SQLite and Tcl

Overview

SQLite is an SQL database engine. There are dozens of similar packages, both free and commercial. SQLite happens to be among the fastest and most compact. But there are also qualitative differences with SQLite that make it especially well suited for use in standalone Tcl and Tcl/Tk based software. If you have not used SQLite in your Tcl or Tcl/Tk applications before, you should give it another look.

Features Of SQLite

SQLite is a serverless database engine. Most other SQL database engines use a separate server process to manage data on the disk. Client programs employ interprocess communication (usually hidden behind a client-side library) to send SQL statements to the server and to receive back results. With SQLite, there is no server. Client programs talk directly to the database file using ordinary disk I/O operations.

Because there is no separate server process to install, set up, and administer, SQLite is a zero-configuration database engine. A program that uses SQLite does not require a database administrator (DBA) to create a database instance, set up access permissions, assign passwords, allocate disk space, and so forth. SQLite just needs to know the name of the single ordinary disk file that holds its database.

SQLite understands most of SQL-92. You can create tables and indices, of course. SQLite also support row triggers on tables and INSTEAD OF triggers on views. There are no arbitrary limits on the number of tables, indices, views, or triggers, nor on the number of columns in a table or index. Independent subqueries are supported, though correlated subqueries are not. You can join up to 31 tables in a single query using both inner and left-outer joins. Compound queries are supported using UNION, UNION ALL, INTERSECT, and EXCEPT. SQLite recognizes but does not enforce CHECK and referential integrity constraints. There are other features including some interesting extensions to the SQL language and a few standard SQL language features that are omitted. The details can be found on the website. The important point is that SQLite implements the 95% of SQL-92 that is used most often.

Transactions in SQLite are atomic, consistent, isolated, and durable (ACID), even in the face of crashes and power failures.

SQLite has no arbitrary limits on the size of rows, columns, or tables. You can efficiently store huge BLOBs without having to worry about size constraints. A single database file can in theory grow as big as 2^{47} bytes (about 250 terabytes) though to write to a database, SQLite needs to malloc for 1KB of temporary memory space for every 4MB of database. This puts a practical limit on the database size of around a hundred gigabytes.

Like TCL, SQLite does not enforce data types on variables. SQLite allows you to put any kind of data - strings, integers, floating point numbers, or blobs - into any column regardless of the declared type of that column. (One exception: SQLite only allows integers to be put into a column of type INTEGER PRIMARY KEY.) For compatibility with other SQL databases, SQLite does make an attempt to convert data into the declared type of a column as the data is inserted. But if the conversion is not possible the data is inserted anyway, using its original type. For example, if a column has type TEXT and you try to insert an integer into that column, the integer will be converted into its text representation before being inserted. If the column type is INTEGER and you try to insert text, SQLite will examine the text to see if it looks like an integer. If it does, then it is converted to an integer. If the text cannot be converted into an integer without loss of information, then the data is inserted as text.

Using SQLite From Tcl

SQLite is written in ANSI-C and comes bundled with bindings for Tcl. (Third-party binds for about two dozen other languages are also available on the internet. Only the TCL bindings are included with the core package.) To use SQLite from Tcl, you can either create a custom tclsh or wish that calls `Sqlite3_Init()` when it initializes or you can link SQLite as a shared library using the **load** command:

```
load ./libtclsqlite3.so Sqlite3
```

By putting the shared library in the right directory, you can also arrange for SQLite to be loaded using **package require**. However you get the Tcl bindings, the result will be a single new Tcl command named **sqlite3**.

The **sqlite3** command is used to create a connection to a new or an existing database. It works like this:

```
sqlite3 db example1.db
```

In the example above, the name of the file holding the database is "`example1.db`". Any filename can be used - file names do not have to have a particular suffix. If the file does not exist, it will be created the first time you try to modify the database. The file may be read-only. In order to write to the database, you must have write permission on both the database file itself and on the directory that contains the database file. If the name of the database file is "`:memory:`" then the database is held in memory and is never written to disk.

