SQLite: A Lightweight, Portable, and Stable Database Solution

By Marc Charbonneau, Morgan Weaver, Yevgeni Kamenski

Table of Contents

Abstract

This text serves as a mid-level introduction to SQLite, a serverless, open-source relational database solution that is widely implemented and highly flexible. The authors explore aspects of SQLite which differentiate it from other relational database management systems, in addition to discussing appropriate usage to better leverage SQLite's unique design, prominent architectural features, query planning and optimization, isolation and concurrency, testing and validation of the SQLite source, and finally a selection of code that illustrates a number of SQLite's aforementioned features and ease of use.

Introduction

SQLite is an open source relational database management system (RDBMS). The original software was written, in C language, 17 years ago by D. Richard Hipp. He created the original design while working for General Dynamics on a project with the United States Navy. D. Hipp coded SQLite with one main goal in mind, to allow his program to operate without installing a database management system or requiring a database administrator. So from the start SQLite was designed to be lite which is where it got its name.

Since its original release (05/29/2000) SQLite is on its third version and its popularity has grown exponentially. Today, SQLite is likely used more than all other database engines combined. Billions of copies of SQLite is embedded in software. For example, SQLite is found in:

- Every Android and iPhone devices
- Every iOS, Mac and Windows 10
- Airbus flight software for the A350 XWB family of aircraft.

Distinctive Features

Here we explore some of SQLite's distinctive features:

Serverless and self-contained:

Traditional RDBMS software (Figure 1) relies on a large server package, which are often dedicated machines that interact with the clients and database (DB) files. The database files are generally organized into many directory trees on the server filesystem (DB files in Figure 1). This may result in convoluted data organization and maintenance-heavy architecture.

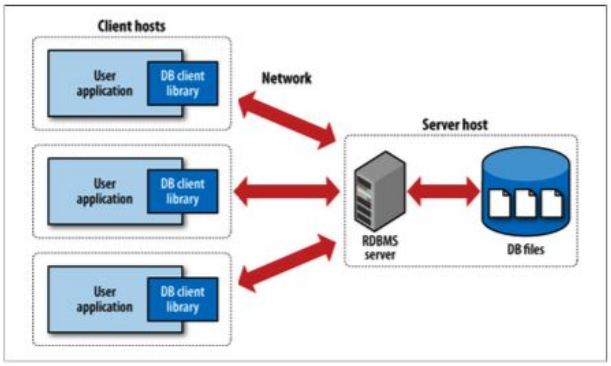

Figure 1: Traditional RDBMS[1]

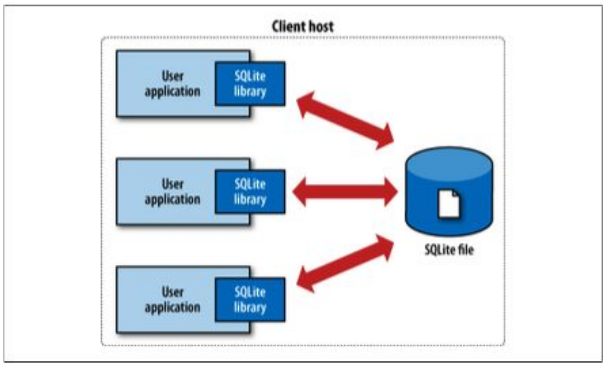

Figure 2: SQLite serverless architecture[2]

On the other hand, SQLite does not have a client/server architecture (Figure 2) or does it requires a separate server process or system to operate. The SQLite library accesses its storage files directly. On the client side (User Application in figure 2), SQLite consists of just one lightweight library that contains the entire database system.

Zero Configuration:

From an end-user perspective, SQLite requires minor effort to install and configure, as the library (database engine) is already installed on the application. There is also no server connection configuration required, which means there is no need for the traditional DB admin tasks like creating new DB instances or creating and managing user permissions.

Dynamic-type System for Tables:

Most SQL databases use static typing, which means only one data type can be defined for each column in a table and only that data type can then be stored in the records of that column. SQLite relaxes that condition by using a dynamic-type scheme for its tables. This results in the data type being defined at the record level. Therefore any value of any datatype can be stored into any column regardless of the of the initial declared type of that column.

