select statement. That is, finding the top scores for the combined tables. First, a look at all the data:

```
$ sqlite3
SQLite version 3.0.8
Enter ".help" for instructions
sqlite> attach database examdatabase as e1;
sqlite> attach database examdatabase2 as e2;
sqlite> .header on
```

```
sqlite> .mode column
sqlite> select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam;
db          ekey        fn          ln          exam        score       timeEnter
----------  ----------  ----------  ----------  ----------  ----------  -------------------
e1          1           Bob         Anderson    1           75          2004-10-17 22:01:42
e1          2           Bob         Anderson    2           82          2004-10-17 22:02:19
e1          3           Bob         Anderson    3           92          2004-10-17 22:05:04
e1          4           Bob         Anderson    4           95          2004-10-17 22:05:16
e1          5           Tom         Stoppard    1           88          2004-10-17 22:05:24
e1          6           Tom         Stoppard    2           90          2004-10-17 22:05:31
e1          7           Tom         Stoppard    3           92          2004-10-17 22:05:40
e1          8           Tom         Stoppard    4           95          2004-10-17 22:05:50
e2          1           Sue         Carter      1           89          2004-10-17 22:03:10
e2          2           Sue         Carter      2           100         2004-10-17 22:03:10
e2          3           Sue         Carter      3           99          2004-10-17 22:05:57
e2          4           Sue         Carter      4           95          2004-10-17 22:06:05
sqlite>
```

Below, continuing with the same attached setup, is an example of horizontal averages and horizontal maximum values.

```
sqlite> .headers on
sqlite> .mode column
sqlite> select db,ln as lastname,fn as first,
                sum(score*(1-abs(sign(exam-1)))) as exam1,
                sum(score*(1-abs(sign(exam-2)))) as exam2,
                sum(score*(1-abs(sign(exam-3)))) as exam3,
                sum(score*(1-abs(sign(exam-4)))) as exam4,
                avg(score) as avg, max(score) as max
   from (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
   group by ln,fn,db ;


db          lastname    first       exam1       exam2       exam3       exam4       avg         max
----------  ----------  ----------  ----------  ----------  ----------  ----------  ----------  ----------
e1          Anderson    Bob         75          82          92          95          86          95
e2          Carter      Sue         89          100         99          95          95.75       100
e1          Stoppard    Tom         88          90          92          95          91.25       95
```

Try finding the deltas, or the differences between each exam score. For hints on this, see the end of this article in the LONGWINDED TIPS section.

Pivot Table "Spreadsheet Format" to Normalized Data

Consider the reverse: Suppose you had a pivot table, or the data in a spreadsheet-like format, and you wanted a normalized table of exams. For this example, the table nonormal is needed. This table is defined and created as follows:

```
SQLite version 3.0.8
Enter ".help" for instructions
sqlite> attach database 'examdatabase' as e1;
sqlite> attach database 'examdatabase2' as e2;
sqlite> create table e1.nonormal as
```

```
    select ln,fn,
        sum(score*(1-abs(sign(exam-1)))) as exam1,
        sum(score*(1-abs(sign(exam-2)))) as exam2,
        sum(score*(1-abs(sign(exam-3)))) as exam3,
        sum(score*(1-abs(sign(exam-4)))) as exam4
 from (select 'e1' as db,* from e1.exam union select 'e2' as db,* from e2.exam)
        group by ln,fn;

sqlite> .header on
sqlite> .mode column
sqlite> select * from e1.nonormal;
ln          fn          exam1       exam2       exam3       exam4
----------  ----------  ----------  ----------  ----------  ----------
Anderson    Bob         75          82          92          95
Carter      Sue         89          100         99          95
Stoppard    Tom         88          90          92          95
```

The nonormal table was created in the examdatabase, since "e1." was given before the name. Again, the objective here is to go backwards and create a normalized table from the pivot table, a table that will list all exam scores in one field and all the exam numbers in another, without having a separate field for each exam. In addition, the goal is to do all this in one select statement without looping through the data. First, it is necessary to create a number table, "enum", and it must have the field "e" from 1..N where N is the number of exams (which is four in this case).

```
sqlite> CREATE TABLE enum (e int);
sqlite> INSERT INTO "enum" VALUES(1);
sqlite> INSERT INTO "enum" VALUES(2);
sqlite> INSERT INTO "enum" VALUES(3);
sqlite> INSERT INTO "enum" VALUES(4);
```