The first argument to the **sqlite3** command is the name of a new Tcl command that is used to control the database connection. In the examples given here the connection is called "**db**" but you are free to call the connection by another name if you want. All interaction with the database will be through the new **db** command - the **sqlite3** command is only used to create new connections. You can have as many database connections (with different names, of course) open at the same time. There is no arbitrary limit on the number of simultaneous connections though each connection uses 2 or 3 file descriptors which means that most systems can only have a few hundred active connections at once. Different connections can be used simultaneously in separate threads.

Close a connection to a database by deleting the corresponding Tcl command, or using the **close** method on the connection:

```
db close
```

Execute arbitrary SQL statements using the **eval** method, like this:

```
db eval {CREATE TABLE t1(a TEXT, b INTEGER)}
```

You can execute multiple SQL statements in a single **eval** by using a semicolon separator between statements.

```
db eval {
   INSERT INTO t1 VALUES('one',1);
   INSERT INTO t1 VALUES('two',2);
   INSERT INTO t1 VALUES(NULL,3);
}
```

If your statement is a query, the results of the query are returned as a list. Each column of each row is a separate element of the list. NULLs are represented in Tcl as empty strings.

```
puts [db eval {SELECT * FROM t1}]
```

The statement above, for example, generates the following output:

```
one 1 two 2 {} 3
```

It is common to want to extract a single value from a database. Because the **eval** method returns a list, you will need to use **lindex** on the result if you do this. For example:

```
set value [lindex [db eval {SELECT b FROM t1 WHERE a='one'}] 0]
```

This idiom is so common that there is a special method **onecolumn** designed to make it easier:

```
set value [db onecolumn {SELECT b FROM t1 WHERE a='one'}]
```

If you add a script argument to the end of the **eval** method, the script will be executed once for each row in the result set. Prior to each execution of the script, local variables are set to the values of columns in the row.

```
db eval {SELECT * FROM t1} {
   puts a=$a
   puts b=$b
}
```

The resulting output is:

```
a=one
b=1
a=two
b=2
a=
b=3
```

The names of the variables are the labels of the associate columns. You can change the local variable names using an AS clause in the SQL statement. For example:

```
db eval {SELECT a AS x, b AS y FROM t1} {
   puts x=$x
   puts y=$y
}
```

Within the script of a query, the **break** and **continue** commands cause the query to abort or a jump to the next row of the query, respectively, which is analogous to the way these commands work inside of **while** or **for** statements. This leads to a common short-hand way of storing the contents of a single row of the database in local Tcl variables:

```
do eval {SELECT * FROM t1} break
```

The last statement above sets local variable **a** to "one" and **b** to "1". As another example, notice that the following two statements accomplish exactly the same thing:

```
set value [db onecolumn {SELECT b FROM t1 WHERE a='one'}]
db eval {SELECT b AS value FROM t1 WHERE a='one'} break
```

If you supply the name of an array variable in between the SQL statement and the script, column values are put into elements of the array instead of into local variables. A special array element "*" contains a list of all the column names from the query result.

```
do eval {SELECT * FROM t1} v {parray v}
```

Results in...

```
v(*) = a b
v(a) = one
v(b) = 1
v(*) = a b
v(a) = two
v(b) = 2
v(*) = a b
v(a) =
v(b) = 3
```

SQLite understands Tcl variable names embedded in SQL statements. A variable name can occur in any context where it is legal to put a string or numeric literal. Suppose, for example, that local variables $a1 and $b1 contain information that you want to insert into table t1.

```
db eval {INSERT INTO t1 VALUES($a1,$b1)}
```

Notice that the SQL statement is inside curly braces so that the variables are not expanded by the Tcl interpreter. The SQL statement is passed into SQLite as written. SQLite itself recognizes the $a1 and $b1, looks up the values of those variables and uses those values for its insert. You could, of course, do the same thing in the Tcl layer. But to do so you would have to put the values of $a1 and $b1 in single quotes and escape any single quotes that happen to occur within the content $a1 and $b1, like this:

```
set qa1 [string map {' ''} $a1]
set qb1 [string map {' ''} $b1]
db eval "INSERT INTO t1 VALUES('$qa1','$qb1')"
```