Querying Multiple Databases:

For a traditional RDBMS, databases must be linked in order to query them simultaneously. This process sometimes involves the database administrator, this can slow down and complicate the development process by adding extra code. On the contrary, SQLite allows a single DB connection to query multiple database files without any special permissioning. This allows a developer to seamlessly integrate SQL statements that query different databases into their code.

Usage

Like any tool, much of the power of SQLite's design is accessed through appropriate usage. While SQLite has many appropriate usages, there are also a few scenarios in which SQLite is not the most ideal

[1] Using SQLite, O'Reilly, p.25
[2] Using SQLite, O'Reilly, p.25

choice, though SQLite can be used alongside other database solutions as a complementary client-side component.

One of the simplest use cases to illustrate some of SQLite's advantages is the class of hardware comprising embedded devices, especially data-capturing devices in the modern Internet of Things (IoT). The compiled SQLite library is a mere 700 Kb, but upon removing advanced features (this is in part possible due to SQLite's intentionally modular design) can be reduced to a mere 300 Kb, and finally configured to require under 256 Kb of memory. As for other hardware considerations, only a 32-bit processor is required. With such minor resource requirements, including SQLite in, say, a refrigerator's temperature-monitoring computer or a remote camera becomes not just plausible but practical--which makes its usage in more resource-equipped devices such as mobile phones seem trivial relative to a fully-equipped traditional database.

As SQLite is so compact, it lends itself to some locally-scaled uses for which larger, server-based relational database systems would be impractical. Tracking local file changes is one example of this, and since SQLite is capable of running entirely in-memory, it may be used as an application caching system. Its high reliability lends itself to this especially in the case of power failures or limited resource availability. An in-memory database such as this can be leveraged to handle payloads from a larger RDBMS, offering quick and efficient searching and processing. Web browsers such as FIrefox SQLite databases are also advantageous in that they can be accessed as read-only, so a SQLite file could even be offered on read-only media such as a read-only disc. In this way SQLite lends its model to the archiving of data. This format (and the implicit ease of use and minor setup considerations) is also great for developing small projects that do not yet warrant the resource usage of a large RDBMS server-client setup, such as prototype apps, projects that are stand-alone and do not need central database access, and academic projects.

SQLite's greatest advantages are speed, simplicity, ease of use, portability, and stability. However, it is not intended for intensive transaction loads. Therefore, it should not replace a large, heavily concurrent, high-traffic relational database. This is because SQLite only offers file-level locking, and while a file can support multiple readers, an exclusive lock is required for writes to the database. Additionally, while multiple SQLite clients may access a database file via a local network, this is still risky as the system then becomes dependent on the network locking mechanisms, which tend to be more crude and less specialized for this type of usage.

Another case in which SQLite should be avoided is very large, many-gigabyte single databases. While modern filesystems can generally process single files of this size, this may be at a performance cost. A more specialized database product may be appropriate here to ensure performance.

Replication and redundancy attempts should be avoided when using SQLite. A database may be copied in the way that the database file itself is copied, but there is no native mechanism for ensuring the consistency of multiple copies of a SQLite database. SQLite does not lend itself to high-traffic, multiple backup distributed computing in this sense.

Architecture of SQLite

SQLite has four major components (figure 3) which consist of the core, SQL compiler, backend, and accessories. This paper will cover the first three.

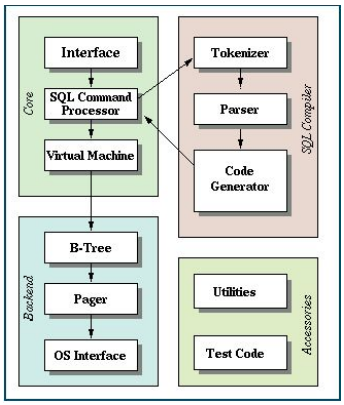

Figure 3: Architecture Overview [3]

Core:

The first component of SQLite's core is the interface. SQLite's native programming interface is in C language. But wrappers and extensions are available for all major languages, including PHP, Perl, Python, Java, C++ and .NET, among others[4]. It is important to note that these wrappers are not supported by SQLite, so they have to be downloaded and installed separately. But SQLite maintains a list of preferred third party vendors for available wrappers. The next component is the SQL command processor. It receives the user commands from the interface then passes the SQL text statement to the compiler. Once the compiler finishes, it passes back the command in bytecode format. The bytecode is then handed to a virtual machine, which makes for the third component. The virtual machine is responsible for executing the query. The execution can lead to four possible outcomes: the statement is completed (write), the statement returns rows (read), a fatal error occurs during execution, or the execution is interrupted.

