SQLite Documentation

wizardforcel

Published
with GitBook

Table of Contents

SQLite Documentation

From: SQLite Documentation

Categorical Index Of SQLite Documents

Overview Documents	
About SQLite	A high-level overview of what SQLite is and why you might be interested in using it.
Appropriate Uses For SQLite	This document describes situations where SQLite is an appropriate database engine to use versus situations where a client/server database engine might be a better choice.
Distinctive Features	This document enumerates and describes some of the features of SQLite that make it different from other SQL database engines.
How SQLite Is Tested	The reliability and robustness of SQLite is achieved in large part by thorough and careful testing. This document identifies the many tests that occur before every release of SQLite.
Copyright	SQLite is in the public domain. This document describes what that means and the implications for contributors.
Frequently Asked Questions	The title of the document says all...
Books About SQLite	A list of independently written books about SQLite.
Alphabetical Listing Of Documents	A list of all titled pages on this website, sorted by title.
Website Keyword Index	A cross-reference from keywords to various pages within this website.
Permuted Title Index	Also known as a "keyword in context" or "KWIC" index or as a concordance, this document is a listing of all other documents sorted by keyword.
SQLite Programming Interfaces	Documentation describing the APIs used to program SQLite, and the SQL dialect that it interprets.
SQLite In 5 Minutes Or Less	A very quick introduction to programming with SQLite.
Introduction to the C/C++ API	This document introduces the C/C++ API. Users should read this document before the C/C++ API Reference Guide linked below.
How To Compile SQLite	Instructions and hints for compiling SQLite C code and integrating that code with your own application.
C/C++ API Reference	This document describes each API function separately.

Result Codes	A description of the meanings of the numeric result codes returned by various C/C++ interfaces.
Tcl API	A description of the TCL interface bindings for SQLite.
SQL Syntax	This document describes the SQL language that is understood by SQLite.
Pragma commands	This document describes SQLite performance tuning options and other special purpose database commands.
Core SQL Functions	General-purpose built-in scalar SQL functions.
Aggregate SQL Functions	General-purpose built-in aggregate SQL functions.
Date and Time SQL Functions	SQL functions for manipulating dates and times.
JSON SQL Functions	SQL functions for creating, parsing, and querying JSON content.
DataTypes	SQLite version 3 introduces the concept of manifest typing, where the type of a value is associated with the value itself, not the column that it is stored in. This page describes data typing for SQLite version 3 in further detail.
SQLite Features and Extensions	Pages describing specific features or extension modules of SQLite.
8+3 Filenames	How to make SQLite work on filesystems that only support 8+3 filenames.
Autoincrement	A description of the AUTOINCREMENT keyword in SQLite, what it does, why it is sometimes useful, and why it should be avoided if not strictly necessary.
Backup API	The online-backup interface can be used to copy content from a disk file into an in-memory database or vice versa and it can make a hot backup of a live database. This application note gives examples of how.
Command-Line Shell	Notes on using the "sqlite3.exe" command-line interface that can be used to create, modify, and query arbitrary SQLite database files.
Error and Warning Log	SQLite supports an "error and warning log" design to capture information about suspicious and/or error events during operation. Embedded applications are encouraged to enable the error and warning log to help with debugging application problems that arise in the field. This document explains how to do that.
Foreign Key Support	This document describes the support for foreign key constraints introduced in version 3.6.19.
Full Text Search	A description of the SQLite Full Text Search (FTS3) extension.

Indexes On Expressions	Notes on how to create indexes on expressions instead of just individual columns.
Internal versus External Blob Storage	Should you store large BLOBs directly in the database, or store them in files and just record the filename in the database? This document seeks to shed light on that question.
Limits In SQLite	This document describes limitations of SQLite (the maximum length of a string or blob, the maximum size of a database, the maximum number of tables in a database, etc.) and how these limits can be altered at compile-time and run-time.
Memory-Mapped I/O	SQLite supports memory-mapped I/O. Learn how to enable memory-mapped I/O and about the various advantages and disadvantages to using memory-mapped I/O in this document.
Multi-threaded Programs and SQLite	SQLite is safe to use in multi-threaded programs. This document provides the details and hints on how to maximize performance.
Null Handling	Different SQL database engines handle NULLs in different ways. The SQL standards are ambiguous. This (circa 2003) document describes how SQLite handles NULLs in comparison with other SQL database engines.
Partial Indexes	A partial index is an index that only covers a subset of the rows in a table. Learn how to use partial indexes in SQLite from this document.
R-Trees	A description of the SQLite R-Tree extension. An R-Tree is a specialized data structure that supports fast multi-dimensional range queries often used in geospatial systems.
Run-Time Loadable Extensions	A general overview on how run-time loadable extensions work, how they are compiled, and how developers can create their own run-time loadable extensions for SQLite.
Shared Cache Mode	Version 3.3.0 and later supports the ability for two or more database connections to share the same page and schema cache. This feature is useful for certain specialized applications.
Unlock Notify	The "unlock notify" feature can be used in conjunction with shared cache mode to more efficiently manage resource conflict (database table locks).
URI Filenames	The names of database files can be specified using either an ordinary filename or a URI. Using URI filenames provides additional capabilities, as this document describes.
WITHOUT ROWID Tables	The WITHOUT ROWID optimization is a option that can sometimes result in smaller and faster databases.
Write-Ahead Log (WAL) Mode	Transaction control using a write-ahead log offers more concurrency and is often faster than the default rollback transactions. This document explains how to use WAL mode for improved performance.