Using Tcl to escape and quote the strings $a1 and $b1 involves making multiple copies of the string content. When SQLite looks up the values for itself no extra copies of the data are made. As a result, the first approach - letting SQLite look up variable values for itself - is an order of magnitude faster when the size of the strings grow large (more than about 10KB). Another advantage of letting SQLite look up variable values for itself is that variables can contain binary (BLOB) data with embedded '\000' characters. And, of course, the code is much easier to write if you do not have to worry about escaping single-quotes within strings. So in the final analysis, you should strive to always put the SQL statement in curly braces, just as you always put the argument to **expr** and the expressions of **if** and **while** in curly braces.

SQLite Extensions Written In Tcl

SQLite allows the TCL programmer to extend the functionality of the underlying SQL language by adding new SQL functions written in TCL and by intercepting and modifying the processing of SQL commands at strategic points.

Tcl procedures can be used to create new SQL functions. As a trivial example, consider giving SQLite an **sqrt()** function (which it lacks by default) using a Tcl script:

```
proc sql_sqrt {v} {return [expr {sqrt($x)}]
db function sqrt sql_sqrt
```

Once a new function is defined in this way, it can be used wherever any of the built-in SQL functions are allowed. For instance:

```
db eval {CREATE TABLE t2 AS SELECT sqrt(b) FROM t1}
db eval {SELECT x FROM t3 WHERE sqrt(a*a+b*b)>10}
```

The Tcl procedures that SQLite calls to implement these programmer-defined functions need not be simple math functions such as shown above - they can be arbitrarily complex procedures. And, though the example above shows a procedure with a single argument, you can add as many additional arguments as you like. SQLite is reentrant so the Tcl procedures that implement SQL functions can call SQLite recursively if desired. As a more interesting case, consider a new SQL command that executes arbitrary Tcl code that is supplied as its argument.

```
proc sql_eval {code} {uplevel #0 $code}
db function eval sql_eval
```

The SQLite syntax allows you to omit the FROM clause in a SELECT statement. This provides a convenient mechanism for invoking Tcl procedures from within triggers. For example:

```
db function check_ok_to_delete check_ok_to_delete
db eval {
  CREATE TRIGGER r1 BEFORE DELETE ON t2 FOR EACH ROW BEGIN
    SELECT check_ok_to_delete(old.rowid, old.b);
  END;
}
```

The **authorizer** method allows a program to receive a callback when key operations are being coded by the SQLite compiler and to change the outcome of the compilation. This feature can be used to validate user-entered SQL to prevent unauthorized commands. For example, the following authorizer procedure disallows any SQL statements other than queries and it causes any attempt to read the USER.PASSWD column to return a NULL instead of its actual content.

```
proc authorizer1 {op a1 a2 a3} {
  if {$op=="SQLITE_SELECT"} {
    return SQLITE_OK
  } elseif {$op=="SQLITE_READ"} {
    if {$a1=="USER" && $a2=="PASSWD"} {
      return SQLITE_IGNORE
    }
    return SQLITE_OK
  } else {
    return SQLITE_DENY
  }
}
```

The **commit_hook** method registers a callback that occurs right before SQLite commits a transaction. If this callback throws an exception or returns non-zero, the commit is turned into a rollback.

The **collate** method registers a callback that implements a string comparison function that is used to implement new sort orders in SQLite. The related **collation_needed** method registers a callback that occurs when SQLite encounters a collating sequence that it does not know about. This allows collating functions to be registered on-demand.

Other Interfaces

If a lengthy SQL operation is undertaken in an interactive program, the program can register a **progress** callback to be invoked periodically while the operation is underway. The progress callback can be used to update a progress bar, or to just call **::update** to prevent the screen from freezing.