SQL Compiler:

The compiler is divided into three sub-components: the tokenizer, the parser and the code generator. The tokenizer is the first step in the query evaluation process. The open source code for this functionality is available in file tokenize.c. The tokenizer is responsible for splitting the SQL text (query) into tokens. The tokenizer then hands the tokens one by one to the parser. The parser is responsible for assigning a meaning to tokens received from the tokenizer and constructing a parse tree. SQLite uses Lemon LALR(1) parser generator. Lemon was also created by D. Richard Hipp. The parser code is available in file parse.y. Finally, the code generator is responsible for analysing the parse tree, from which it generates bytecode that it returns to the core component. For more complex queries, it will call the query planner which will be covered in section *Planner and Optimizer*.

[3] http://www.sqlite.org/arch.html
[4] http://www.sqlite.org/cvstrac/wiki?p=SqliteWrappers

Backend:
Before discussing the backend component, it is important to understand that SQLite database is divided into blocks. The block-size is configurable (default 1MB). Each block is subdivided into configurable size pages (default 4096 bytes).

The backend is divided into three sub-components: b-tree, page cache and operating system (OS) interface. SQLite uses two types of b-trees, the first is a table b-tree which stores both keys and data at the leaf level. SQLite only uses table b-trees for storage. So there is one table b-tree in the database file for each row id table in the database schema. The second kind is an index b-tree which is used for indices and stores only keys (handles) at the leaf level. By default, SQLite creates a b-tree index for each of the primary keys defined in the tables. SQLite also creates a temporary b-tree index to speed up the query analysis process; this will be discussed in more detail in the *Planner and Optimizer* section.

As mentioned in the first paragraph of the Backend section, the database is divided into blocks, which are in turn divided into pages. The page cache sub-component is responsible for loading the pages into memory. It is also responsible for providing important abstractions to the system: rollback capabilities, atomic commit and locking of the database file.

Finally, the OS interface provides a common set of methods available to the other system components for communicating with the operating system. These methods include opening, reading, writing and closing files on disk.

Planner and Optimizer

A SQL statement maybe implemented in many ways depending on its complexity and the underlying database schema (availability of indices for example). The task of the query planner is to figure out the best algorithm, that minimizes disk I/O and CPU overhead, to accomplish a SQL statement. When analyzing SQL statements, a lot of the work happens at the join levels since this is where the complexity resides. All join statements are computed using nested loops. SQLite uses one loop for each table in the join. The query planner decomposes join analyses into two subtasks: picking the nested order of the various loops and choosing appropriate indices for each loop.

In the documentation, the SQLite authors makes several recommendations for avoiding problems (mainly to speedup performance) with the query planner[5]:

- Create appropriate indices. They claim that the majority of performance issues can be solved by creating indices.
- Avoid creating low-quality indices index where there are more than 10 or 20 rows in the table that have the same value for the leftmost column of the index (ex: bool or emum).
- Run analyze command (see below).

From a developer's perspective there are a few key things to know about how SQLite optimizer works. First, SQLite will create automatic indexes when none are available. These indices are temporary and last

[5] http://www.sqlite.org/optoverview.html

only for the time the query is processed. SQLite makes the following tradeoff calculation in order to decide on creating a temporary index. It compares the cost of constructing the automatic index which is O(N log N) where N is the number of entries in the table to the cost of doing a full table scan which is O(N). So an automatic index will only be created if SQLite expects that the lookup will be run more than logN times during the course of the SQL statement.

