Appropriate Uses For SQLite

SQLite is not directly comparable to client/server SQL database engines such as MySQL, Oracle, PostgreSQL, or SQL Server since SQLite is trying to solve a different problem.

Client/server SQL database engines strive to implement a shared repository of enterprise data. They emphasize scalability, concurrency, centralization, and control. SQLite strives to provide local data storage for individual applications and devices. SQLite emphasizes economy, efficiency, reliability, independence, and simplicity.

SQLite does not compete with client/server databases. SQLite competes with fopen().

Situations Where SQLite Works Well

- **Embedded devices and the internet of things**

  Because an SQLite database requires no administration, it works well in devices that must operate without expert human support. SQLite is a good fit for use in cellphones, set-top boxes, televisions, game consoles, cameras, watches, kitchen appliances, thermostats, automobiles, machine tools, airplanes, remote sensors, drones, medical devices, and robots: the "internet of things".

  Client/server database engines are designed to live inside a lovingly-attended datacenter at the core of the network. SQLite works there too, but SQLite also thrives at the edge of the network, fending for itself while providing fast and reliable data services to applications that would otherwise have dodgy connectivity.

- **Application file format**

  SQLite is often used as the on-disk file format for desktop applications such as version control systems, financial analysis tools, media cataloging and editing suites, CAD packages, record keeping programs, and so forth. The traditional File/Open operation calls sqlite3_open() to attach to the database file. Updates happen automatically as application content is revised so the File/Save menu option becomes superfluous. The File/Save_As menu option can be implemented using the backup API.

  There are many benefits to this approach, including improved performance, reduced cost and complexity, and improved reliability. See technical notes "aff_short.html" and "appfileformat.html" and "fasterthanfs.html" for more information. This use case is closely related to the data transfer format and data container use cases below.

- **Websites**

SQLite works great as the database engine for most low to medium traffic websites (which is to say, most websites). The amount of web traffic that SQLite can handle depends on how heavily the website uses its database. Generally speaking, any site that gets fewer than 100K hits/day should work fine with SQLite. The 100K hits/day figure is a conservative estimate, not a hard upper bound. SQLite has been demonstrated to work with 10 times that amount of traffic.

The SQLite website (https://www.sqlite.org/) uses SQLite itself, of course, and as of this writing (2015) it handles about 400K to 500K HTTP requests per day, about 15-20% of which are dynamic pages touching the database. Dynamic content uses about 200 SQL statements per webpage. This setup runs on a single VM that shares a physical server with 23 others and yet still keeps the load average below 0.1 most of the time.

- **Data analysis**

  People who understand SQL can employ the sqlite3 command-line shell (or various third-party SQLite access programs) to analyze large datasets. Raw data can be imported from CSV files, then that data can be sliced and diced to generate a myriad of summary reports. More complex analysis can be done using simple scripts written in Tcl or Python (both of which come with SQLite built-in) or in R or other languages using readily available adaptors. Possible uses include website log analysis, sports statistics analysis, compilation of programming metrics, and analysis of experimental results. Many bioinformatics researchers use SQLite in this way.

  The same thing can be done with an enterprise client/server database, of course. The advantage of SQLite is that it is easier to install and use and the resulting database is a single file that can be written to a USB memory stick or emailed to a colleague.

- **Cache for enterprise data**

  Many applications use SQLite as a cache of relevant content from an enterprise RDBMS. This reduces latency, since most queries now occur against the local cache and avoid a network round-trip. It also reduces the load on the network and on the central database server. And in many cases, it means that the client-side application can continue operating during network outages.

- **Server-side database**

  Systems designers report success using SQLite as a data store on server applications running in the datacenter, or in other words, using SQLite as the underlying storage engine for an application-specific database server.

  With this pattern, the overall system is still client/server: clients send requests to the server and get back replies over the network. But instead of sending generic SQL and getting back raw table content, the client requests and server responses are high-level and application-specific. The server translates requests into multiple SQL queries, gathers the results, does post-processing, filtering, and analysis, then constructs a high-level reply containing only the essential information.

  Developers report that SQLite is often faster than a client/server SQL database engine in this scenario. Database requests are serialized by the server, so concurrency is not an issue. Concurrency is also improved by "database sharding": using separate database files for different subdomains. For example, the server might have a separate SQLite database for each user, so that the server can handle hundreds or thousands of simultaneous connections, but each SQLite database is only used by one connection.

- **Data transfer format**

  Because an SQLite database is a single compact file in a well-defined cross-platform format, it is often used as a container for transferring content from one system to another. The sender gathers content into an SQLite database file, transfers that one file to the receiver, then the receiver uses SQL to extract the content as needed.

  An SQLite database facilitates data transfer between systems even when the endpoints have different word sizes and/or byte orders. The data can be a complex mix of large binary blobs, text, and small numeric or boolean values. The data format can be easily extended by adding new tables and/or columns, without breaking legacy receivers. The SQL query language means that receivers are not required to parse the entire transfer all at once, but can instead query the received content as needed. The data format is "transparent" in the sense that it is easily decoded for human viewing using a variety of universally available, open-source tools, from multiple vendors.

- **File archive and/or data container**

  The SQLite Archive idea shows how SQLite can be used as a substitute for ZIP archives or Tarballs. An archive of files stored in SQLite is only very slightly larger, and in some cases actually smaller, than the equivalent ZIP archive. And an SQLite archive features incremental and atomic updating and the ability to store much richer metadata.

  Fossil version 2.5 and later offers SQLite Archive files as a download format, in addition to traditional tarball and ZIP archive. The sqlite3.exe command-line shell version 3.22.0 and later will create, list, or unpack an SQL archiving using the .archive command.

