SQLite: A Lightweight, Portable, and Stable Database Solution

by Marc Charbonneau, Yevgeni Kamenski, Morgan Weaver

SQLite in a Nutshell

SQLite is an embedded SQL database engine.

SQLite is likely used more than all other database engines combined. Billions and billions of copies of SQLite exist in the IT world. SQLite is found in:

- Every Android device
- Every iPhone and iOS device
- Every Mac
- Every Windows10 machine

The code is in the public domain, so it is free for use for any purpose, commercial or private.

Why should you use it?

- ➔ Popular for many reasons!
- ➔ Let's look at its distinctive features

Distinctive Features

The “Lite” in SQLite does not refer to its capabilities.

Rather, SQLite is lightweight when it comes to setup complexity, administrative overhead, and resource usage.

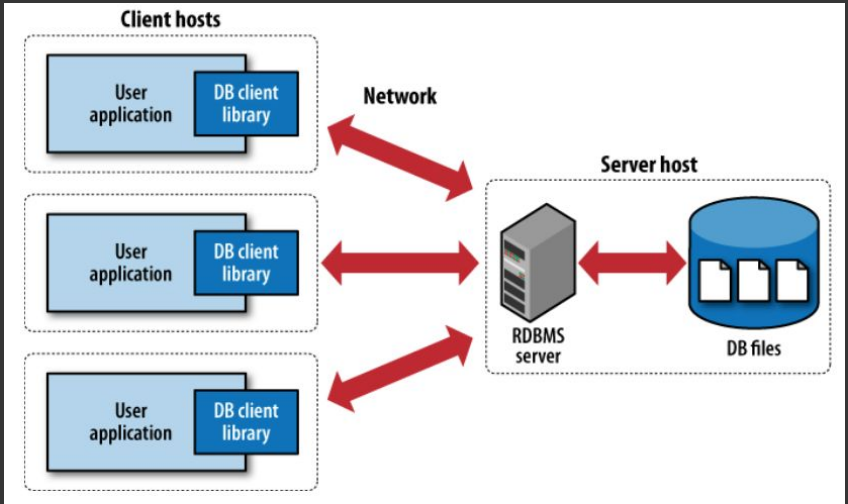
Distinctive Features

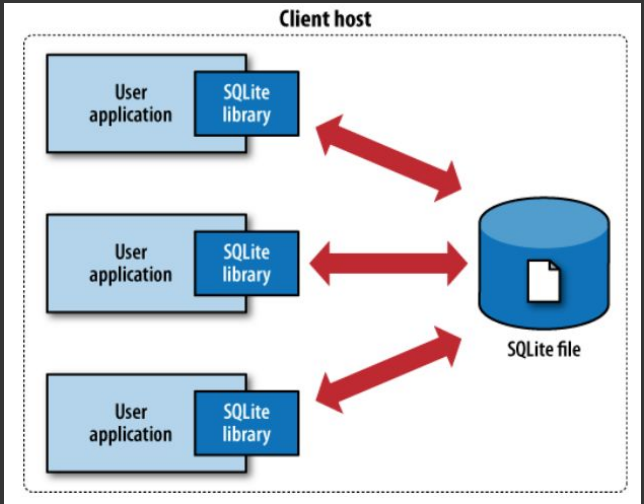
Serverless:

SQLite does not require a separate server process or system to operate. The SQLite library accesses its storage files directly.

Self-Contained:

A single library contains the entire database system, which integrates directly into a host application.





Distinctive Features

Zero Configuration:
No server means no setup. Creating an SQLite database instance is as easy as opening a file.

Cross-Platform:
The entire database instance resides in a single cross-platform file, requiring no administration.

Transactional:
SQLite transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.

Full-Featured:
SQLite supports most of the query language features found in the SQL standard.

Highly Reliable:
The SQLite development team takes code testing and verification very seriously.

When should you use it?

- ➔ Local, small-scale, and paired with traditional RDBMSs
- ➔ Not for high-traffic, write-heavy multi-user

Appropriate Usage

SQLite is an excellent choice for scenarios with limited resources.

