SQLite—A Standalone, Embeddable Database Engine

Albert Danial

LAMPsig Meeting

May 21, 2005

Overview of SQLite in Three Parts

- Introduction to SQL
- SQLite
  - Features
  - Comparison to traditional SQL engines
  - Installation
- Demos
  - Unix command line & shell scripting
  - C interface
  - Perl interface
  - SQL with a real database

SQL = Structured Query Language

- A computer language for creating and extracting data from a “database” in a relational manner
- SQL is interpreted by SQL engines or programs. Examples: Oracle, DB2, Sybase, MySQL, Postgres, Firebird, Microsoft SQL Server, SQLite
- Initial concept by E.F. Codd, IBM, 1970 (but Oracle was first to market)
- ANSI standard since 1986
- My opinion: incomplete as a computer language (still need Perl, PHP, Python, C, tcl, etc. to drive SQL)
- While incomplete as a language, SQL can be incredibly complex; take years to master (and I'm <u>not</u> a master!)

Why SQL?

- SQL came into being to solve problems in business applications.
- Wade through large collections of data and extract useful information.
- SQL is a complicated solution.
- If you can get away with it, a flat file + tools like `grep` & `wc` go a long way.
- If you need to write code (Perl, PHP, Python, C) to extract useful information, SQL may help.
- SQLite minimizes the start-up headaches involved with an SQL solution.

SQL Basic Concepts

- Information is stored in tables.
- You define the number of columns in a table and the kind of data that each column can hold.
- SQL terminology: a column is called a field.
- A database has one or more related tables.
- SQL commands to
  - Insert rows into tables
  - Extract rows from tables
  - Create/delete entire tables
  - Manage user access & permissions (n/a to SQLite)

SQL Basic Concepts, continued

- You have to decide how many tables to create, and which fields to put in the tables.
- A given collection of tables and their fields is known as a database <u>schema</u>.
- Much thought needs to go into the design of the database schema. Goals:
  - A piece of information only appears once.
  - Queries (data extraction) are easy.
  - Queries are fast.

SQL Schema Example

- You want to archive your family's recipes.
- You want to perform searches that answer questions like:
  - Which dessert dishes require four eggs?
  - Is there an Italian appetizer that includes raisins?
  - What should I put on my shopping list to make French bread, a cheesecake, and chocolate chip cookies?

Storing a Cookbook

- A recipe has these attributes:
  - Has a name.
  - Belongs to one or more categories (Italian, dessert, sea food, etc.).
  - Has one or more ingredients.
  - Has preparation instructions, yield, preparation duration, nutritional information, source, rating, et cetera.
- Storage options:
  - text file + use an editor to search for recipes.
  - text file + write a program to do searches.
  - Spreadsheet + pivot tables (?) + macros (?)
  - SQL database

A naïve SQL Schema

Table RECIPE

name, cat1, cat2, cat3, yield, ingr1, ingr2, ingr3, ingr4, .., ingr20, prep

A single table with 26 fields, each of which is a string.

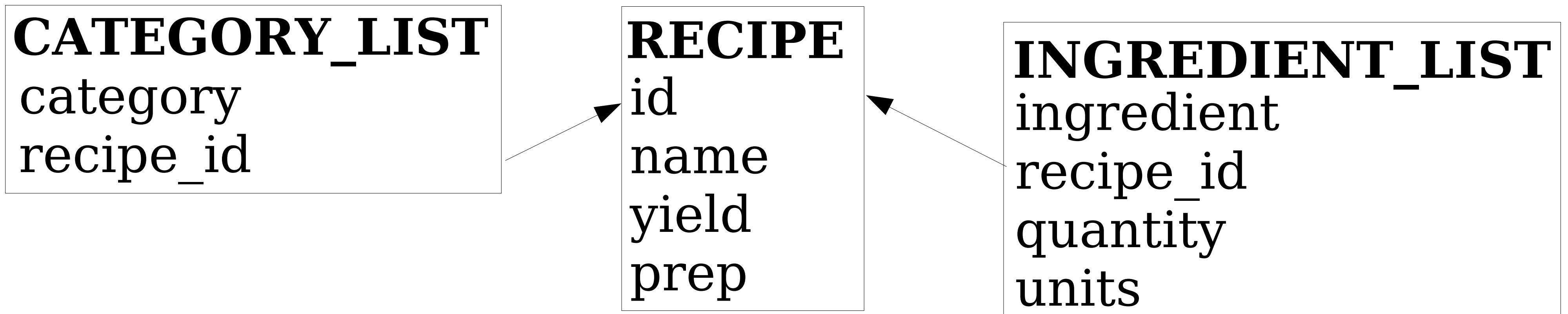
Identical to a comma separated value data loaded into a spreadsheet.