There is **trace** method that registers a callback which is invoked with the text of each SQL statement just before that statement is executed. This is useful for keeping a log of SQL commands for debugging purposes.

The **busy** method registers a callback which is invoked whenever SQLite encounters a database lock to prevents it from proceeding. The busy callback can request that SQLite try the lock again after a delay or abandon the operation. The **timeout** method does a similar job. It tells SQLite to retry busy locks for a certain period of time before giving up.

The **complete** method scans an input string and returns true or false depending on whether or not the input string appears to be a complete SQL statement. This is used by routines that accept raw SQL statements interactively to know whether an input line should be sent to **eval** or whether to issue a continuation prompt and continue accepting input.

The **list_insert_rowid** method returns the ROWID of the most recently insert database row.

The **changes** method returns the number of rows that were modified by the most recent INSERT, UPDATE, or DELETE statement. The **total_changes** method returns the total number of such changes since the database handle was originally opened.

Appropriate Uses for SQLite

Perhaps the most obvious use for SQLite in a Tcl or Tcl/Tk program is as a stand-in or replacement for an enterprise client/server database engine. Many program that can benefit from the use of SQL do not require the full functionality of an enterprise class database engine. SQLite is a natural for use in such situations. If your program really does need the services of a full-featured enterprise database server like Oracle or PostgreSQL, you can still use SQLite as an efficient and low-hassle substitute during development or for demonstrations. You might not want to set up Oracle on your laptop in order to show your product at a trade show - so use SQLite instead. Or if you are giving out trial copies of your software, consider using SQLite so that your customers do not have to go to the trouble of setting up a database engine in order to evaluate your product.

But SQLite is much more than just a cheap substitute for a large-scale database server. The fact that SQLite is lightweight, embedded, cross-platform and requires no administration opens it up to many uses for which a traditional database would be inappropriate. For example, SQLite makes an excellent structured data storage subsystem for generic Tcl/Tk programs. Tcl is an outstanding language (the best currently available in my opinion) but if you have to criticize it, the most obvious complaint is its lack of sophisticated data storage mechanisms. Most programming languages have a huge repetoire of data containers so that they are likely to have a storage mechanism on hand that closely matches your needs. Tcl, in contrast, has only singleton variables and associative arrays. You can do a lot with those, but having more can be very helpful. SQLite is ready to help fill the gap.

Remember that SQLite does not require external storage to operate. You can create an in-memory database by specifying "`:memory:`" as the filename.

```
  sqlite3 db :memory:
```

Once you have such a database, you can create tables and indices to hold your data in a cleanly structured way without having to get clever with associatiative arrays. Your program will be easier to read and maintain both by yourself and by others. In many cases, developers new to a project can get a good sense of how a program works just by looking over the schema for the database. Furthermore, the code will be simpler. Joins and complex queries which used to require a subroutine can now be done in a single command. And because an in-memory database does not require any disk I/O, it is very quick.

When you begin to see the power of using SQLite as a generalized structured data container, it is a relatively small step to start using SQLite database files as your application file format. Instead of having your application data stored as an XML file, or worse as an *ad hoc* text file, consider using an SQLite database file instead. There are many benefits. SQLite database files are cross-platform so moving files from one machine to another is not a problem. You will save a small mountain of code by not having to write encoders and decoders for your data - you can use the data directly out of the database instead. You can store binary data (in BLOBs) without special encodings. You can easily do complex queries and updates on your data using a separate command-line tool. And, with no effort at all, File/Save becomes a failure-proof atomic and serialized operation.

There are a couple of approaches to using an SQLite database as your appliation file format. You could operate directly out of the database on disk. So instead of having menu options File/Open and File/Close you would instead using something like File/Connect and File/Disconnect. As you make changes to your data, the data is written to disk immediately. This is is not what users are accustomed to but it turns out to be a superior paradigm in many situations. If you prefer the more traditional File/Save metaphor you can just start a transaction when you first connect to the database and then either COMMIT when the user selects File/Save or ROLLBACK to abandon the changes. If the program crashes, changes will be rolled back automatically the next time someone else opens the file.