- Its compiled library is just 700 Kb, but can be trimmed to 300 Kb
- Can be configured to require <256 Kb of memory

Ideal for embedded devices and Internet of Things

Great for small-scale applications that don't need connectivity with a larger server

Even used in application caching and tracking of local file changes

Appropriate Usage

What about scenarios where a large RDBMS is required?

SQLite can be used in a complementary role:

For example, a large SQL server pushes a set of results to a client, which stores these results in a temporary SQLite database.

This makes local management simple and fast.

Appropriate Usage

In testing and evaluation, SQLite can be a stand-in for a future implementation of a SQL server.

With no hassle and no setup, non-experts can work on the larger project and avoid wasting time with setting up a traditional SQL server.

SQLite is at its best when only a single system needs access to the database file, and usage is read-heavy OR not massively parallel due to locking behavior

- One process may have access to multiple DBs
- Multithreading is implemented at the single user level

Appropriate Usage

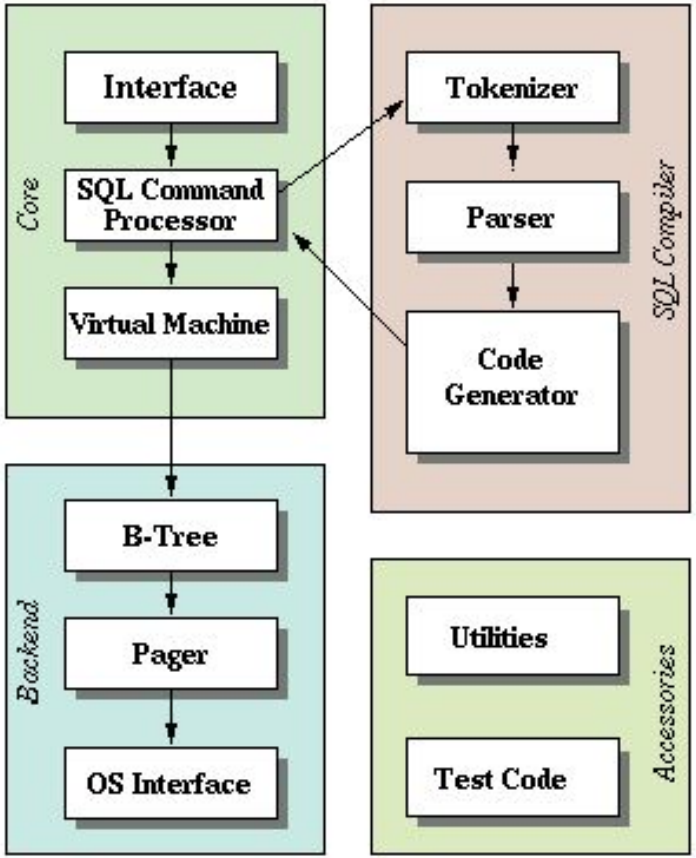
SQLite is NOT well suited to:

- Very large (many gigabytes--140 TB max) databases
- Write-heavy multi-user applications
- Client-Server models
- Distributed systems

SQLite Stack

➔ Getting in the weeds...

Architecture of SQLite



Core - Interface

- ➔ SQLite programming interface is in C
- ➔ Wrappers and extensions available for all major languages: PHP, Perl, Python, Java, C++, ODBC, .NET. See for complete list http://www.sqlite.org/cvstrac/wiki?p=SqliteWrappers
- ➔ These wrappers are not supported by SQLite (have to be downloaded and installed separately).
- ➔ Code Example: Open Command in C

```
1  int sqlite3_open(   const char *filename, sqlite3 **db_ptr )
2  int sqlite3_open16( const void *filename, sqlite3 **db_ptr )
```

- filename encoded in UTF-8 and in UTF-16

```
4  int sqlite3_open_v2( const char *filename, sqlite3 **db_ptr,
5                       int flags, const char *vfs_name )
```

- filename assumed UTF-8 (no UTF-16 variant)
- flags is a set of bit-field flags.
- vfs is generally set to NULL to use default VFS.