Severe Limitations:
- hardcoded maximum number of categories (3 here)
- hardcoded maximum number of ingredients (20 here)

A Better Schema

Separate tables for

- food categories
- ingredients
- instructions



The arrows denote keys (compatible fields in different tables).

Sample Data

Table RECIPE

id,name,yield,prep

```
27, "burgers"  , 4, "Heat grill..."
28, "apple pie", 6, "Peel apples..."
29, "omelettes", 2, "whisk eggs..."
```

Table CATEGORY_LIST

category, recipe_id

```
"American",  27
"barbeque",  27
"dessert",   28
"picnic",    27
"breakfast", 29
"brunch",    29
```

Table INGREDIENT_LIST

ingred., recipe_id, quant., units

```
"Ground beef"  , 27,  1,  lb
"burger buns"  , 27,  4,
"ketchup"      , 27, 20,  oz
"mustard"      , 27, 10,  oz
"apples"       , 28,  3,  lb
"flour"        , 28,  1,  cup
"onions"       , 29,  1,
"onions"       , 27,  1,
"ketchup"      , 29,  5,  oz
"sugar"        , 28,  1,  cup
"moz. Cheese"  , 29,  8,  oz
```

The “Better” Schema is also Deficient

We are violating data duplication in two places:

The same category can appear many times in the category_list table.

The same ingredients can appear many times in the ingredient_list table

Joins

- The fields

  ```
  recipe.id
  ingredient_list.recipe_id
  category_list.recipe_id
  ```

  are keys which link the tables together.
- An SQL query that establishes a relationship between tables by equating their keys is called a join.
- Example:

```
select name,ingredient from recipe, ingredient_list
    where recipe.name = “apple pie” and
         recipe.id = ingredient_list.recipe_id;
```

SQLite http://www.sqlite.org/

- A command line tool which implements an SQL engine and a C library you can link your code to.
- The engine is easy to use.
- The engine is fast.
- Implements most of ANSI 92 standard.
- ACID compliant
- Stand-alone (or embedded), not client/server.
- Entire database stored in one file.
- Public domain license.

ACID?

- Atomic—database transactions are done “all or nothing”. If a failure happens while processing, the transaction will be rolled back.
- Consistent—database will not insert bogus data.
- Isolated—transactions that arrive simultaneously are queued up, executed sequentially so transactions don't conflict.
- Durable—a transaction that has been committed won't be lost.

Comparison to Client/Server DB's

- Oracle/Postgres/MySQL/MS SQL/...
  - Need a database daemon (a database server program running somewhere).
  - Considerable effort to install, set up.
  - Security issue w/open ports.
  - Database client can be on remote computer.
  - Have to “dump” database to a file to relocate it, back it up.
  - Easily handle simultaneous users.
  - MySQL/Postgres performance at best equals SQLite; mostly slower.

- SQLite
  - No daemon.
  - Easy to install; trivial to create database.
  - No open ports.
  - Database is one file.
  - Easy to write stand-alone SQL app.
  - Can force entire database to reside in memory (no db file!)
  - Cannot run in client/server mode (?)
  - File system must handle locking.
  - Small code base; great for embedded processors (e.g., runs in VxWorks).
  - “Manifest typing” -- can insert any datatype in any field but will store in native form where it can.

When to use SQLite?

- Data storage, information extraction too cumbersome for a flat file or spreadsheet.
- Want full power of SQL.
- SQL performance is important.
- Type checking not important.
- Only need to support one writing user at a time.
- Don't need to run the database app on a remote computer (unless it can see the database file via NFS for example).

Installation

Use v3.2.1 (latest as of May 21, 2005) or newer when available.

Optional dependencies: readline, tcl

```
cd /tmp
wget http://www.sqlite.org/sqlite-3.2.1.tar.gz
tar zxfv sqlite-3.2.1.tar.gz    # 1.3 MB file
mkdir build
cd    build
../sqlite-3.2.1/configure --prefix=/usr/local/sqlite-3.2.1 \
                          --with-tcl=/usr/local/tcl8.4.9/lib
make

make test  # Requires tcl/tk.  19,747 tests; takes a few minutes.

# parts of v3.2.1 "make install" rely on tclsh.  Not necessary.
make -n install | grep -v tclsh > my_make_install
sh my_make_install

ln -s /usr/local/sqlite-3.2.1/bin/sqlite3 /usr/local/bin/sqlite
```

Quick Installation
(but can't do `make test`)