The coalesce function is used in an interesting way for this example.

```
sqlite> .mode list
sqlite> select n.ln,n.fn,

  1*(1-abs(sign(e.e-1)))+
  2*(1-abs(sign(e.e-2)))+
  3*(1-abs(sign(e.e-3)))+
  4*(1-abs(sign(e.e-4))),

 coalesce(0/(e.e-1),n.exam1)+
 coalesce(0/(e.e-2),n.exam2)+
 coalesce(0/(e.e-3),n.exam3)+
 coalesce(0/(e.e-4),n.exam4)

 from enum as e,e1.nonormal as n;


Anderson|Bob|1|75
Carter|Sue|1|89
Stoppard|Tom|1|88
Anderson|Bob|2|82
Carter|Sue|2|100
```

```
Stoppard|Tom|2|90
Anderson|Bob|3|92
Carter|Sue|3|99
Stoppard|Tom|3|92
Anderson|Bob|4|95
Carter|Sue|4|95
```

For more examples, see this article.

Max Min Problems

Assume you have the following table of names, ages, and salaries. Find the age, name, and salary of the youngest person making the overall highest salary, or first find the highest salary, then, from this group, select the youngest person.

```
create table salary (name varchar(3),age int, salary double);
insert into salary values ('dan',23,67);
insert into salary values ('bob',45,94);
insert into salary values ('tom',24,94);
insert into salary values ('sue',23,45);
insert into salary values ('joe',45,51);
insert into salary values ('sam',22,51);
```

Once you have the data entered, you will have the following;

```
sqlite> .headers on
sqlite> .mode column
sqlite> select * from salary;
name        age         salary
----------  ----------  ----------
dan         23          67
bob         45          94
tom         24          94
sue         23          45
joe         45          51
sam         22          51
sqlite>
```

The following select will give you the youngest person making the top salary in the company:

```
sqlite>  select 1000-max(salary*1000-age)%1000 from salary;

1000-max(salary*1000-age)%1000
------------------------------
24
```

This is the correct answer. The highest salary is 94 for Bob and Tom. Tom is the youngest at 24.

Why the number 1000? Well, no one lives to be 1000, so we know that age will never be >= 1000. Therefore, max(salary*1000-age) will clearly choose the highest salary independent of age, as long as salary is >= 1. In cases of a tie in salary, the youngest person will subtract the least amount from the salary, so this value will return as the highest. It's easy to remove the salary part from this number. Since salary is multiplied by 1000, it will disappear with mod 1000,

since it's a perfect factor of 1000.

To understand how this works, it is helpful to break the statement into separate, smaller parts, as follows:

```
sqlite> select salary*1000-age,salary*1000,-age from salary;
salary*1000-age  salary*1000  -age
---------------  -----------  ----------
66977            67000        -23
93955            94000        -45
93976            94000        -24
44977            45000        -23
50955            51000        -45
50978            51000        -22
sqlite>
```

But what about the negative value for age? With the non-Knuth method of the mod function, "%", when x < 0, then x % y will return x, if abs(x) < abs(y).

x % y is defined as follows:

```
x % y ==  x - INT(x/y)*y
```

and undefined for y == 0. C and Fortran use this method.

In contrast, the Knuth method, found in Python and accepted in mathematics, defines this function as follows:

```
x mod y == x - floor(x/y),
```

and equal to x if y == 0

The difference between the two shows up with negative values for x.

Or, put another way, as long as -x != y, then -x % y = -x. For example, assume x=4 and y=5, then -4 % 5 will return a -4. Here are a few other examples. Again this is *not* the Knuth method for the mod function.

```
-1 % 5 = -1
-2 % 5 = -2
-3 % 5 = -3
```

So what we are really doing is the following:

```
1000 +  -1*(1000-age) = age
```

C and C++ API

Simple C Program

The following is a simple C program, simplesqlite3.c, which will open a database and execute a SQL string.