Second, the analyze command which SQLite makes available for the developer to run. The command gathers and updates statistics about the tables and the indices. The optimizer then accesses the information compiled by the analyze command in order to make a more accurate time estimation which ultimately leads to the selection of a better evaluation plan. The analyse command is not run automatically, it needs to be invoked by the programmer. The PRAGMA optimize command will automatically run analyze, which SQLite recommends invoking before closing each database connection.

Finally, the developers at SQLite are presently planning an enhancement to the optimizer. The existing version of SQLite does a full table scan when the analyze command is invoked, which is time consuming on large databases. Future work will use random sampling to obtain estimates on larger tables which will allow analyze to run faster and more frequently.

Isolation and Concurrency

SQLite's main enforcer of isolation consists of file-level locking. This can be shared read-locking, in which multiple readers may access the same SQLite database file, or exclusive-level file locking in which only a single writer may access the file. This is not ideal for high-volume write usage. However, it is highly desirable for data access that consists mainly or exclusively of reads. SQLite does make use of advanced pragma commands that modify its behavior, allowing for greater concurrency, though this should only be visited when the database user is aware of the full implications. The locking_mode pragma controls how locks are used. In EXCLUSIVE mode, the locks are not released once a transaction finishes. It can provide better performance for the transaction using it by eliminating the need to check the header for new changes upon a fresh database read. This is the default mode where there is only one application using a database, such as in-memory app caching. The locks may also be manually released.

Read_uncommitted is another concurrency manipulation option that is used with shared SQLite databases. It allows connections sharing a database to read in mid-write of other connections, before a transaction has committed. This may be perfectly fine for certain types of applications, and increases concurrency, at the risk of data corruption. This mode is only available to systems using a shared cache mode, which is not typically seen in desktop applications. Like locking_mode, read_uncommitted should be used judiciously in situations where the likelihood of corruption stemming from uncommitted reads combined with power failure or sudden shutdown is understood.

SQLite uses a transient journaling system, consisting of a journal file that appears in the same directory as the SQLite database file. The journaling system has two modes, the default of these being rollback mode. In this mode, changes are applied directly to the SQLite database, while a separate journal is constructed to roll back the database to its previous state in case of a transaction failure for any reason. As of 2010,

there is also a write-ahead log mode (or "WAL mode") which allows snapshot isolation, where the database is not directly written to, but a write-ahead version is created which is then applied to the database after transaction success. This may be enabled by the journal mode=WAL pragma. This model prevents long waits and starvation when multiple applications or readers require access to a single SQLite database and is well-suited to the constraint of file-level locking.

Another important thing to remember upon consideration of SQLite concurrency and isolation is the fact that while *separate database connections* are guaranteed isolation by default, transactions within a single connection are not guaranteed isolation and can lead to undefined behavior. This is more frequently a problem with very large, complex statements run with the BEGIN_IMMEDIATE command that are run with one or more other transactions where at least one transaction utilizes a write command like DELETE, UPDATE or INSERT. The behavior of the database will then depend on numerous factors such as SQLite version, schema, whether the ANALYZE command has been run, and the nature of the queries.

Overall, SQLite's isolation model is generally sufficient for the kinds of applications and usages for which it is intended. The developer should be aware of its models and caveats when approaching the boundaries of these usages, such as high-volume querying within a single database connection and high write to read ratio. When a project requires these latter uses, a more traditional RDBMS should be considered, with SQLite as a client-side component.

Testing and Validation

SQLite is among the most heavily tested and validated bodies of code in existence. It is no exaggeration to say this, as SQLite has been approved for high-risk scenarios such as usage in devices in space, and is a standard component of many reputable software packages and products such as iPhones, Mac OS, Firefox, and others. SQLite's testing includes 100% branch coverage, with three independent collections of test code maintained by three separate groups of developers. These extensive tests include fuzz tests, undefined behavior tests, valgrind analysis, I/O error tests, corrupt database scenario tests, and out-of-memory tests, to name a few.

The first main body of tests, TCL, are contained within the same source tree as the SQLite core, and comprise the oldest body of tests. THese are open-source, like SQLite, and include over 39,000 distinct test cases, primarily written in TCL Scripting Language, running on a C-based interface. This number is misleading, because many of the tests are parameterized, so when all scenarios are run, many are run repeatedly, covering millions of actual tests within the aforementioned set.