Advocacy	Documents that strive to encourage the use of SQLite.
SQLite As An Application File Format	This article advocates using SQLite as an application file format in place of XML or JSON or a "pile-of-file".
Well Known Users	This page lists a small subset of the many thousands of devices and application programs that make use of SQLite.
SQLite Technical/Design Documentation	These documents are oriented toward describing the internal implementation details and operation of SQLite.
How Database Corruption Can Occur	SQLite is highly resistant to database corruption. But application, OS, and hardware bugs can still result in corrupt database files. This article describes many of the ways that SQLite database files can go corrupt.
Temporary Files Used By SQLite	SQLite can potentially use many different temporary files when processing certain SQL statements. This document describes the many kinds of temporary files that SQLite uses and offers suggestions for avoiding them on systems where creating a temporary file is an expensive operation.
In-Memory Databases	SQLite normally stores content in a disk file. However, it can also be used as an in-memory database engine. This document explains how.
How SQLite Implements Atomic Commit	A description of the logic within SQLite that implements transactions with atomic commit, even in the face of power failures.
Dynamic Memory Allocation in SQLite	SQLite has a sophisticated memory allocation subsystem that can be configured and customized to meet memory usage requirements of the application and that is robust against out-of-memory conditions and leak-free. This document provides the details.
Customizing And Porting SQLite	This document explains how to customize the build of SQLite and how to port SQLite to new platforms.
Locking And Concurrency In SQLite Version 3	A description of how the new locking code in version 3 increases concurrency and decreases the problem of writer starvation.
Isolation In SQLite	When we say that SQLite transactions are "serializable" what exactly does that mean? How and when are changes made visible within the same database connection and to other database connections?
Overview Of The Optimizer	A quick overview of the various query optimizations that are attempted by the SQLite code generator.
The Next-Generation Query Planner	Additional information about the SQLite query planner, and in particular the redesign of the query planner that occurred for version 3.8.0.

Architecture	An architectural overview of the SQLite library, useful for those who want to hack the code.
VDBE Opcodes	This document is an automatically generated description of the various opcodes that the VDBE understands. Programmers can use this document as a reference to better understand the output of EXPLAIN listings from SQLite.
Virtual Filesystem	The "VFS" object is the interface between the SQLite core and the underlying operating system. Learn more about how the VFS object works and how to create new VFS objects from this article.
Virtual Tables	This article describes the virtual table mechanism and API in SQLite and how it can be used to add new capabilities to the core SQLite library.
SQLite File Format	A description of the format used for SQLite database and journal files, and other details required to create software to read and write SQLite databases without using SQLite.
Compilation Options	This document describes the compile time options that may be set to modify the default behavior of the library or omit optional features in order to reduce binary size.
Upgrading SQLite, Backwards Compatibility	
Moving From SQLite 3.5 to 3.6	A document describing the differences between SQLite version 3.5.9 and 3.6.0.
Moving From SQLite 3.4 to 3.5	A document describing the differences between SQLite version 3.4.2 and 3.5.0.
Release History	A chronology of SQLite releases going back to version 1.0.0
Backwards Compatibility	This document details all of the incompatible changes to the SQLite file format that have occurred since version 1.0.0.
Private Branches	This document suggests procedures for maintaining a private branch or fork of SQLite and keeping that branch or fork in sync with the public SQLite source tree.
Obsolete Documents	The following documents are no longer current and are retained for historical reference only. These documents generally pertain to out-of-date, obsolete, and/or deprecated features and extensions.
Asynchronous IO Mode	This page describes the asynchronous IO extension developed alongside SQLite. Using asynchronous IO can cause SQLite to appear more responsive by delegating database writes to a background thread. *NB: This extension is deprecated. WAL mode is recommended as a replacement.*
Version 2 C/C++ API	A description of the C/C++ interface bindings for SQLite through version 2.8
Version 2	A description of how SQLite version 2 handles SQL datatypes.