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>


static int callback(void *NotUsed, int argc, char **argv, char **azColName){
  NotUsed=0;
  int i;
  for(i=0; i<argc; i++){
    printf("%s = %s\n", azColName[i], argv[i] ? argv[i]: "NULL");
  }
  printf("\n");
  return 0;
}

int main(int argc, char **argv){
  sqlite3 *db;
  char *zErrMsg = 0;
  int rc;

  if( argc!=3 ){
    fprintf(stderr, "Usage: %s DATABASE SQL-STATEMENT\n", argv[0]);
    exit(1);
  }
  rc = sqlite3_open(argv[1], &db);
  if( rc ){
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);
    exit(1);
  }
  rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
  if( rc!=SQLITE_OK ){
    fprintf(stderr, "SQL error: %s\n", zErrMsg);
  }
  sqlite3_close(db);
  return 0;
}
```

The command to compile and run the program is shown below. Note the "-Wl,-R/usr/local/lib" options, which will be needed if you installed the sqlite3 source, since the path "/usr/local/lib" may not be listed in your "/etc/ld.so.conf" file.

```
gcc -o simplesqlite3 simplesqlite3.c  -Wall -W -O2 -Wl,-R/usr/local/lib -lsqlite3
```

You either have to use the compile option above or add the directory where the sqlite3 library

"libsqlite3.so" is installed to the file "/etc/ld.so.conf", then run ldconfig from the shell. I prefer to use the "-Wl,-R" option instead, but there are the steps.

```
$ locate libsqlite3.so
/usr/local/lib/libsqlite3.so.0.8.6
/usr/local/lib/libsqlite3.so.0
/usr/local/lib/libsqlite3.so    <--- note directory is /usr/local/lib

$ echo "/usr/local/lib" >> /etc/ld.so.conf
$ ldconfig
```

After you have entered and compiled the program, it will run as follows:

```
$ ./simplesqlite3 test.db "create table notes (t text)"

$ ./simplesqlite3 test.db "insert into  notes (t) values ('
> This is some random
> stuff to add'
>);"

$ ./simplesqlite3 test.db "select * from notes"

t =
This is some random
stuff to add
```

There are really only three important statements, sqlite3_open(), which takes the name of the database and a database pointer, sqlite3_exec(), which executes the SQL commands in argv[2] and lists the callback function used to display the results, and sqlite3_close(), which closes the database connection.

A C++ Program -- Building a Class to Do the Work

It is possible to build a class, SQLITE3 (defined below), which reads the returned data into a vector. Note that instead of using the sqlite3_exec() function, sqlite3_get_table() is used instead. It copies the result of the SQL statement into the variable array of string result. Note this variable must be freed with sqlite3_free_table() after it has been used to copy the returned SQL headings and data into the vectors vcol_head and vdata. Note that the first row is the heading.

```
class SQLITE3 {
private:
  sqlite3 *db;
  char *zErrMsg;
  char **result;
  int rc;
  int nrow,ncol;
  int db_open;

public:

  std::vector<std::string> vcol_head;
  std::vector<std::string> vdata;

  SQLITE3 (std::string tablename="init.db"): zErrMsg(0), rc(0),db_open(0) {
```

```
    rc = sqlite3_open(tablename.c_str(), &db);
    if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      sqlite3_close(db);
    }
    db_open=1;
  }

  int exe(std::string s_exe) {
      rc = sqlite3_get_table(
                  db,             /* An open database */
                  s_exe.c_str(),       /* SQL to be executed */
                  &result,       /* Result written to a char *[]  that this points to */
                  &nrow,             /* Number of result rows written here */
                  &ncol,          /* Number of result columns written here */
                  &zErrMsg          /* Error msg written here */
                  );

      if(vcol_head.size()<0) { vcol_head.clear();  }
      if(vdata.size()<0)     { vdata.clear(); }


     if( rc == SQLITE_OK ){
      for(int i=0; i < ncol; ++i)
      vcol_head.push_back(result[i]);   /* First row heading */
      for(int i=0; i < ncol*nrow; ++i)
      vdata.push_back(result[ncol+i]);
     }
     sqlite3_free_table(result);
      return rc;
  }

  ~SQLITE3(){
      sqlite3_close(db);
  }
};
```

The complete program can be found in this example or viewed here: simplesqlite3cpp2.cc.

With the class defined above, it can be used in main or in a function as follows:

```
std::string s;
SQLITE3 sql("database.db");


sql.exe("create table notes (t text)");
s="insert into notes (t) values ('sample data')";
sql.exe(s);

s="select t from notes";
sql.exe(s);
```

The following, still assuming this code will be entered in main or a function, is an example of printing the data from a select. Note the headings section and the data sections.