The second family of tests is TH3, a proprietary set of tests specialized for embedded devices and specialized platforms and scenarios. TH3 is written in C, and provides 100% branch coverage to the SQLite core library. Over 41,000 distinct cases are covered, but like TCL, these are run multiple times with various parameters, covering *hundreds* of millions of scenarios. This suite also includes a soak test to check performance under real-world use scenarios--for instance, processing 100,000 transactions over the course of 12 hours.

Finally, the SQL Logic Tests (also called SLT) set runs enormous numbers of queries against SQLite and other SQL database models--specifically PostgreSQL, MySQL, Microsoft SQL Server, and Oracle 10g--to ensure parity among query results across other products and SQLite. This is not just important for SQLite's own internal accuracy, but also because SQLite is frequently used in harmony with larger database systems such as these. The test data is about 1.2 GB in size, and about 7.2 million queries are run to ensure this parity.

Additional, smaller bodies of tests have been developed for other aspects of SQLite's functionality. For instance, the speedtest1 program estimates performance under a normal workload for SQLite, while fuzzershell tests SQLite's ability to handle out of range, mis-typed, and garbage input parameters. Threadtest3 simulates high thread volume, while mptester is a package that stress tests for a scenario in which multiple processes reading and writing within a single connection are run against SQLite.

SQLite includes a vast array of tests which exemplify the variety and reach of testing that has been slowly implemented over many years, which is indicative of a mature, highly stable, and globally trusted software product, and one that is frequently cited as the second-most used piece of software in the world. These include third-party tests such as Google's OSS Fuzzer, as of the program's inception in 2016, and American Fuzzy Lop, two fuzz testing programs developed by entities other than the SQLite foundation.

While open-source software, especially rapidly-evolving open-source software such as SQLite, does not generally have a reputation for being ruthlessly and thoroughly tested, SQLite is an exemplary instance of rigorously verified open source. This is in part due to its widespread usage, as well as its use in many mission-critical scenarios where failure is not an option.

Installation Process

SQLite can be integrated into the project as a library and linked statically or dynamically. Operating systems such as MacOS and Linux distributions already come pre-installed with the SQLite library and required headers to compile and link with zero configuration.

The recommended way to use SQLite is to directly integrate the source files into the project. The official source code, known as 'amalgamation' is a single file "sqlite3.c" (180,000 lines) and is all that is needed to use SQLite inside an application. This will allow the project to be more stable should it rely on any features that may become incompatible in the future versions and also makes it easier to maintain and distribute the source code of the application.

SQLite also comes with numerous extensions that extend the base functionality. They are disabled by default to reduce the compilation size, but can be easily enabled through C preprocessor flags. The most notable extension flags are the following:

-*DSQLITE_ENABLE_FTS5* will enable full-text search functionality
-*DSQLITE_ENABLE_JSON1* will enable JSON processing functionality.
-*DSQLITE_ENABLE_RTREE* allows efficient querying of multidimensional data sets.

To use SQLite's C API, sqlite3.h header file will be needed. Also, sqlite3ext.h header file exposes additional functionality for extending the core SQLite functionality with custom extensions. Finally, “shell.c” file is also included in the distribution that features a fully-functional SQL command-line shell.

Since SQLite is written in C, it will compile on any system supporting a C compiler.

C/C++ Programming API

The SQLite C API allows for direct access to SQLite features without an intermediary shell layer. There are two primary structs used to manage the state of the database: `sqlite3`, which is the database connection object and `sqlite3_stmt`, which manages an instance of a prepared statement. The typical workflow of a client application utilizing the C API will be the following:

1. Call *sqlite3_open()* to open a database connection and store the state in sqlite3 struct.
2. Call *sqlite3_prepare()* to compile an SQL command consisting of text SQL into an intermediary bytecode representation.
3. Call *sqlite3_bind()* to specify the user parameters of the SQL statement, for example, values to be inserted into the database or filter items for a SELECT statement.
4. Call *sqlite3_step()* to execute the prepared statement and retrieve results from the query as appropriate.
5. Call *sqlite3_column()* if needed to extract specific column values from the results of the prepared statement.
6. Call *sqlite3_finalize()* to release resources for the prepared statement.
7. Call *sqlite3_close()* to release the resources of the database connection.