Core - Interface

➔ Code Example: Simple program in C

```c
#include "sqlite3.h"
#include <stdlib.h>

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

    if ( argc > 1 ){
        file = argv[1];
    }

    sqlite3_initialize( );

    rc = sqlite3_open_v2( file, &db,
        SQLITE_OPEN_READWRITE|SQLITE_OPEN_CREATE, NULL );

    if ( rc != SQLITE_OK){ // checks for return code
        sqlite3_close( db );
        exit( -1 );
    }

    /* do db stuff */

    sqlite3_close( db );
    sqlite3_shutdown( );
}
```

Core - SQL Command Processor & VM

- The SQL command processor receives a command from the interface
  - It passes the SQL text statement to the compiler and receives back bytecode
- It then hands over the bytecode to the virtual machine, that executes it
- Possible outcomes from VM:
  - Completed
  - Returns rows
  - Hits fatal error
  - Interrupted

Compiler

Tokenizer:

- First step in query evaluation.
- Splits the SQL text into tokens.
- Tokens are handed one by one to the parser.
- The tokenizer is in file **tokenize.c.**

Parser:

- Responsible for assigning a meaning to tokens received from Tokenizer and constructing a parse tree.
- Uses Lemon LALR(1) parser generator. Lemon was created by Richard Hipp who is also the creator of SQLite!
- The parser is in file **parse.y**

Code Generator:

- Analyzes the parse tree, generates bytecode and returns it to core.
- Calls the query planner, if needed, discussed in next section.

Backend

The entire database file is divided into "blocks". The block-size is configurable (default 1MB). Each block is subdivided into configurable sized pages (default 4096 bytes).

B-Tree:

- Two types of B-Trees:
  - Table b-tree: stores both key and data at leaves level
  - Index b-tree: stores keys only
- One table b-tree in the database file for each rowid table in the database schema, including system tables such as sqlite_master.
- One index b-tree in the database file for each index in the schema and one for each PK's.

Backend cont'd

Page Cache:

- 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.

OS Interface:

- Provide portability across operating systems, SQLite uses abstract object called the VFS.
- VFS provides methods for opening, read, writing, and closing files on disk, and for other OS-specific task (cache management) .

Planner and Optimizer

$$\left(1+\frac{2}{x}\right)^{x+5} = \left(\left(1+\frac{2}{x}\right)^{\frac{x}{2}}\right)^2 * \left(1+\frac{2}{x}\right)^5$$

$$e^2 * 1 = e^2 \quad \lim_{x\to a} \sqrt[p]{f(x)} = \sqrt[p]{\lim_{x\to a} f(x)}$$

$$\lim b^{f(x)} = b^{A} \quad b = \text{const}, \lim_{x\to a} f(x)$$

➔ Caution: Math

Planner

- The task of the query planner is to figure out the best algorithm to accomplish an SQL statement.

- A lot of the work happens at the join level, since this is where the complexity is:
  - SQLite computes joins using nested loops, one loop for each table in the join
  - Thus query planning decomposes into two subtasks:
    - Picking the nested order of the various loops
    - Choosing good indices for each loop

- Tips for avoiding query planner problems:
  1. Create appropriate indices
  2. Avoid creating low-quality indices
     a. 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 enum)
  3. Run analyze command (see next section)

Optimizer

Key things to know from a developer's perspective:

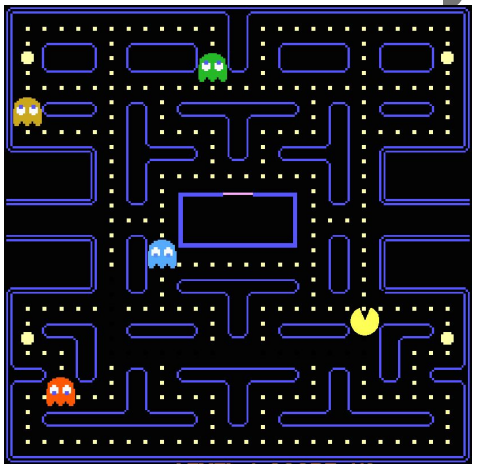
- Automatic Indices:
  - When no indices are available, SQLite can create an automatic index.
  - Lasts only for the duration of a single SQL statement.
  - Automatic index is created if:
    - Cost of constructing the automatic index is O(NlogN) (where N is the number of entries in the table)
    - Cost of doing a full table scan 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.