The second approach is the more familiar idea of reading all of the application data off of disk and into memory, modifying the data in memory, then writing everything back to disk when the user selects File/Save. This can be accomplished easily using the ATTACH extension to the SQL language that SQLite implements. Suppose your program already has an in-memory database open as shown above and further suppose that the user navigates a File/Open menu (or the equivalent) to select a file named "`appdata1.xyz`". You can very easily transfer all of the data off of disk and into memory as follows:

```
# Duplicate the schema of appdata1.xyz into the in-memory db database
sqlite3 other appdata1.xyz
other eval {SELECT sql FROM sqlite_master WHERE sql NOT NULL} {
   db eval $sql
}
other close

# Copy data content from appdata1.xyz into memory
db eval {ATTACH ’appdata1.xyz’ AS app}
db eval {SELECT name FROM sqlite_master WHERE type=’table’} {
  db eval "INSERT INTO $name SELECT * FROM app.$name"
}
db eval {DETACH app}
```

The first loop above makes an exact copy of the database schema for appdata1.xyz into the in-memory database (assuming the in-memory database is initially empty). The second loop copies all of the data out of the appdata1.xyz database into the in-memory database. If you want to ensure that the data transfer is atomic, all you have to do is enclose the code in a transaction. Writing data back out to the appdata1.xyz file works the same way only in reverse and with the additional detail that you must first remove the old data from appdata1.xyz before inserting the new database. Assuming you use a transaction, that data deletion step is perfectly safe because if you program crashes or there is a power failure before the write completes, the old database content will be restored automatically.

If an SQLite database is used as the application file format, it is simple matter to use database triggers to implement a completely general Undo/Redo mechanism. A short script can automatically generate three separate triggers on each table that record in a special history table all information needed to undo each INSERT, UPDATE, and DELETE. Any changes made to your appliation data are automatically recorded by these triggers - the application code does not need to get involved. An undo or a redo is a simple matter of playing back appropriate entries in the history table. The whole undertaking can be accomplished in around 200 lines of Tcl code. The important point is that the same undo/redo code, once written, works for *any* application that uses an SQLite database as it data store. You do not need to constantly adjust the code to accomodate changes in your data design. Because the triggers that drive the history file are generated automatically from the schema, the undo/redo module automatically adapts to changes in the schema.

Database As Program

A traditional TCL program consists of a single "main" program file that is read into the TCL interpreter first. This main file then uses the "source" command to load other files off of the disk. Sometimes it is convenient to bundle the entire program into a single file. For those occasions, the collections of source files is put into a ZIP archive or a Metakit database and read from there. But in a wrapped program, the ZIP archive or Metakit database serves as a virtual filesystem. From the point of view of the TCL interpreter, the program still consists of a main file that "sources" auxiliary files. Nothing has really changed.

Suppose one were to structure a TCL program differently. Instead of organizing the program as a series of files, why not put all of the source code in an SQLite database. Each procedure could be stored as a separate entry in the database, with the procedure body and header comments stored in separate columns. The interpreter starts the program by looking up the procedure ::main in the database and running it. The ::main procedure would typically invoke other subprocedures. The subprocedures would not be initially loaded into the interpreter. Each procedure gets loaded automatically (using the "unknown" mechanism) the first time it is invoked. The same database could also hold initial values for variables. All variables within a particular namespace could be initialized prior to invoking the first procedure from that namespace.

The database can hold more than just TCL code. It might also hold shared libraries implementing extensions. The database might hold versions of the same extension for different platforms so that the program is cross-platform. If the database also contained a list of commands that each extension implements, then extensions could be loaded automatically using the "unknown" mechanism, just as procedures are.

Since the source code is now in a structured database file instead of multiple free-format text files, it becomes much easier to build a simple development environment designed to view and update the software. Such an environment could even be built into the application under development. In other words, the application becomes its own development environment.

