Small. Fast. Reliable.
Choose any three.

Appropriate Uses For SQLite

SQLite is different from most other SQL database engines in that its primary design goal is to be simple:

- **Simple to administer**
- **Simple to operate**
- **Simple to embed in a larger program**
- **Simple to maintain and customize**

Many people like SQLite because it is small and fast. But those qualities are just happy accidents. Users also find that SQLite is very reliable. Reliability is a consequence of simplicity. With less complication, there is less to go wrong. So, yes, SQLite is small, fast, and reliable, but first and foremost, SQLite strives to be simple.

Simplicity in a database engine can be either a strength or a weakness, depending on what you are trying to do. In order to achieve simplicity, SQLite has had to sacrifice other characteristics that some people find useful, such as high concurrency, fine-grained access control, a rich set of built-in functions, stored procedures, esoteric SQL language features, XML and/or Java extensions, tera- or peta-byte scalability, and so forth. If you need some of these features and do not mind the added complexity that they bring, then SQLite is probably not the database for you. SQLite is not intended to be an enterprise database engine. It is not designed to compete with Oracle or PostgreSQL.

The basic rule of thumb for when it is appropriate to use SQLite is this: Use SQLite in situations where simplicity of administration, implementation, and maintenance are more important than the countless complex features that enterprise database engines provide. As it turns out, situations where simplicity is the better choice are more common than many people realize.

Another way to look at SQLite is this: SQLite is not designed to replace Oracle. It is designed to replace fopen().

Situations Where SQLite Works Well

- **Application File Format**

  SQLite has been used with great success as the on-disk file format for desktop applications such as financial analysis tools, CAD packages, record keeping programs, and so forth. The traditional File/Open operation does an sqlite3_open () and executes a BEGIN TRANSACTION to get exclusive access to the content.

File/Save does a COMMIT followed by another BEGIN TRANSACTION. The use of transactions guarantees that updates to the application file are atomic, durable, isolated, and consistent.

Temporary triggers can be added to the database to record all changes into a (temporary) undo/redo log table. These changes can then be played back when the user presses the Undo and Redo buttons. Using this technique, an unlimited depth undo/redo implementation can be written in surprisingly little code.

- **Embedded devices and applications**

  Because an SQLite database requires little or no administration, SQLite is a good choice for devices or services that must work unattended and without human support. SQLite is a good fit for use in cellphones, PDAs, set-top boxes, and/or appliances. It also works well as an embedded database in downloadable consumer applications.

- **Websites**

  SQLite usually will work great as the database engine for low to medium traffic websites (which is to say, 99.9% of all websites). The amount of web traffic that SQLite can handle depends, of course, 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.

- **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.

- **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.

- **Command-line dataset analysis tool**

  Experienced SQL users can employ the command-line `sqlite3` program to analyze miscellaneous datasets. Raw data can be imported from CSV files, then that data can be sliced and diced to generate a myriad of summary reports. Possible uses include website log analysis, sports statistics analysis, compilation of programming metrics, and analysis of experimental results.

  You can also do the same thing with an enterprise client/server database, of course. The advantages to using SQLite in this situation are that SQLite is much

easier to set up and the resulting database is a single file that you can store on a floppy disk or flash-memory stick or email to a colleague.

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

  If you are writing a client application for an enterprise database engine, it makes sense to use a generic database backend that allows you to connect to many different kinds of SQL database engines. It makes even better sense to go ahead and 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.

- **Database Pedagogy**

  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. This is not to say that SQLite is an accurate model of how other database engines are implemented, but rather a student who understands how SQLite works can more quickly comprehend the operational principles of other systems.

- **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 Another RDBMS May Work Better

- **Client/Server Applications**

  If you have many client programs accessing a common database over a network, you should consider using 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, the file locking logic of many network filesystems implementation contains bugs (on both Unix and Windows). If file locking does not work like it should, it might be possible for two or more client programs to modify the same part of the same database at the same time, resulting in database 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 that you should avoid using SQLite in situations where the same database will be accessed simultaneously from many computers over a network filesystem.

- **High-volume Websites**

  SQLite will normally work fine as the database backend to a website. But if you website is so busy that you are thinking of splitting the database component off

onto a separate machine, then you should definitely consider using an enterprise-class client/server database engine instead of SQLite.

- **Very large datasets**

  With the default page size of 1024 bytes, an SQLite database is limited in size to 2 terabytes (2^{41} bytes). 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 uses reader/writer locks on the entire database file. That means if any process is reading from any part of the database, all other processes are prevented from writing any other part of the database. Similarly, if any one process is writing to the database, all other processes are prevented from reading any other part of the database. For many situations, this is not a problem. 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.