  SQLite is a good solution for any situation that requires bundling diverse content into a self-contained and self-describing package for shipment across a network. Content is encoded in a well-defined, cross-platform, and stable file format. The encoding is efficient, and receivers can extract small subsets of the content without having to read and parse the entire file.

  SQL archives are useful as the distribution format for software or content updates that are broadcast to many clients. Variations on this idea are used, for example, to transmit TV programming guides to set-top boxes and to send over-the-air updates to vehicle navigation systems.

- **Replacement for *ad hoc* disk files**

  Many programs use fopen(), fread(), and fwrite() to create and manage files of data in home-grown formats. SQLite works particularly well as a replacement for these *ad hoc* data files. Contrary to intuition, SQLite can be faster than the filesystem for reading and writing content to disk.

- **Internal or temporary databases**

  For programs that have a lot of data that must be sifted and sorted in diverse ways, it is often easier and quicker to load the data into an in-memory SQLite database and use queries with joins and ORDER BY clauses to extract the data in the form and order needed rather than to try to code the same operations manually. Using an SQL database internally in this way also gives the program greater flexibility since new columns and indices can be added without having to recode every query.

- **Stand-in for an enterprise database during demos or testing**

  Client applications typically use a generic database interface that allows connections to various SQL database engines. It makes good sense to include SQLite in the mix of supported databases and to statically link the SQLite engine in with the client. That way the client program can be used standalone with an SQLite data file for testing or for demonstrations.

- **Education and Training**

  Because it is simple to setup and use (installation is trivial: just copy the **sqlite3** or **sqlite3.exe** executable to the target machine and run it) SQLite makes a good database engine for use in teaching SQL. Students can easily create as many databases as they like and can email databases to the instructor for comments or grading. For more advanced students who are interested in studying how an RDBMS is implemented, the modular and well-commented and documented SQLite code can serve as a good basis.

- **Experimental SQL language extensions**

  The simple, modular design of SQLite makes it a good platform for prototyping new, experimental database language features or ideas.

Situations Where A Client/Server RDBMS May Work Better

- **Client/Server Applications**

  If there are many client programs sending SQL to the same database over a network, then use a client/server database engine instead of SQLite. SQLite will work over a network filesystem, but because of the latency associated with most network filesystems, performance will not be great. Also, file locking logic is buggy in many network filesystem implementations (on both Unix and Windows). If file locking does not work correctly, two or more clients might try to modify the same part of the same database at the same time, resulting in corruption. Because this problem results from bugs in the underlying filesystem implementation, there is nothing SQLite can do to prevent it.

  A good rule of thumb is to avoid using SQLite in situations where the same database will be accessed directly (without an intervening application server) and simultaneously from many computers over a network.

- **High-volume Websites**

  SQLite will normally work fine as the database backend to a website. But if the website is write-intensive or is so busy that it requires multiple servers, then consider using an enterprise-class client/server database engine instead of SQLite.

- **Very large datasets**

  An SQLite database is limited in size to 281 terabytes (2^{48} bytes, 256 tibibytes). And even if it could handle larger databases, SQLite stores the entire database in a single disk file and many filesystems limit the maximum size of files to something less than this. So if you are contemplating databases of this magnitude, you would do well to consider using a client/server database engine that spreads its content across multiple disk files, and perhaps across multiple volumes.

- **High Concurrency**

  SQLite supports an unlimited number of simultaneous readers, but it will only allow one writer at any instant in time. For many situations, this is not a problem. Writers queue up. Each application does its database work quickly and moves on, and no lock lasts for more than a few dozen milliseconds. But there are some applications that require more concurrency, and those applications may need to seek a different solution.

Checklist For Choosing The Right Database Engine

1. **Is the data separated from the application by a network? → choose client/server**

   Relational database engines act as bandwidth-reducing data filters. So it is best to keep the database engine and the data on the same physical device so that the high-bandwidth engine-to-disk link does not have to traverse the network, only the lower-bandwidth application-to-engine link.

   But SQLite is built into the application. So if the data is on a separate device from the application, it is required that the higher bandwidth engine-to-disk link be across the network. This works, but it is suboptimal. Hence, it is usually better to select a client/server database engine when the data is on a separate device from the application.

   Nota Bene: In this rule, "application" means the code that issues SQL statements. If the "application" is an application server and if the content resides on the same physical machine as the application server, then SQLite might still be appropriate even though the end user is another network hop away.

2. **Many concurrent writers? → choose client/server**

   If many threads and/or processes need to write the database at the same instant (and they cannot queue up and take turns) then it is best to select a database engine that supports that capability, which always means a client/server database engine.

   SQLite only supports one writer at a time per database file. But in most cases, a write transaction only takes milliseconds and so multiple writers can simply take turns. SQLite will handle more write concurrency than many people suspect. Nevertheless, client/server database systems, because they have a long-running server process at hand to coordinate access, can usually handle far more write concurrency than SQLite ever will.

3. **Big data? → choose client/server**

   If your data will grow to a size that you are uncomfortable or unable to fit into a single disk file, then you should select a solution other than SQLite. SQLite supports databases up to 281 terabytes in size, assuming you can find a disk drive and filesystem that will support 281-terabyte files. Even so, when the size of the content looks like it might creep into the terabyte range, it would be good to consider a centralized client/server database.

4. **Otherwise → choose SQLite!**

   For device-local storage with low writer concurrency and less than a terabyte of content, SQLite is almost always a better solution. SQLite is fast and reliable and it requires no configuration or maintenance. It keeps things simple. SQLite "just works".