```
if( sql.vcol_head.size() > 0 )
  {
    std::cout << "Headings" << std::endl;
    copy(sql.vcol_head.begin(),
            sql.vcol_head.end(),
            std::ostream_iterator<std::string>(std::cout,"\t"));

    std::cout << std::endl << std::endl;
    std::cout << "Data" << std::endl;
    copy(sql.vdata.begin(),
            sql.vdata.end(),
             std::ostream_iterator<std::string>(std::cout,"\t"));

    std::cout << std::endl;
  }
```

Defining SQLite User Functions

There are two types of functions, aggregate functions and simple functions. Simple functions like sign(), which was created above, can be used in any expression. Aggregate functions like "avg()" can only be used in the select statement. Some functions like "min" and "max" can be defined as both. min() with 1 argument is an aggregate function, whereas min() with an unlimited number of arguments is a simple function.

Here is an example which illustrates the difference:

```
$ sqlite3
SQLite version 3.0.8
Enter ".help" for instructions
sqlite> create table junk (a integer);
sqlite> insert into junk (a) values (1);
sqlite> insert into junk (a) values (2);
sqlite> insert into junk (a) values (3);
sqlite> select * from junk;
1
2
3

sqlite> select * from junk where a=min(1,2,3,4,5);
1
sqlite> select * from junk where a=min(1);
SQL error: misuse of aggregate function min()
sqlite>
```

Note above that the min() function, with only one variable, is an aggregate function. Since it is only an aggregate function, it cannot be used after the where clause. An aggregate function can only be used after the select clause as follows:

```
sqlite> select min(a) from junk
1
```

If you add a second argument, you're calling the simple function. Note below that each row is compared to 2.3. Look closely; there is a subtle but important difference here.

```
sqlite> select min(a,2.3) from junk
2
2.3
1
```

Creating a User-defined Sign Function: msign

Below is an example of the sign function. It is called msignFunc so as not to interfere with the permanent sign function that was created earlier.

```
void msignFunc(sqlite3_context *context, int argc, sqlite3_value **argv){
  assert( argc==1 );
  switch( sqlite3_value_type(argv[0]) ){
    case SQLITE_INTEGER: {
      long long int iVal = sqlite3_value_int64(argv[0]);
      iVal = ( iVal > 0) ? 1: ( iVal < 0 ) ? -1: 0;
      sqlite3_result_int64(context, iVal);
      break;
    }
    case SQLITE_NULL: {
      sqlite3_result_null(context);
      break;
    }
    default: {
      double rVal = sqlite3_value_double(argv[0]);
      rVal = ( rVal > 0) ? 1: ( rVal < 0 ) ? -1: 0;
      sqlite3_result_double(context, rVal);
      break;
    }
  }
}
```

This function is initiated as follows:

```
sqlite3_create_function(db, "msign", 1, SQLITE_UTF8, NULL,
                &msignFunc, NULL, NULL));
```

Note that "msign" is the name of the function in sqlite3. It is the name you would use in a select statement: "select msign(3);". The 1 is the number of arguments. The msign function here only takes one argument. SQLITE_UTF8 is for the text representation. Then, skipping over NULL, & msignFunc is the name of the C function. The last two values must be NULL for a simple function; again, a simple function can be used in any part of the select where clause.

Aggregate Functions

A good place to look for ideas on creating functions is the ./src/func.c file in the sqlite3 source. Suppose you would like to create a new sum function call S. It will create the aggregate sum of the rows.

The following data is used to explain this function.

