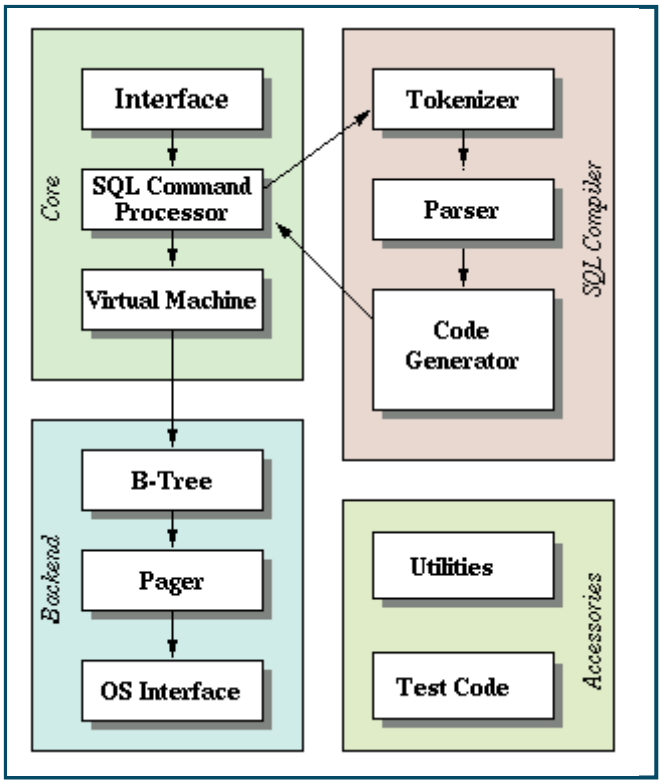
Architecture of SQLite

Introduction

This document describes the architecture of the SQLite library. The information here is useful to those who want to understand or modify the inner workings of SQLite.

A nearby diagram shows the main components of SQLite and how they interoperate. The text below explains the roles of the various components.



Overview

SQLite works by compiling SQL text into bytecode, then running that bytecode using a virtual machine.

The sqlite3_prepare_v2() and related interfaces act as a compiler for converting SQL text into bytecode. The sqlite3_stmt object is a container for a single bytecode program using to implement a single SQL statement. The sqlite3_step() interface passes a bytecode program into the virtual machine, and runs the program until it either completes, or forms a row of result to be returned, or hits a fatal error, or is interrupted.

Interface

Much of the C-language Interface is found in source files **main.c**, **legacy.c**, and **vdbeapi.c** though some routines are scattered about in other files where they can have access to data structures with file scope. The sqlite3_get_table() routine is implemented in **table.c**. The sqlite3_mprintf() routine is found in **printf.c**. The sqlite3_complete() interface is in **tokenize.c**. The TCL Interface is implemented by **tclsqlite.c**.

To avoid name collisions, all external symbols in the SQLite library begin with the prefix **sqlite3**. Those symbols that are intended for external use (in other words,

those symbols which form the API for SQLite) add an underscore, and thus begin with **sqlite3_**. Extension APIs sometimes add the extension name prior to the underscore; for example: **sqlite3rbu_** or **sqlite3session_**.

Tokenizer

When a string containing SQL statements is to be evaluated it is first sent to the tokenizer. The tokenizer breaks the SQL text into tokens and hands those tokens one by one to the parser. The tokenizer is hand-coded in the file **tokenize.c**.

Note that in this design, the tokenizer calls the parser. People who are familiar with YACC and BISON may be accustomed to doing things the other way around — having the parser call the tokenizer. Having the tokenizer call the parser is better, though, because it can be made threadsafe and it runs faster.

Parser

The parser assigns meaning to tokens based on their context. The parser for SQLite is generated using the Lemon LALR(1) parser generator. Lemon does the same job as YACC/BISON, but it uses a different input syntax which is less error-prone. Lemon also generates a parser which is reentrant and thread-safe. And Lemon defines the concept of a non-terminal destructor so that it does not leak memory when syntax errors are encountered. The grammar file that drives Lemon and that defines the SQL language that SQLite understands is found in **parse.y**.

Because Lemon is a program not normally found on development machines, the complete source code to Lemon (just one C file) is included in the SQLite distribution in the "tool" subdirectory.