DataTypes	Short summary: Everything is a string.
VDBE Tutorial	The VDBE is the subsystem within SQLite that does the actual work of executing SQL statements. This page describes the principles of operation for the VDBE in SQLite version 2.7. This is essential reading for anyone who want to modify the SQLite sources.
SQLite Version 3	A summary of the changes between SQLite version 2.8 and SQLite version 3.0.
Version 3 C/C++ API	A summary of the API related changes between SQLite version 2.8 and SQLite version 3.0.
Speed Comparison	The speed of version 2.7.6 of SQLite is compared against PostgreSQL and MySQL.

Overview Documents

About SQLite

See Also...

- Home
- Features
- When to use SQLite
- Frequently Asked Questions
- Well-known Users
- Books About SQLite
- Getting Started
- SQL Syntax
    - Pragmas
    - SQL functions
    - Date & time functions
    - Aggregate functions
- C/C++ Interface Spec
    - Introduction
    - List of C-language APIs
- The TCL Interface Spec
- Development Timeline
- Report a Bug
- News
- Sitemap

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is the most widely deployed database in the world with more applications than we can count, including several high-profile projects.

SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. The database file format is cross-platform - you can freely copy a database between 32-bit and 64-bit systems or between big-endian and little-endian architectures. These features make SQLite a popular choice as an Application File Format. Think of SQLite not as a replacement for Oracle but as a replacement for fopen()

SQLite is a compact library. With all features enabled, the library size can be less than 500KiB, depending on the target platform and compiler optimization settings. (64-bit code is larger. And some compiler optimizations such as aggressive function inlining and loop unrolling can cause the object code to be much larger.) If optional features are omitted, the size of the SQLite library can be reduced below 300KiB. SQLite can also be made to run in minimal stack space (4KiB) and very little heap (100KiB), making SQLite a popular database engine choice on memory constrained gadgets such as cellphones, PDAs, and MP3 players. There is a tradeoff between memory usage and speed. SQLite generally runs faster the more memory you give it. Nevertheless, performance is usually quite good even in low-memory environments.

SQLite is very carefully tested prior to every release and has a reputation for being very reliable. Most of the SQLite source code is devoted purely to testing and verification. An automated test suite runs millions and millions of test cases involving hundreds of millions of individual SQL statements and achieves 100% branch test coverage. SQLite responds gracefully to memory allocation failures and disk I/O errors. Transactions are ACID even if interrupted by system crashes or power failures. All of this is verified by the automated tests using special test harnesses which simulate system failures. Of course, even with all this testing, there are still bugs. But unlike some similar projects (especially commercial competitors) SQLite is open and honest about all bugs and provides bugs lists and minute-by-minute chronologies of bug reports and code changes.

The SQLite code base is supported by an international team of developers who work on SQLite full-time. The developers continue to expand the capabilities of SQLite and enhance its reliability and performance while maintaining backwards compatibility with the published interface spec, SQL syntax, and database file format. The source code is absolutely free to anybody who wants it, but professional support is also available.

We the developers hope that you find SQLite useful and we charge you to use it well: to make good and beautiful products that are fast, reliable, and simple to use. Seek forgiveness for yourself as you forgive others. And just as you have received SQLite for free, so also freely give, paying the debt forward.

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 emphasis 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 application performance, reduced cost and complexity, and improved reliability. See technical notes here and here for details.

- **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. Each dynamic page does roughly 200 SQL statements. 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.

- **File archives**

  The SQLite Archiver project 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.

  SQLite 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.

- **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 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 140 terabytes (2<sup><small>47</small></sup> bytes, 128 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. Writer 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 that 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 140 terabytes in size, assuming you can find a disk drive and filesystem that will support 140-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 thing simple. SQLite "just works".

Distinctive Features Of SQLite

This page highlights some of the characteristics of SQLite that are unusual and which make SQLite different from many other SQL database engines.

Zero-Configuration

> SQLite does not need to be "installed" before it is used. There is no "setup" procedure. There is no server process that needs to be started, stopped, or configured. There is no need for an administrator to create a new database instance or assign access permissions to users. SQLite uses no configuration files. Nothing needs to be done to tell the system that SQLite is running. No actions are required to recover after a system crash or power failure. There is nothing to troubleshoot.
>
> SQLite just works.
>
> Other more familiar database engines run great once you get them going. But doing the initial installation and configuration can be intimidatingly complex.

Serverless

> Most SQL database engines are implemented as a separate server process. Programs that want to access the database communicate with the server using some kind of interprocess communication (typically TCP/IP) to send requests to the server and to receive back results. SQLite does not work this way. With SQLite, the process that wants to access the database reads and writes directly from the database files on disk. There is no intermediary server process.
>
> There are advantages and disadvantages to being serverless. The main advantage is that there is no separate server process to install, setup, configure, initialize, manage, and troubleshoot. This is one reason why SQLite is a "zero-configuration" database engine. Programs that use SQLite require no administrative support for setting up the database engine before they are run. Any program that is able to access the disk is able to use an SQLite database.
>
> On the other hand, a database engine that uses a server can provide better protection from bugs in the client application - stray pointers in a client cannot corrupt memory on the server. And because a server is a single persistent process, it is able control database access with more precision, allowing for finer grain locking and better concurrency.
>
> Most SQL database engines are client/server based. Of those that are serverless, SQLite is the only one that this author knows of that allows multiple applications to access the same database at the same time.