```
$ ./myfuncpp DATABASE "create table t(a integer, b integer, c integer)"
```

```
$ ./myfuncpp DATABASE "insert into t (a,b,c) values (1,-1,2)"
$ ./myfuncpp DATABASE "insert into t (a,b,c) values (2,-2,4)"
$ ./myfuncpp DATABASE "insert into t (a,b,c) values (3,-3,8)"
$ ./myfuncpp DATABASE "insert into t (a,b,c) values (4,-4,16)"
$ ./myfuncpp DATABASE "select * from t"
a = 1 b = -1 c = 2
a = 2 b = -2 c = 4
a = 3 b = -3 c = 8
a = 4 b = -4 c = 16
a = 4 b = -4 c = 16
```

Now for how the function S will create a list of the sums. Unlike the standard aggregate sum() function, a list is returned.

```
$ ./myfuncpp DATABASE "select S(a),S(b),S(c) from t"
S(a) = (1,3,6,10,14) S(b) = (-1,-3,-6,-10,-14) S(c) = (2,6,14,30,46)
```

Note that a column which contains the values in the table (1,2,3,4,5) shows the cumulative sum (1,1+2=3,1+2+3=6,1+..) in a list. This is different from any function defined in ./src/func.c, since the data must be in a string.

To view this example and all other examples, see the download. Since a list is returned, this example will use the C++ std::stringstream, since this is fast and well suited for all types of data, integer, double, and text.

Since S is an aggregate function, there are two functions, "SStep" and "SFinalize". Aggregate functions always have a "Step" and a "Finalize". "Step" is called for each row, and after the last row, the "Finalize" function is called.

Both the Step and the Finalize can make use of a structure for holding the cumulated data collected from each row. For this function, the structure SCtx is defined below. std::stringstream is global. I would not advise putting an additional variable in SCtx, "char *ss". You may think that this could be dynamically increased with realloc, which will work. However, the problem is freeing the memory hanging off the structure. There's a bit of confusion here. As the sqlite documentation correctly points out, the structure SCtx will be freed; but again, in my testing, any additional memory allocated off members in the structure will not. On the other hand, an array of std:stringstrings "BS" will have to be kept for when this function is called more than once in the same select "select S(a),S(b),...S(100th)" The overhead appears minimal.

```
#define MAXSSC 100

typedef struct SCtx SCtx;
struct SCtx {
  double sum;     /* Sum of terms */
  int cnt;        /* Number of elements summed */
  int sscnt;      /* Keeps counts for ss */
};

std::stringstream ss[MAXSSC];
int sscnt=0;
```

Below is the step function. p gets initialized the first time through SStep. On the first pass, all the values in the SCtx structure will be zeroed. This is a feature of "sqlite3_aggregate_context".

Since std::stringstream s0 is defined as a global variable, care will have to be taken to ensure that when S is called in the same select "Select S(a),S(b)...", S(a) does not use S(b)'s stringstream.

```
static void SStep(sqlite3_context *context, int argc, sqlite3_value **argv){
  SCtx *p=NULL;
  int i;

  std::string d;
  if( argc<1 ) return;
  p = (SCtx *) sqlite3_aggregate_context(context, sizeof(*p));
  if( p->cnt == 0)    /* When zero first time through */
    {
      if ( sscnt  >= MAXSSC )
        { fprintf(stderr,"MAXSSC needs to increase\n");
          exit(1);
        }
      p->sscnt=sscnt;
      sscnt++;
      ss[p->sscnt].str("");
      ss[p->sscnt] << "(";
     d="";
    } else {
       d=",";

    }

    p->sum += sqlite3_value_double(argv[0]);
    p->cnt++;
    ss[p->sscnt] << d <<  p->sum ;


/*
 *      If the simple function is not used, this
 *      comes into play.
 */
    if (p->cnt == 1)
      {
        for(i=1; i< argc; ++i) {
          p->cnt++;
          p->sum+=sqlite3_value_double(argv[i]);
          ss[p->sscnt] <<  "," << p->sum ;
          }

      }

}
```

The line:

```
p = (SCtx *) sqlite3_aggregate_context(context, sizeof(*p));
```

will initialize p->sum, p->cnt, and p->sscnt to zero on the first entry into this function. On each successive entry, the old values be passed back. Although the std:stringstream ss variable is

global, S(a) called in the select uses ss[0], S(b) will use ss[1], etc.

Also note the comment "*If* the simple function is not used, this comes into play". Below that statement, i walks through the argument count. It is possible to have a function name "S", in this case defined as both an aggregate function and a simple function. The distinction is made with the number of arguments in the calling function. This is set in sqlite3_create_function. For example, a name could be assigned to a simple function and an aggregate function. Normally, this is set up so that the simple function takes two or more arguments max(1,2,3,4,5) and the aggregate function just takes one argument max(a). Take a look at max in ./src/func.c.

Here is the Finalize function:

```
static void SFinalize(sqlite3_context *context){
  SCtx *p=NULL;
  char *buf=NULL;
  p = (SCtx *) sqlite3_aggregate_context(context, sizeof(*p));

  ss[p->sscnt] << ")";
  buf = (char *) malloc (sizeof(char)*(ss[p->sscnt].str().size()+1));
  if (buf == NULL)
    fprintf(stderr,"malloc error in SNFinalize, buf\n");



  snprintf(buf,ss[p->sscnt].str().size(),"%s",ss[p->sscnt].str().c_str());
  sqlite3_result_text(context,buf,ss[p->sscnt].str().size()+1,free );
  sscnt--;    /* reclaim this stream */

}
```

After all the rows in select have gone through the "SStep" function, the "SFinalize" function is called. The last value for the SCTx structure is assigned to p in the statement "p = (SCtx *) sqlite3_aggregate_context(context, sizeof(*p));". Note that p->sscnt is needed for indexing the correct ss. The proper memory size is allocated using +1 in ss[p->sscnt].str().size()+1, to allow for the NUL character. sqlite3_result_text takes care of freeing the memory allocated for buf.

The user functions "SStep" and "SFinalize" need to be added to the SQL language interpreter. This is done with the "sqlite3_create_function":

```
 if (sqlite3_create_function(db, "S", 1, SQLITE_UTF8, NULL, NULL, &SStep,
                        &SFinalize) != 0)
   fprintf(stderr,"Problem with S using SStep and SFinalize\n");
```

Note the 1 for the third argument. This aggregate function is used when one argument is passed. To have it both ways, to have "S" defined as both an aggregate and a simple function, an SFunc would have to be created. That could handle 2 to N variables. Once this function is created, the additional "sqlite3_create_function" would be defined in main as follows:

```
... still in main
 if (sqlite3_create_function(db, "S", -1, SQLITE_UTF8, NULL, &SFunc, NULL,
                        NULL) != 0)
   fprintf(stderr,"Problem with S using SFunc -- simple function\n");
```

Here is an example SFunc function:

```
static void SFunc(sqlite3_context *context, int argc, sqlite3_value **argv){
  std::stringstream s;
  std::string d;
  double sum=0;
  char *buf=NULL;
  int i;

  s.str("");

  s << "(";
  d="";
  for(i=0; i < argc; i++)
   {
  switch( sqlite3_value_type(argv[i]) ){
    case SQLITE_INTEGER: {
      sum+=(double) sqlite3_value_int64(argv[i]);
      s << d << sum;
      d=",";
      break;
    }
    case SQLITE_NULL: {
      s << d << "()";
      d=",";
      break;
    }
    default: {
      sum+=sqlite3_value_int64(argv[i]);
      s << d <<  sum;
      d=",";
      break;
     }
    }

    }

  s << ")";
  buf = (char *) malloc (sizeof(char)*(s.str().size()+2));
  if (buf == NULL)
    fprintf(stderr,"malloc error in SNFunc, buf\n");
  snprintf(buf,s.str().size()+1,"%s",s.str().c_str());
  sqlite3_result_text(context,buf,s.str().size()+1,free );

}
```

Now, S works as both a simple function and an aggregate function. The simple function can go in any expression, but the aggregate only works after the select. Hence, this goes back to the power of the sign function, which is a simple function.