Here we explore each step of the workflow:

Opening Database Connections:

The *sqlite3_open(const char* filename, sqlite3** db)* command is used to open a new database connection. The filename parameter can accept a valid path to a database file or one of the special URI forms. For example, passing *":memory:"* as a file name, will signal SQLite to store the entire database completely in RAM. Additional flags can also be used directly in the URI or passed as flags. For example, the db can be opened as read-only by appending the mode modifier to the file URI like so: *'file:data.db?mode=ro'*. If a database is opened in memory, there must be sufficient memory to store the entire db. Such a database can later be written to the disk using *sqlite3_backup_init()*.

Prepared Statements:

While SQLite can be interfaced with by using plaintext SQL commands, a preferred way is to use prepared statement API functions to improve performance and guard against SQL injection attacks. Additionally, prepared SQL statements compile to intermediate VM bytecode and are therefore more efficient to execute multiple times rather than going through the parsing, planning, and optimizing stages for each iteration of the command execution.

The command *sqlite3_prepare()* accepts the SQL command string and stores the state information in the *sqlite3_stmt* struct. When using prepared statements, the actual SQL statement is not evaluated during preparation. However, query planning and optimizing stages are processed, and referenced in the *sqlite3_stmt* struct.

Binding Values:

After the statement preparation command is processed, the next step is to bind user parameters. The parser has special tokens that serves as a placeholder for binding parameters before the prepared statement execution. For example, when preparing the SQL statement "*INSERT INTO people (id, name) VALUES (**:id, :name**);*", the two bounded parameters are **:id** and **:name**. This is just one of the possible user parameter formats used, and is the arguably the most convenient way to explicitly name the user parameters. This allows for a more flexible and maintainable code base because the client is forced to be more explicit about which variables bind to which parts of the command.

Initially, the parameters begin with a value of NULL, until binding is performed using *sqlite3_bind_X()* functions to bind any C primitive value type or pointer to a parameter placeholder. The core supported SQLite data types are *NULL*, *INTEGER*, *REAL* (8-byte floating point), *TEXT* (encoded as UTF-8 or UTF-16), and *BLOB*. *BLOB* type allows the fields to store any arbitrary length binary data, and therefore allows the marshalling and unmarshaling of any arbitrary type to and from the database.

Execute statement:

The next step is to execute the statement using the command *sqlite3_step()*. Each call to *sqlite3_step()* makes available exactly one row of the results for reading. For example, if a SELECT statement is executed that returns multiple rows, the *sqlite3_step()* can be called inside the loop to extract the data from each row and processed as needed. For each available row, the *sqlite3_column_X()* family of functions can extract the data from the columns available. These functions will extract all previously mentioned data formats.

Because of the dynamic typing nature of the SQLite column types, the requested type does not need to match the stored type. However, when that happens, the value types are implicitly converted according to the table below:

Internal Type	Requested Type	Conversion
NULL	INTEGER	Result is 0
NULL	FLOAT	Result is 0.0
NULL	TEXT	Result is a NULL pointer
NULL	BLOB	Result is a NULL pointer
INTEGER	FLOAT	Convert from integer to float
INTEGER	TEXT	ASCII rendering of the integer
INTEGER	BLOB	Same as INTEGER->TEXT
FLOAT	INTEGER	CAST to INTEGER
FLOAT	TEXT	ASCII rendering of the float
FLOAT	BLOB	CAST to BLOB
TEXT	INTEGER	CAST to INTEGER
TEXT	FLOAT	CAST to REAL
TEXT	BLOB	No change
BLOB	INTEGER	CAST to INTEGER
BLOB	FLOAT	CAST to REAL
BLOB	TEXT	Add a zero terminator if needed

The pointers returned from TEXT or BLOB data requests remain valid until the next *sqlite3_step()*,

reset(), or *finalize()* commands.