Single Database File

> An SQLite database is a single ordinary disk file that can be located anywhere in the directory hierarchy. If SQLite can read the disk file then it can read anything in the database. If the disk file and its directory are writable, then SQLite can change anything in the database. Database files can easily be copied onto a USB memory stick or emailed for sharing.
>
> Other SQL database engines tend to store data as a large collection of files. Often these files are in a standard location that only the database engine itself can access. This makes the data more secure, but also makes it harder to access. Some SQL database engines provide the option of writing directly to disk and bypassing the filesystem all together. This provides added performance, but at the cost of considerable setup and maintenance complexity.

Stable Cross-Platform Database File

> The SQLite file format is cross-platform. A database file written on one machine can be copied to and used on a different machine with a different architecture. Big-endian or little-endian, 32-bit or 64-bit does not matter. All machines use the same file format. Furthermore, the developers have pledged to keep the file format stable and backwards compatible, so newer versions of SQLite can read and write older database files.
>
> Most other SQL database engines require you to dump and restore the database when moving from one platform to another and often when upgrading to a newer version of the software.

Compact

> When optimized for size, the whole SQLite library with everything enabled is less than 500KiB in size (as measured on an ix86 using the "size" utility from the GNU compiler suite.) Unneeded features can be disabled at compile-time to further reduce the size of the library to under 300KiB if desired.
>
> Most other SQL database engines are much larger than this. IBM boasts that its recently released CloudScape database engine is "only" a 2MiB jar file - an order of magnitude larger than SQLite even after it is compressed! Firebird boasts that its client-side library is only 350KiB. That's as big as SQLite and does not even contain the database engine. The Berkeley DB library from Oracle is 450KiB and it omits SQL support, providing the programmer with only simple key/value pairs.

Manifest typing

> Most SQL database engines use static typing. A datatype is associated with each column in a table and only values of that particular datatype are allowed to be stored in that column. SQLite relaxes this restriction by using manifest typing. In manifest typing, the datatype is a property of the value itself, not of the column in which the value is stored. SQLite thus allows the user to store any value of any datatype into any column regardless of the declared type of that column. (There are some exceptions to this rule: An INTEGER PRIMARY KEY column may only store integers. And SQLite attempts to coerce values into the declared datatype of the column when it can.)
>
> As far as we can tell, the SQL language specification allows the use of manifest typing. Nevertheless, most other SQL database engines are statically typed and so some people feel that the use of manifest typing is a bug in SQLite. But the authors of SQLite feel very strongly that this is a feature. The use of manifest typing in SQLite is a deliberate design decision which has proven in practice to make SQLite more reliable and easier to use, especially when used in combination with dynamically typed programming languages such as Tcl and Python.

Variable-length records

> Most other SQL database engines allocated a fixed amount of disk space for each row in most tables. They play special tricks for handling BLOBs and CLOBs which can be of wildly varying length. But for most tables, if you declare a column to be a VARCHAR(100) then the database engine will allocate 100 bytes of disk space regardless of how much information you actually store in that column.
>
> SQLite, in contrast, use only the amount of disk space actually needed to store the information in a row. If you store a single character in a VARCHAR(100) column, then only a single byte of disk space is consumed. (Actually two bytes - there is some overhead at the beginning of each column to record its datatype and length.)
>
> The use of variable-length records by SQLite has a number of advantages. It results in smaller database files, obviously. It also makes the database run faster, since there is less information to move to and from disk. And, the use of variable-length records makes it possible for SQLite to employ manifest typing instead of static typing.

Readable source code

> The source code to SQLite is designed to be readable and accessible to the average programmer. All procedures and data structures and many automatic variables are carefully commented with useful information about what they do. Boilerplate commenting is omitted.

SQL statements compile into virtual machine code

> Every SQL database engine compiles each SQL statement into some kind of internal data structure which is then used to carry out the work of the statement. But in most SQL engines that internal data structure is a complex web of interlinked structures and objects. In SQLite, the compiled form of statements is a short program in a machine-language like representation. Users of the database can view this virtual machine language by prepending the EXPLAIN keyword to a query.
>
> The use of a virtual machine in SQLite has been a great benefit to the library's development. The virtual machine provides a crisp, well-defined junction between the front-end of SQLite (the part that parses SQL statements and generates virtual machine code) and the back-end (the part that executes the virtual machine code and computes a result.) The virtual machine allows the developers to see clearly and in an easily readable form what SQLite is trying to do with each statement it compiles, which is a tremendous help in debugging. Depending on how it is compiled, SQLite also has the capability of tracing the execution of the virtual machine - printing each virtual machine instruction and its result as it executes.