```
./myfuncpp DATABASE 'select S(1,2,3,4)'
S(1,2,3,4) = (1,3,6,10)
```

For a few more examples, take a look at myfuncpp.cc in the download. There are some

interesting functions there. For instance, there is an I or index function that works as follows:

```
$ ./myfuncpp DATABASE "select S(1,2,3)"
S(1,2,3) = (1,3,6)

$ ./myfuncpp DATABASE "select I(S(1,2,3),0)"
I(S(1,2,3),0) = 1

$ ./myfuncpp DATABASE "select I(S(1,2,3),1)"
I(S(1,2,3),1) = 3
```

, which takes the index in the list. The first index starts at zero.

Reading Images (Blob data)

First, a demonstration of how the program eatblob.c works. This program is a C API which inserts binary (blob) data into a table.

The program can be run in two ways. First, script commands can be redirected into it. For example, you can create the following script file "sqlcommands":

```
$ cat sqlcommands

  create table blobtest (des varchar(80),b blob);
     insert into blobtest (des,b)
            values ('A test file: test.png',?);
     select * from  blobtest;
```

Note the "?" on the line "values ('A test file: test.png',?);". This serves as a place holder for blob data in the SQL statement. Using this file, the program is executed as follows:

```
$ ./eatblob test3.db test.png < sqlcommands
```

The image file "test.png" will be read into the program and inserted into the field b, since this is where the question mark is placed.

The program also works interactively, as follows:

```
$ ./eatblob test.db test.png
eatblob:0> create table blobtest2 (des varchar(30), b blob);
eatblob:0> insert into blobtest2 (des,b) values ('A second test: test.png',?);
eatblob:1> insert into blobtest2 (des,b) values ('A third test: test.png',?);
eatblob:2> select * from blobtest2;
A second test: test.png
A third test: test.png
eatblob:2> .q
[chirico@third-fl-71 sqlite_examples]$ ls outdata.*
outdata.0.png  outdata.1.png
```

The blob data is not shown. Instead, it is written to the file outdata.n.png, where n is the record number.

Examining the C code in eatblob.c

The program works by reading all of the binary data from the filename given as the third argument to the command. The complete file is read into memory. One way to do this is to get the total file size first, then allocate that amount of memory with malloc. That approach is not taken here. Instead, a more general approach is used. For instance, if you were to read data from a socket, you may not know beforehand how big the file will be. This general approach will take advantage of the realloc function. The function in the program addmem will give us a number. The number will be the new number of units to reallocate. We want to increase the amount of memory in a non-linear fashion to minimize the number of reallocations for large files.

```
        #define MULT_START     16384
        #define MULT_INCREMENT 2
        #define INIT_SIZE      1024

        long memindx = MULT_START;
        long memnext;
...

    1     long addmem(char **buf, long size)
    2     {
    3         memnext = (size > 0) ? size + memindx: INIT_SIZE;
    4         memindx = memindx * MULT_INCREMENT;
    5         char *tbuf = realloc(*buf, memnext);
    6         if (tbuf == NULL) {
    7                 fprintf(stderr, "Can't allocate memory in addmem\n");
    8                 return size;
    9         } else {
   10                  *buf = tbuf;
   11                  return memnext;
   12          }
   13       }

...
```

One line 3, the first time this function is called, the variable size is the current number of bytes allocated. If no memory has been allocated (size is 0), the new size will be INIT_SIZE. For this program, INIT_SIZE is set to 1024 in the define statement. However, if size is greater than zero, the new size will be the initial size plus memindx, which starts at 1024.

```
  Call   memidx       Number Returned           size
    1     1024         1024 =     0 +   1024        0
    2     2048         3072 =  1024 + (1024*2)    1024
    3     4096         7168 =  3072 + (2048*2)    3072
    4     8192        15360 =  7168 + (4096*2)    7168
    5    16384        31744 = 15360 + (8192*2)   15360
  ...
```

As you can see, the number returned increases exponentially. "memindx" is doubled each time this function is called. This doubled value is added to the size.