Reset:
The execution of a prepared statement finishes when *sqlite3_step()* returns result SQLITE_DONE. However it cannot be run again using the *step()*. The statement can be reset with the *sqlite3_reset()* command,which returns it to a state ready to be re-executed. This can be advantageous when the same SQL command needs to be run multiple times, since none of the preparation steps need to be evaluated again. For example, if multiple records need to be inserted into the table, the *sqlite3_reset()* command can be called inside of a loop for an efficient insert routine.

Also note that resetting the prepared statement does not change the bindings of any variables previously set. The *sqlite3_clear_bindings()* command can be used to set all of the variable bindings back to the NULL state, ready to be re-bound with different values as necessary.

Finalize:
When the prepared statement is no longer needed, all of the resources associated with it can be released and the prepared statement handle destroyed by using the *sqlite3_finalize()* command. It is important to finalize all of the prepared statements in order to avoid memory leaks.

Closing Database Connections:
The *sqlite3_close(sqlite3* db)* call closes the database handle passed and releases associated resources, by performing tasks such as closing file handles and releasing any allocated memory.

Convenience Routine

SQLite contains a number of convenience routines that make it easy to debug and obtain query results without going through the described query process. One such command is:

*sqlite3_get_table(sqlite3 *db, char *sql, char ***result, int *numRow, int *numCol, char **errMsg);*

This function prepares the statement, executes it, and parses the results into a 2D array of string pointers. The first row of pointers consists of column names, and each additional row contains data. Once created, this memory data structure can be displayed to the user with minimal effort. After processing the user data, calling *sqlite3_free_table()* will free the in-memory data table.

JSON support

SQLite supports JSON through an included set of functions that can be turned on with a compile flag. Any well-formed JSON object can be manipulated and saved, or be used as part of the SQL query. For example, the *json_extract()* function is used to extract arbitrary parameters from a JSON object and the dynamic nature of SQLite types will automatically return appropriate column types:

json_extract('{"city":"Seattle","state":"WA"}', '$.city') will return 'Seattle'

Another significant use of JSON functionality is in json_each() and json_tree() functions. These functions act like select statements, and therefore must be used in the FROM clause of SQL. The *json_each()* function walks the immediate children of the JSON DOM, and is useful to break the JSON arrays into

individual rows for processing. The *json_tree()* function walks the JSON DOM recursively, returning rows for all levels of the tree, not just the top-level children.

These and many other JSON features make SQLite an ideal fit for applications that deal with JSON.

Summary

SQLite provides a powerful database solution without many of the caveats of traditional relational database management systems. Its lightweight system requirements, miniscule footprint, and broad support for many operating systems and environments make it ideal for embedded devices, small applications, and limited-scope uses. Its simple file-based database format and serverless design make it highly portable, and have encouraged its use as a complementary component of larger relational database systems that incorporate servers. SQLite is rigorously tested with tens of millions of test cases and boasts excellent stability and failure recovery, to ensure sound performance in mission-critical scenarios. Beyond these advantages, SQLite is not just another SQL database interface--it boasts several unique features, such as dynamic typing for tables, the ability to manipulate multiple databases simultaneously via a single database connection, and the ability to run fully in-memory for application caching and other specialized uses. Its ease of setup, portability, and friendly API continue to ensure its accessibility to users at all skill levels.

We hope that this discussion of SQLite's numerous benefits and features has adequately illustrated its potential as a relational database manager, while providing guidance on its most most appropriate usage, operation and features to further leverage its full potential for the developer.

References

"Architecture of SQLite." SQLite Foundation, n.d. Web. 22 May 2017. <http://www.sqlite.org/arch.html>.

"How SQLite Is Tested." SQLite Foundation, n.d. Web. 22 May 2017. <https://www.sqlite.org/testing.html>.

"Isolation in SQLite." SQLite Foundation, n.d. Web. 22 May 2017. <https://sqlite.org/isolation.html>.

"SQLite Wrappers." SQLite Foundation, n.d. Web. 22 May 2017, <http://www.sqlite.org/cvstrac/wiki?p=SqliteWrappers>.

"The SQLite Query Planner." SQLite Foundation, n.d. Web. 22 May 2017. <http://www.sqlite.org/optoverview.html>.

Using SQLite Jay A.Kreibich - Shroff Publishers & Distr - 2014.