Public domain

> The source code for SQLite is in the public domain. No claim of copyright is made on any part of the core source code. (The documentation and test code is a different matter - some sections of documentation and test logic are governed by open-source licenses.) All contributors to the SQLite core software have signed affidavits specifically disavowing any copyright interest in the code. This means that anybody is able to legally do anything they want with the SQLite source code.
>
> There are other SQL database engines with liberal licenses that allow the code to be broadly and freely used. But those other engines are still governed by copyright law. SQLite is different in that copyright law simply does not apply.
>
> The source code files for other SQL database engines typically begin with a comment describing your license rights to view and copy that file. The SQLite source code contains no license since it is not governed by copyright. Instead of a license, the SQLite source code offers a blessing:
>
> > *May you do good and not evil May you find forgiveness for yourself and forgive others May you share freely, never taking more than you give.*

SQL language extensions

> SQLite provides a number of enhancements to the SQL language not normally found in other database engines. The EXPLAIN keyword and manifest typing have already been mentioned above. SQLite also provides statements such as REPLACE and the ON CONFLICT clause that allow for added control over the resolution of constraint conflicts. SQLite supports ATTACH and DETACH commands that allow multiple independent databases to be used together in the same query. And SQLite defines APIs that allows the user to add new SQL functions and collating sequences.

How SQLite Is Tested

1.0 Introduction

The reliability and robustness of SQLite is achieved in part by thorough and careful testing.

As of version 3.12.0, the SQLite library consists of approximately 116.3 KSLOC of C code. (KSLOC means thousands of "Source Lines Of Code" or, in other words, lines of code excluding blank lines and comments.) By comparison, the project has 787 times as much test code and test scripts - 91577.3 KSLOC.

1.1 Executive Summary

- Three independently developed test harnesses
- 100% branch test coverage in an as-deployed configuration
- Millions and millions of test cases
- Out-of-memory tests
- I/O error tests
- Crash and power loss tests
- Fuzz tests
- Boundary value tests
- Disabled optimization tests
- Regression tests
- Malformed database tests
- Extensive use of assert() and run-time checks
- Valgrind analysis
- Undefined behavior checks
- Checklists

2.0 Test Harnesses

There are three independent test harnesses used for testing the core SQLite library. Each test harness is designed, maintained, and managed separately from the others.

1. The **TCL Tests** are the oldest set of tests for SQLite. They are contained in the same source tree as the SQLite core and like the SQLite core are in the public domain. The TCL tests are the primary tests used during development. The TCL tests are written using the TCL scripting language. The TCL test harness itself consists of 24.7 KSLOC of C code used to create the TCL interface. The test scripts are contained in 1038 files

totaling 12.8MB in size. There are 37706 distinct test cases, but many of the test cases are parameterized and run multiple times (with different parameters) so that on a full test run millions of separate tests are performed.

2. The **TH3** test harness is a set of proprietary tests, written in C that provide 100% branch test coverage (and 100% MC/DC test coverage) to the core SQLite library. The TH3 tests are designed to run on embedded and specialized platforms that would not easily support TCL or other workstation services. TH3 tests use only the published SQLite interfaces. TH3 is free to SQLite Consortium members and is available by license to others. TH3 consists of about 55.6 MB or 758.5 KSLOC of C code implementing 40100 distinct test cases. TH3 tests are heavily parameterized, though, so a full-coverage test runs about 1.5 million different test instances. The cases that provide 100% branch test coverage constitute a subset of the total TH3 test suite. A soak test prior to release does hundreds of millions of tests. Additional information on TH3 is available separately.
3. The **SQL Logic Test** or SLT test harness is used to run huge numbers of SQL statements against both SQLite and several other SQL database engines and verify that they all get the same answers. SLT currently compares SQLite against PostgreSQL, MySQL, Microsoft SQL Server, and Oracle 10g. SLT runs 7.2 million queries comprising 1.12GB of test data.

In addition to the three major test harnesses, there several other small programs that implement specialized tests.

1. The "speedtest1.c" program estimates the performance of SQLite under a typical workload.
2. The "mptester.c" program is a stress test for multiple processes concurrently reading and writing a single database.
3. The "threadtest3.c" program is a stress test for multiple threads using SQLite simultaneously.
4. The "fuzzershell.c" program is used to run some fuzz tests.

All of the tests above must run successfully, on multiple platforms and under multiple compile-time configurations, before each release of SQLite.

Prior to each check-in to the SQLite source tree, developers typically run a subset (called "veryquick") of the Tcl tests consisting of about 135.5 thousand test cases. The veryquick tests include most tests other than the anomaly, fuzz, and soak tests. The idea behind the veryquick tests are that they are sufficient to catch most errors, but also run in only a few minutes instead of a few hours.