```
    1     long addmem(char **buf, long size)
    2     {
    3         memnext = (size > 0) ? size + memindx: INIT_SIZE;
```

```
 4          memindx = memindx * MULT_INCREMENT;
 5          char *tbuf = realloc(*buf, memnext);
 6          if (tbuf == NULL) {
 7                  fprintf(stderr, "Can't allocate memory in addmem\n");
 8                  return size;
 9          } else {
10                   *buf = tbuf;
11                   return memnext;
12           }
13       }
```

...

So, the function gives us a number that we can pass to realloc.

realloc works as follows: If realloc is successful, it will copy the contents pointed to by buf to a location of memory with the larger size memnext, then free the old region of memory. This new region of memory will be assigned to tbuf. Since the old location (the location pointed to by *buf) has been released, we need to assign the new value to *buf (*buf = tbuf).

If realloc cannot get the new size memnext, *buf is left untouched and tbuf will be null.

Note that in the above program, buf is a pointer to a pointer, *buf is a pointer, and **buf is the first byte of data.

Perl and sqlite3

To use Perl with sqlite3, DBI and DBD::SQLite must be installed. To install the packages from CPAN, use the following commands.

```
# perl -MCPAN -e shell
cpan> install DBI
cpan> install DBD::SQLite
```

The following program will create a database and enter records:

```
#!/usr/bin/perl

use DBI;

$dbh = DBI->connect( "dbi:SQLite:data.dbl" ) || die "Cannot connect: $DBI::errstr";

$dbh->do( "CREATE TABLE authors ( lastname, firstname )" );
$dbh->do( "INSERT INTO authors VALUES ( 'Conway', 'Damian' ) " );
$dbh->do( "INSERT INTO authors VALUES ( 'Booch', 'Grady' ) " );
$dbh->do( "CREATE TABLE books ( title, author )" );
$dbh->do( "INSERT INTO books VALUES ( 'Object Oriented Perl',
                                      'Conway' ) " );
$dbh->do( "INSERT INTO books VALUES ( 'Object-Oriented Analysis and Design',
                                      'Booch' ) ");
$dbh->do( "INSERT INTO books VALUES ( 'Object Solutions', 'Booch' ) " );


$res = $dbh->selectall_arrayref( q( SELECT a.lastname, a.firstname, b.title
                                    FROM books b, authors a
                                    WHERE b.title like '%Orient%'
                                  AND a.lastname = b.author ) );

foreach( @$res ) {
 foreach $i (0..$#$_) {
    print "$_->[$i] "
    }
 print "\n";

}

$dbh->disconnect;
```

For a more elaborate Perl example that defines functions, see perlExample.pl in the download.

Also, consider using the Perl Debugger, for stepping through complex Perl sqlite programs where you are not sure of what is returned. To get into the Perl debugger, execute the following command, and to get out of the Perl Debugger type "q".

```
$ perl -de 42
```

A Simple Everyday Application -- Keeping Notes in a Database

This simple bash script allows you to take notes. The notes consist of a line of text followed by an optional category. It doesn't require you to type "sqlite3 <database> <sql statement>". Instead, you just need a simple one-letter command:

```
$ n 'Take a look at sqlite3 transactions -
    http://www.sqlite.org/lang.html#transaction' 'sqlite3'
```

This enters the text into a notes table under the category "sqlite3". Whenever a second field appears, it is considered the category. To extract records for the day, enter "n -l" (which is similar to "l -l") to "note list".

With just "n", help is listed for all the commands.

```
$ n
This command is used to list notes in
a database.

n <option>
 -l list all notes
 -t list notes for today
 -c list categories
 -f <search string> search for text
 -e <cmd> execute command and add to notes
 -d delete last entry
```

References

Over 90 Linux Tips
: See TIP 50 on working with the libraries in C and C++. This tip details how to create dynamic and static libraries and make use of the -Wl and -R switches in gcc

Solving Complex SQL Problems
: This is a list of examples using the sign function.

http://www.sqlite.org/
: The homepage for the SQLite project.

Lemon Parser Generator Tutorial
: A tutorial on the parser used with sqlite.

<

Mike Chirico, a father of triplets (all girls) lives outside of Philadelphia, PA, USA. He has worked with Linux since 1996, has a Masters in Computer Science and Mathematics from Villanova University, and has worked in computer-related jobs from Wall Street to the University of Pennsylvania. His hero is Paul Erdos, a brilliant number theorist who was known for his open collaboration with others.

Mike's notes page is souptonuts. For open source consulting needs, please send an email to mailto:mchirico@comcast.net?subject=Open Source Consulting Needs. All consulting work must include a donation to SourceForge.net.