Optimizer cont'd

- Analyze Command:
  - Gathers statistics about tables and indices.
  - The optimizer accesses that information when available.
  - Not run automatically, programmer needs to invoke it.
  - The PRAGMA optimize command will automatically run ANALYZE.
    - Recommended to invoke before closing each database connection.

- Future Enhancements:
  - Current version of SQLite does a full table scan for ANALYZE.
  - Slow large databases.
  - Future work, use random sampling to obtain estimates on larger tables.

Isolation and Concurrency



- ➔ Great isolation for single processes/connections
- ➔ File-level locking

Isolation

SQLite provides file-level locking.

- Great for single applications, system processes
- Provides ACID
- Bad for write-intensive and high-traffic, highly concurrent loads

Separate database connections are guaranteed isolation by default

Starvation is more likely when concurrent processes or multiple connections to a database are sustained, especially at this lock granularity.

High write-to-read ratio is best for performance

Notes on Locking

Several lock specifications available

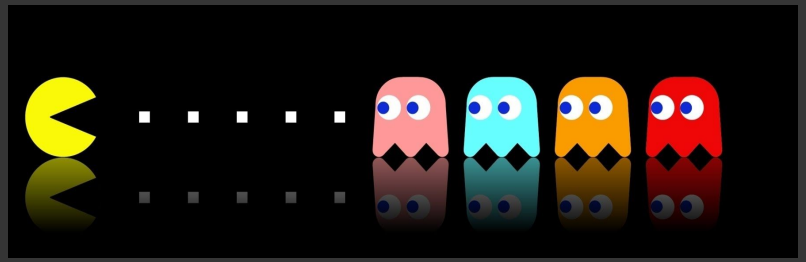
IS, IX, SIX not used because granularity is at file level

In EXCLUSIVE mode, locks are not released once a transaction finishes, which leads to better isolation

- This is engaged by default

What about starvation? Should be less likely if using SQLite appropriately: for a single DB connection

Concurrency



SQLite uses a transient journalling system by default

- AKA Rollback Mode

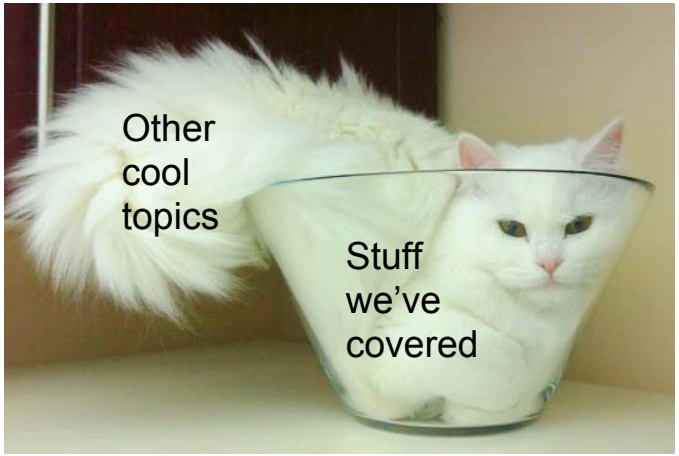
Upon failure, this modified copy is discarded and the untouched DB file remains the same.

This can be modified to write-first in Write-Ahead Log (WAL) Mode

- This enables snapshot isolation
- Must be specified with a pragma command

Read_uncommitted pragma also available; use judiciously

Other Topics



- ➔ Too many features to fit into 30 minutes
- ➔ Here we discuss testing/validation

Testing in Brief

A few details on SQLite's test coverage:

- 100% branch coverage
- 3rd party testers in addition to SQLite Foundation tests (Google)
- Fuzz tests
- Undefined behavior tests
- Valgrind analysis
- I/O error tests
- Corrupt database scenario tests
- Tens of millions of individual scenarios tested
- Out-of-memory tests . . . to name a few.