Mark Smith has written a small TCL module (less than 500 lines of code) that does something along these lines in the traditional file-based program model. Mark’s module brings up a new toplevel window with a tree widget on the left and a text editor widget on the right. The tree widget shows all the namespaces and under each namespace the procedures in that namespace. One can click on any procedure to see the program code for that procedure. You can make changes to the procedure code in the text editor and press "Save" to update that procedure in the running program. A separate button allows you to write the procedure text out to a file so that it can be reintegrated into the original source code manually.

The advantage of organizing the program as a database is that it would eliminate the need to manually reintegrate changes into the original file based program code. You still have a window that allows procedures to be edited on the fly, but now when you press "Save" the change is added both to the running program and to the database.

The same database that stores the program code can also store the on-line help screens. The addition of a simple help screen editor would then allow the help screens to be edited on the fly, just like program code. And because the help screens are stored in a database, it becomes much easier to add features like full-text searching to the help system.

You will probably want to disable the code-editing features prior to delivering an application to the end user. But the help screen editor, if suitably robust, could be left turned on. This would allow the end-user to make additions or comments to help screens. Put in place some kind of administrative procedure for gathering the changes inserted by end users and you suddenly have a very powerful mechanism for getting direct feedback about your product from your end users.

The database need not store just the most recent version of each procedure. It can just as well store historical versions (or the diffs needed to reconstruct them). Thus a program would consist of not just its latest iteration, but also all prior versions as well. Or, viewed another way, the application comes bundled with its own configuration management environment and source code repository. The program could even incorporate bitkeeper-style "push" and "pull" operations for synchronizing changes between multiple developers. If you wanted to get really elaborate, you could even build in support for tracking trouble tickets and bug reports.

The particular version of the program that should be run could be determined by a global variable that is initialized from the database. Thus, to go back and run an older version of the code, all you have to do is change a single value in the database and rerun the program. The correct editions of all procedures are fetched automatically.

For delivery, you may want to delete all but the most recent version of the code from the database, if for no other reason then just to keep the size of the executable to a minimum. But the versioning logic could be left turned on. That way, network connected end users could do a "pull" to upgrade there code to newer versions and could "push" back their help-screen edits. If a user did an upgrade and didn’t like the results, they could always fall back to a prior version since the older code still exists in the database.

This new program architecture might result in a significant reduction in memory utilization. When TCL executes a "proc" command, it normally stores a copy of the procedure text and compiled bytecode for the procedure in memory it obtains from malloc(). Even if the procedure is only executed once, the memory for holding the procedure body and bytecode is used for as long as the program continues to run. In the new architecture, it is a simple matter to reclaim this memory by just deleting the procedure. If the procedure ends up being needed again, it will be automatically reloaded, so there is no danger of breaking a program by deleting procedures that are seldom needed. You could even create a background timer that periodically deletes a randomly selected procedure - a garbage collector for procedures definitions.

Note also that because procedures are only loaded on demand, many procedures will never be loaded into memory during normal operation. Most users of a complex program will typically exercise only a small fraction of the code base. So the total memory consumption of a program in normal operation should be significantly less.

Concluding Thoughts

The increasing popularity of SQLite is seen in the fact that the main website daily serves about a gigabyte of data to around 3000 unique IP addresses. SQLite has been eagerly embraced by PHP, Perl, and Python programmers. What most of these enthusiastic users fail to realize is that SQLite bindings for the three P-languages are an afterthought. SQLite was designed from the beginning to be used with Tcl. Tcl bindings have been in the SQLite core since before version 1.0 and almost half of the SQLite source code base consists of regression test scripts written in Tcl. SQLite wants to be programmed in Tcl, not those other languages.

SQLite is small (the shared library is between 200K and 300K) and fast and is easy to use. And SQLite fills the important data storage gap. Once you begin using SQLite in your applications you will quickly discover new uses for it in places that you never before thought of using a database. You will soon begin to wonder how you ever got by without SQLite.