3.0 Anomaly Testing

Anomaly tests are tests designed to verify the correct behavior of SQLite when something goes wrong. It is (relatively) easy to build an SQL database engine that behaves correctly on well-formed inputs on a fully functional computer. It is more difficult to build a system that responds sanely to invalid inputs and continues to function following system malfunctions. The anomaly tests are designed to verify the latter behavior.

3.1 Out-Of-Memory Testing

SQLite, like all SQL database engines, makes extensive use of malloc() (See the separate report on dynamic memory allocation in SQLite for additional detail.) On servers and workstations, malloc() never fails in practice and so correct handling of out-of-memory (OOM) errors is not particularly important. But on embedded devices, OOM errors are frighteningly common and since SQLite is frequently used on embedded devices, it is important that SQLite be able to gracefully handle OOM errors.

OOM testing is accomplished by simulating OOM errors. SQLite allows an application to substitute an alternative malloc() implementation using the sqlite3_config(SQLITE_CONFIG_MALLOC,...) interface. The TCL and TH3 test harnesses are both capable of inserting a modified version of malloc() that can be rigged to fail after a certain number of allocations. These instrumented mallocs can be set to fail only once and then start working again, or to continue failing after the first failure. OOM tests are done in a loop. On the first iteration of the loop, the instrumented malloc is rigged to fail on the first allocation. Then some SQLite operation is carried out and checks are done to make sure SQLite handled the OOM error correctly. Then the time-to-failure counter on the instrumented malloc is increased by one and the test is repeated. The loop continues until the entire operation runs to completion without ever encountering a simulated OOM failure. Tests like this are run twice, once with the instrumented malloc set to fail only once, and again with the instrumented malloc set to fail continuously after the first failure.

3.2 I/O Error Testing

I/O error testing seeks to verify that SQLite responds sanely to failed I/O operations. I/O errors might result from a full disk drive, malfunctioning disk hardware, network outages when using a network file system, system configuration or permission changes that occur in the middle of an SQL operation, or other hardware or operating system malfunctions. Whatever the cause, it is important that SQLite be able to respond correctly to these errors and I/O error testing seeks to verify that it does.

I/O error testing is similar in concept to OOM testing; I/O errors are simulated and checks are made to verify that SQLite responds correctly to the simulated errors. I/O errors are simulated in both the TCL and TH3 test harnesses by inserting a new Virtual File System object that is specially rigged to simulate an I/O error after a set number of I/O operations. As with OOM error testing, the I/O error simulators can be set to fail just once, or to fail continuously after the first failure. Tests are run in a loop, slowly increasing the point of failure until the test case runs to completion without error. The loop is run twice, once with the I/O error simulator set to simulate only a single failure and a second time with it set to fail all I/O operations after the first failure.

In I/O error tests, after the I/O error simulation failure mechanism is disabled, the database is examined using PRAGMA integrity_check to make sure that the I/O error has not introduced database corruption.

3.3 Crash Testing

Crash testing seeks to demonstrate that an SQLite database will not go corrupt if the application or operating system crashes or if there is a power failure in the middle of a database update. A separate white-paper titled Atomic Commit in SQLite describes the defensive measure SQLite takes to prevent database corruption following a crash. Crash tests strive to verify that those defensive measures are working correctly.

It is impractical to do crash testing using real power failures, of course, and so crash testing is done in simulation. An alternative Virtual File System is inserted that allows the test harness to simulate the state of the database file following a crash.

In the TCL test harness, the crash simulation is done in a separate process. The main testing process spawns a child process which runs some SQLite operation and randomly crashes somewhere in the middle of a write operation. A special VFS randomly reorders and corrupts the unsynchronized write operations to simulate the effect of buffered filesystems. After the child dies, the original test process opens and reads the test database and verifies that the changes attempted by the child either completed successfully or else were completely rolled back. The integrity_check PRAGMA is used to make sure no database corruption occurs.

The TH3 test harness needs to run on embedded systems that do not necessarily have the ability to spawn child processes, so it uses an in-memory VFS to simulate crashes. The in-memory VFS can be rigged to make a snapshot of the entire filesystem after a set number of I/O operations. Crash tests run in a loop. On each iteration of the loop, the point at which a snapshot is made is advanced until the SQLite operations being tested run to completion without ever hitting a snapshot. Within the loop, after the SQLite operation under test has completed, the filesystem is reverted to the snapshot and random file damage is introduced that is characteristic of the kinds of damage one expects to see following a power loss. Then the database is opened and checks are made to ensure that it is well-formed and that the transaction either ran to completion or was completely rolled back. The interior of the loop is repeated multiple times for each snapshot with different random damage each time.

3.4 Compound failure tests

The test suites for SQLite also explore the result of stacking multiple failures. For example, tests are run to ensure correct behavior when an I/O error or OOM fault occurs while trying to recover from a prior crash.