No tcl, no shared libraries, small and fast executable.

```
cd /tmp
wget http://www.sqlite.org/sqlite-3.2.1.tar.gz
tar zxfv sqlite-3.2.1.tar.gz    # 1.3 MB file
mkdir build
cd    build
CFLAGS=-O3 ../sqlite-3.2.1/configure \
    --prefix=/usr/local/sqlite-3.2.1 --without-tcl --disable-shared
make
make install  # simple installation without tcl
ln -s /usr/local/sqlite-3.2.1/bin/sqlite3 /usr/local/bin/sqlite
```

If you have GCC v4.1 (beta) on a Pentium4, try

```
CFLAGS="-O3 -march=pentium4 -mcpu=pentium4 -fomit-frame-pointer -pipe"
```

(courtesy Ron Young)

SQLite Performance Tuning wiki by James Done, Kevin Croft:

http://anchor.homelinux.org/SQLiteTuning

Perl Programmer? Install DBD::SQLite

```
cd /tmp
wget http://search.cpan.org/CPAN/authors/id/M/MS/MSERGEANT/DBD-SQLite-1.08.tar.gz
tar zxfv DBD-SQLite-1.08.tar.gz
cd DBD-SQLite-1.08
perl Makefile.PL    # requires DBI
make
make test
make install
```

Or, if you have CPAN configured,

```
perl -MCPAN -eshell
cpan> install DBD::SQLite
```

SQLite 'Manifest' Typing

Conventional database table create command:

```
create table T(a integer primary key,
               b varchar(200) );
```

In SQLite, the field type declarations (green text) are optional. The following statement will work:

```
create table T(a, b);
```

SQLite will try to store numeric data in binary form if it can. If it cannot it will store the data as a string.

Create 1st SQLite Database

Enter the following at a Linux/Unix command line:

```
> sqlite -version          # should show 3.2.1


> echo "create table T(a,b);"                      | sqlite simple.db
> echo "insert into T values ('October',   9);"   | sqlite simple.db
> echo "insert into T values ('November', 13);"   | sqlite simple.db

> sqlite simple.db "insert into t values ('December', 11);"

> sqlite simple.db '.tables'
> sqlite simple.db '.schema'
> sqlite simple.db '.dump'

> sqlite simple.db 'select * from T where b > 10;'

> sqlite simple.db “insert into T values (3.14159265, 'pi');”  # ??
```

`~/.sqliterc`

Get a nicer view of the output with these settings in your `~/.sqliterc`:

```
.header on
.mode column
```

Add a lot of data to 1st Database

Generate a bunch of insert statements from the command line:

```
> perl -e 'for (1..3) { printf "insert into T values
(\"string_%04d\", %4d);\n", $_, $_; }'
```

Produces this output:

```
insert into T values (“string_0001”,     1);
insert into T values (“string_0002”,     2);
insert into T values (“string_0003”,     3);
```

Pipe the commands directly into sqlite:

```
> time perl -e 'for (1..1000) { printf "insert into T values
(\"string_%04d\", %4d);\n", $_, $_; }' | sqlite simple.db
real    0m10.295s
user    0m0.040s
sys     0m0.170s
```

Only 100 inserts/second ? That's lame!

Transactions for Faster Inserts

Group inserts together in blocks of thousands of operations to increase insert performance by orders of magnitude.

```
> rm simple.db
> echo “create table T(a,b);” > inserts.sql
> echo “begin transaction;” >> inserts.sql
> perl -e 'for (1..1000) { printf "insert into t values(\"string_%04d\", %4d);\n", $_, $_; }' >> inserts.sql
> echo “commit;” >> inserts.sql

> time cat inserts.sql | sqlite simple.db
real    0m0.133s
user    0m0.030s
sys     0m0.000s
```

Increase number of inserts from 1,000 to 50,000:

```
real    0m1.384s
user    0m1.170s
sys     0m0.020s
```

36,000 inserts/second – much better!

Aside: Specs for Machine Used in Demos

- Dell Precision 360
- Pentium 4, 3.0 GHz
- 1 GB RAM
- Fast IDE hard drive (don't know brand, model)
- RedHat Enterprise Linux v3.0
- 2.4.21 kernel

- In other words, nothing fancy.