Code Generator

After the parser assembles tokens into a parse tree, the code generator runs to analyze the parser tree and generate bytecode that performs the work of the SQL statement. The prepared statement object is a container for this bytecode. There are many files in the code generator, including: **attach.c**, **auth.c**, **build.c**, **delete.c**, **expr.c**, **insert.c**, **pragma.c**, **select.c**, **trigger.c**, **update.c**, **vacuum.c**, **where.c**, **wherecode.c**, and **whereexpr.c**. In these files is where most of the serious magic happens. **expr.c** handles code generation for expressions. **where*.c** handles code generation for WHERE clauses on SELECT, UPDATE and DELETE statements. The files **attach.c**, **delete.c**, **insert.c**, **select.c**, **trigger.c update.c**, and **vacuum.c** handle the code generation for SQL statements with the same names. (Each of these files calls routines in **expr.c** and **where.c** as necessary.) All other SQL statements are coded out of **build.c**. The **auth.c** file implements the functionality of sqlite3_set_authorizer().

The code generator, and especially the logic in **where*.c** and in **select.c**, is sometimes called the query planner. For any particular SQL statement, there might be hundreds, thousands, or millions of different algorithms to compute the answer. The query planner is an AI that strives to select the best algorithm from these millions of choices.

Bytecode Engine

The bytecode program created by the code generator is run by a virtual machine.

The virtual machine itself is entirely contained in a single source file **vdbe.c**. The **vdbe.h** header file defines an interface between the virtual machine and the rest of the SQLite library and **vdbeInt.h** which defines structures and interfaces that are private the virtual machine itself. Various other **vdbe*.c** files are helpers to the virtual machine. The **vdbeaux.c** file contains utilities used by the virtual machine and interface modules used by the rest of the library to construct VM programs. The **vdbeapi.c** file contains external interfaces to the virtual machine such as the sqlite3_bind_int() and sqlite3_step(). Individual values (strings, integer, floating point numbers, and BLOBs) are stored in an internal object named "Mem" which is implemented by **vdbemem.c**.

SQLite implements SQL functions using callbacks to C-language routines. Even the built-in SQL functions are implemented this way. Most of the built-in SQL functions (ex: abs(), count(), substr(), and so forth) can be found in **func.c** source file. Date and time conversion functions are found in **date.c**. Some functions such as coalesce() and typeof() are implemented as bytecode directly by the code generator.

B-Tree

An SQLite database is maintained on disk using a B-tree implementation found in the **btree.c** source file. A separate B-tree is used for each table and index in the database. All B-trees are stored in the same disk file. The file format details are stable and well-defined and are guaranteed to be compatible moving forward.

The interface to the B-tree subsystem and the rest of the SQLite library is defined by the header file **btree.h**.

Page Cache

The B-tree module requests information from the disk in fixed-size pages. The default page_size is 4096 bytes but can be any power of two between 512 and 65536 bytes. The page cache is responsible for reading, writing, and caching these pages. The page cache also provides the rollback and atomic commit abstraction and takes care of locking of the database file. The B-tree driver requests particular pages from the page cache and notifies the page cache when it wants to modify pages or commit or rollback changes. The page cache handles all the messy details of making sure the requests are handled quickly, safely, and efficiently.

The primary page cache implementation is in the **pager.c** file. WAL mode logic is in the separate **wal.c**. In-memory caching is implemented by the **pcache.c** and **pcache1.c** files. The interface between page cache subsystem and the rest of SQLite is defined by the header file **pager.h**.

OS Interface

In order to provide portability between across operating systems, SQLite uses abstract object called the VFS. Each VFS provides methods for opening, read, writing, and closing files on disk, and for other OS-specific task such as finding the current time, or obtaining randomness to initialize the built-in pseudo-random number generator. SQLite currently provides VFSes for unix (in the **os_unix.c** file) and Windows (in the **os_win.c** file).

Utilities

Memory allocation, caseless string comparison routines, portable text-to-number conversion routines, and other utilities are located in **util.c**. Symbol tables used by the parser are maintained by hash tables found in **hash.c**. The **utf.c** source file contains Unicode conversion subroutines. SQLite has its own private implementation of printf() (with some extensions) in **printf.c** and its own pseudo-random number generator (PRNG) in **random.c**.

Test Code

Files in the "src/" folder of the source tree whose names begin with **test** are for testing only and are not included in a standard build of the library.