4.0 Fuzz Testing

Fuzz testing seeks to establish that SQLite responds correctly to invalid, out-of-range, or malformed inputs.

4.1 SQL Fuzz

SQL fuzz testing consists of creating syntactically correct yet wildly nonsensical SQL statements and feeding them to SQLite to see what it will do with them. Usually some kind of error is returned (such as "no such table"). Sometimes, purely by chance, the SQL statement also happens to be semantically correct. In that case, the resulting prepared statement is run to make sure it gives a reasonable result.

The SQL fuzz generator tests are part of the TCL test suite. During a full test run, about 105.3 thousand fuzz SQL statements are generated and tested.

4.1.1 SQL Fuzz Using The American Fuzzy Lop Fuzzer

The American Fuzzy Lop or "AFL" fuzzer is a recent (circa 2014) innovation from Michal Zalewski. Unlike most other fuzzers that blindly generate random inputs, the AFL fuzzer instruments the program being tested (by modifying the assembly-language output from the C compiler) and uses that instrumentation to detect when an input causes the program to do something different - to follow a new control path or loop a different number of times. Inputs that provoke new behavior are retained and further mutated. In this way, AFL is able to "discover" new behaviors of the program under test, including behaviors that were never envisioned by the designers.

AFL has proven remarkably adept at finding arcane bugs in SQLite. Most of the findings have been assert() statements where the conditional was false under obscure circumstances. But AFL has also found a fair number of crash bugs in SQLite, and even a few cases where SQLite computed incorrect results.

Because of its past success, AFL became a standard part of the testing strategy for SQLite beginning with version 3.8.10. Both SQL statements and database files are fuzzed. Billions and billions of mutations have been tried, but AFL's instrumentation has narrowed them down to less than 50,000 test cases that cover all distinct behaviors. Newly discovered test cases are periodically captured and added to the TCL test suite where they can be rerun using the "make fuzztest" or "make valgrindfuzz" commands.

4.2 Malformed Database Files

There are numerous test cases that verify that SQLite is able to deal with malformed database files. These tests first build a well-formed database file, then add corruption by changing one or more bytes in the file by some means other than SQLite. Then SQLite is used to read the database. In some cases, the bytes changes are in the middle of data. This causes the content of the database to change while keeping the database well-formed. In other cases, unused bytes of the file are modified, which has no effect on the integrity of the database. The interesting cases are when bytes of the file that define database structure get changed. The malformed database tests verify that SQLite finds the file format errors and reports them using the SQLITE_CORRUPT return code without overflowing buffers, dereferencing NULL pointers, or performing other unwholesome actions.

4.3 Boundary Value Tests

SQLite defines certain limits on its operation, such as the maximum number of columns in a table, the maximum length of an SQL statement, or the maximum value of an integer. The TCL and TH3 test suites both contains numerous tests that push SQLite right to the edge of its defined limits and verify that it performs correctly for all allowed values. Additional tests go beyond the defined limits and verify that SQLite correctly returns errors. The source code contains testcase macros to verify that both sides of each boundary have been tested.

5.0 Regression Testing

Whenever a bug is reported against SQLite, that bug is not considered fixed until new test cases that would exhibit the bug have been added to either the TCL or TH3 test suites. Over the years, this has resulted in thousands and thousands of new tests. These regression tests ensure that bugs that have been fixed in the past are not reintroduced into future versions of SQLite.

6.0 Automatic Resource Leak Detection

Resource leak occurs when system resources are allocated and never freed. The most troublesome resource leaks in many applications are memory leaks - when memory is allocated using malloc() but never released using free(). But other kinds of resources can also be leaked: file descriptors, threads, mutexes, etc.

Both the TCL and TH3 test harnesses automatically track system resources and report resource leaks on every test run. No special configuration or setup is required. The test harnesses are especially vigilant with regard to memory leaks. If a change causes a memory leak, the test harnesses will recognize this quickly. SQLite is designed to never leak memory, even after an exception such as an OOM error or disk I/O error. The test harnesses are zealous to enforce this.

7.0 Test Coverage

The SQLite core, including the unix VFS, has 100% branch test coverage under TH3 in its default configuration as measured by gcov. Extensions such as FTS3 and RTree are excluded from this analysis.

7.1 Statement versus branch coverage

There are many ways to measure test coverage. The most popular metric is "statement coverage". When you hear someone say that their program as "XX% test coverage" without further explanation, they usually mean statement coverage. Statement coverage measures what percentage of lines of code are executed at least once by the test suite.

Branch coverage is more rigorous than statement coverage. Branch coverage measures the number of machine-code branch instructions that are evaluated at least once on both directions.