Write C code for maximum insert speed

`sqlite_insert.c` is a small (~120 line) C program that creates a table with four fields (one integer, three reals). It writes the database to `/tmp/a.db`

```
Usage:   ./sqlite_insert  <N>  <X>

        Insert <N> rows into a table of an SQLite database
        using transaction sizes of <X>.
        The table has four columns of numeric data:
          field_1 integer
          field_2 float
          field_3 float
          field_4 float
        The integer field will have values 1..<N> while the
        double precision values are random on [-50.0, 50.0]
```

```
> ./sqlite_insert 100000 50000
 100000 inserts to /tmp/a.db in 0.774 s = 129200.64 inserts/s
```

Wow!

```
> ls -l /tmp/a.db
-rw-r--r--    1 al       al        3909632 Mar 12 11:24 /tmp/a.db
```

A Real Database: Baseball Statistics

- Data from http://baseball1.com/statistics/lahman52_csv.zip
- Information on teams, players, managers, batting, fielding, awards, salaries, etc. from 1871 to 2004.
- I converted `.csv` files to an SQLite dump file:

```
wget http://danial.org/sqlite/lampsig/baseball.sql.bz2
```

- 21 tables, 40 MB SQL dump file, 20 MB SQLite database
- Create the database with

```
bunzip2 -dc baseball.sql.bz2 | sqlite bb.db
```

  Has >340,000 SQL statements so will take ~ a minute to create `bb.db`

Sample Queries

```
> sqlite bb.db '.tables'
> sqlite bb.db '.schema'

# Enter sqlite's interactive mode
> sqlite bb.db

sqlite> select count(*) from batting;

sqlite> select count(*) from fielding;
```

Which team has the best record in history?

```
sqlite> select name,yearid,w,l from teams;

sqlite> select name,sum(w),sum(l) from teams
            group by name;

sqlite> select name,sum(w),sum(l),sum(w)/sum(l)
            from teams group by name;

sqlite> select name,sum(w),sum(l),sum(w)/sum(l) as WL
            from teams group by name order by WL;
```

Joins

Which NL 3rd baseman has the most stolen bases in a season?

- Player position (3rd base) in Fielding table
- Stolen base data in Batting table
- Player's name in Master table

Create the query one step at a time:

- All 3rd basemen

```
select * from Fielding F where F.pos = "3B";
```

- All 3rd basemen in National League

```
select * from Fielding F where F.pos = "3B" and F.lgID = "NL";
```

- Names of 3rd basemen in National League

```
select M.namefirst, M.namelast from Fielding F, Master M
      where F.pos = "3B" and F.lgID = "NL" and
            M.playerID = F.playerID;
```

- Names of 3rd basemen and in National League & their stolen bases

```
select M.namefirst, M.namelast, B.sb from
    Fielding F, Master M, Batting B
       where F.pos = "3B" and F.lgID = "NL" and
       M.playerID = F.playerID and B.playerID = M.playerID;
```

Performance Tip: Indices

- I thought SQLite was supposed to be fast?!
- For optimal join performance, the join fields should be indexed.

```
create index i1 on master(playerid);
create index i2 on fielding(playerid);
create index i3 on batting(playerid);
create index i4 on fielding(lgid);
```

- Repeat:

  Names of 3rd basemen and in National League & their stolen bases

```
 select M.namefirst, M.namelast, B.sb from
 Fielding F, Master M, Batting B
       where F.pos = "3B" and F.lgID = "NL" and
       M.playerID = F.playerID
       and B.playerID = M.playerID;
```

Finish the task...

- name of third baseman with most stolen bases in NL in two steps: first find max stolen bases among 3rd basemen in NL:

```
select max(B.sb) from Fielding F, Master M, Batting B
    where F.pos = "3B" and F.lgID = "NL" and
          M.playerID = F.playerID and
          B.playerID = M.playerID;
```

- second, find name of those matching:

```
select M.namefirst, M.namelast, B.sb from
   Fielding F, Master M, Batting B
    where F.pos = "3B" and F.lgID = "NL" and
        M.playerID = F.playerID and
        B.playerID = M.playerID and
        B.sb = 129;
```

- Who can do it in a single query?

Subqueries

- Can be much faster than joins
- SQLite optimizer gets confused by nested subqueries:

```
select playerid from fielding where pos = "3B" and lgid = "NL";

select playerid,sb from batting where playerid in
(select playerid from fielding where pos = "3B" and lgid = "NL");

select max(sb) from batting where playerid in
(select playerid from fielding where pos = "3B" and lgid = "NL");

select playerid,sb from batting where sb = (
select max(sb) from batting where playerid in
(select playerid from fielding where pos = "3B" and lgid = "NL"));
```

Why does this take so long? The command below is fast...

```
select playerid,sb from batting where sb = 129;
```

Conclusion

- Client/server database engines are overkill for many applications.
- SQLite is simple, fast, powerful.
- SQLite lowers the barrier to entry for data storage, manipulation with SQL.
- An excellent tool for learning SQL.
- An excellent tool for heavy-duty SQL work.