➔ See our whitepaper for many additional topics

Code Demo

➔ Yev, take it away...

Part 1: Install Process

➔ Easy peasy… Let's take a look!

Part 2: Code Template

```cpp
#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include "sqlite3.h"

extern "C" void data_entry_demo(sqlite3** db)
{
  std::cout << "\nStarting custom data entry function" << std::endl;

  // Result code. (Skipping error checking for brevity)
  int rc;

  // Open in-memory database
  rc = sqlite3_open(0, db);

  // Execute a CREATE TABLE statement to hold json data
  rc = sqlite3_exec(*db, "CREATE TABLE big(json JSON);", nullptr, nullptr, nullptr);

  // Create a new INSERT prepared statement
  sqlite3_stmt* stmt;
  rc = sqlite3_prepare_v2(*db, "INSERT INTO big VALUES ( :str )", -1, &stmt, nullptr);

  // Count the number of lines read from file
  int lines_count = 0;

  // Execute BEGIN TRANSACTION statement
  rc = sqlite3_exec(*db, "BEGIN TRANSACTION;", nullptr, nullptr, nullptr);

  // Open a JSON file from disk
  std::ifstream file("businesses.json");

  // For each line in the file
  for (std::string line; std::getline(file, line);)
  {
    // Bind the line to the INSERT prepared statement
    sqlite3_bind_text(stmt, 1, line.c_str(), -1, SQLITE_STATIC);

    // Execute the INSERT prepared statement
    rc = sqlite3_step(stmt);

    // Reset the prepared statement to be re-used again
    sqlite3_reset(stmt);

    // Clear the bound variables to be re-assigned again
    sqlite3_clear_bindings(stmt);

    ++lines_count;
  }

  // Release the resources for the prepared statement
  sqlite3_finalize(stmt);

  // Commit the transaction
  rc = sqlite3_exec(*db, "COMMIT;", nullptr, nullptr, nullptr);

  std::cout << "Inserted " << lines_count << " rows." << std::endl;
  std::cout << "Finished custom data entry\n" << std::endl;
}
```

Part 3: Running Queries

Using Yelp Restaurant Reviews Dataset

```
-- Example SQL to Extract fields directly
-- --------------------------------------

SELECT
    big.rowid as b_id
    json_extract(json, "$.name") as name,
    json_extract(json, "$.city") as city,
    json_extract(json, "$.stars") as stars,
    json_extract(json, "$.review_count") as num_reviews
FROM
    big
WHERE
    num_reviews >= 10 AND
    city = 'Las Vegas' AND
    json_extract(json, "$.is_open") = 1
ORDER BY
    stars ASC
LIMIT
    30;

-- Example SQL to use virtual tables.
-- ----------------------------------

SELECT
    big.rowid as b_id,
    atom as cat_name
FROM
    big,
    json_each(big.json, '$.categories')
WHERE
    atom is NOT NULL
;
```

Part 3: Running Queries cont'd

```
-- Complex Query Example
-- ---------------------

.timer on
.header on
.mode column
.width -10, 25, -15

SELECT * FROM (
    WITH
    business AS
    (
        SELECT
            big.rowid as bid,
            json_extract(json, "$.name") as name,
            json_extract(json, "$.city") as city,
            json_extract(json, "$.stars") as stars,
            json_extract(json, "$.review_count") as num_reviews
        FROM
            big
        WHERE
            json_extract(json, "$.is_open") = 1
    ),
    category AS
    (
        SELECT
            big.rowid as bid,
            atom as name
        FROM
            big,
            json_each(big.json, '$.categories')
        WHERE
            atom is NOT NULL
    )
    SELECT
        printf("%.2f", avg(business.stars)) AS avg_rating,
        category.name AS category_name,
        count(business.bid) AS business_cnt
    FROM
        business JOIN
        category ON
        business.bid = category.bid
    GROUP BY
        category_name
    ORDER BY
        business_cnt DESC
    LIMIT 20
)
ORDER BY
    avg_rating DESC
;
```

Why should you use it?

➔ **Zero-hassle, free, flexible, minimal resource requirements**

Thank You!

Questions?

References:

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

"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>.

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