To illustrate the difference between statement coverage and branch coverage, consider the following hypothetical line of C code:

```
if( a&gt;b && c!=25 ){ d++; }
```

Such a line of C code might generate a dozen separate machine code instructions. If any one of those instructions is ever evaluated, then we say that the statement has been tested. So, for example, it might be the case that the conditional expression is always false and the "d" variable is never incremented. Even so, statement coverage counts this line of code as having been tested.

Branch coverage is more strict. With branch coverage, each test and each subblock within the statement is considered separately. In order to achieve 100% branch coverage in the example above, there must be at least three test cases:

- a<=b <li="">a>b && c==25</li="">
- a>b && c!=25

Any one of the above test cases would provide 100% statement coverage but all three are required for 100% branch coverage. Generally speaking, 100% branch coverage implies 100% statement coverage, but the converse is not true. To reemphasize, the TH3 test harness for SQLite provides the stronger form of test coverage - 100% branch test coverage.

7.2 Coverage testing of defensive code

A well-written C program will typically contain some defensive conditionals which in practice are always true or always false. This leads to a programming dilemma: Does one remove defensive code in order to obtain 100% branch coverage?

In SQLite, the answer to the previous question is "no". For testing purposes, the SQLite source code defines macros called ALWAYS() and NEVER(). The ALWAYS() macro surrounds conditions which are expected to always evaluate as true and NEVER() surrounds conditions that are always evaluated to false. These macros serve as comments to indicate that the conditions are defensive code. In release builds, these macros are pass-throughs:

```
#define ALWAYS(X)  (X)
#define NEVER(X)   (X)
```

During most testing, however, these macros will throw an assertion fault if their argument does not have the expected truth value. This alerts the developers quickly to incorrect design assumptions.

```
#define ALWAYS(X)  ((X)?1:assert(0),0)
#define NEVER(X)   ((X)?assert(0),1:0)
```

When measuring test coverage, these macros are defined to be constant truth values so that they do not generate assembly language branch instructions, and hence do not come into play when calculating the branch coverage:

```
#define ALWAYS(X)  (1)
#define NEVER(X)   (0)
```

The test suite is designed to be run three times, once for each of the ALWAYS() and NEVER() definitions shown above. All three test runs should yield exactly the same result. There is a run-time test using the sqlite3_test_control(SQLITE_TESTCTRL_ALWAYS, ...) interface that can be used to verify that the macros are correctly set to the first form (the pass-through form) for deployment.

7.3 Forcing coverage of boundary values and boolean vector tests

Another macro used in conjunction with test coverage measurement is the `testcase()` macro. The argument is a condition for which we want test cases that evaluate to both true and false. In non-coverage builds (that is to say, in release builds) the `testcase()` macro is a no-op:

```
#define testcase(X)
```

But in a coverage measuring build, the `testcase()` macro generates code that evaluates the conditional expression in its argument. Then during analysis, a check is made to ensure tests exist that evaluate the conditional to both true and false. `Testcase()` macros are used, for example, to help verify that boundary values are tested. For example:

```
testcase( a==b );
testcase( a==b+1 );
if( a&gt;b && c!=25 ){ d++; }
```

Testcase macros are also used when two or more cases of a switch statement go to the same block of code, to make sure that the code was reached for all cases:

```
switch( op ){
  case OP_Add:
  case OP_Subtract: {
    testcase( op==OP_Add );
    testcase( op==OP_Subtract );
    /* ... */
    break;
  }
  /* ... */
}
```

For bitmask tests, `testcase()` macros are used to verify that every bit of the bitmask affects the outcome. For example, in the following block of code, the condition is true if the mask contains either of two bits indicating either a MAIN_DB or a TEMP_DB is being opened. The `testcase()` macros that precede the if statement verify that both cases are tested:

```
testcase( mask & SQLITE_OPEN_MAIN_DB );
testcase( mask & SQLITE_OPEN_TEMP_DB );
if( (mask & (SQLITE_OPEN_MAIN_DB|SQLITE_OPEN_TEMP_DB))!=0 ){ ... }
```

The SQLite source code contains 823 uses of the `testcase()` macro.

7.4 Branch coverage versus MC/DC

Two methods of measuring test coverage were described above: "statement" and "branch" coverage. There are many other test coverage metrics besides these two. Another popular metric is "Modified Condition/Decision Coverage" or MC/DC. Wikipedia defines MC/DC as follows:

- Each decision tries every possible outcome.
- Each condition in a decision takes on every possible outcome.
- Each entry and exit point is invoked.
- Each condition in a decision is shown to independently affect the outcome of the decision.

In the C programming language where `&&` and `||` are "short-circuit" operators, MC/DC and branch coverage are very nearly the same thing. The primary difference is in boolean vector tests. One can test for any of several bits in bit-vector and still obtain 100% branch test coverage even though the second element of MC/DC - the requirement that each condition in a decision take on every possible outcome - might not be satisfied.

SQLite uses `testcase()` macros as described in the previous subsection to make sure that every condition in a bit-vector decision takes on every possible outcome. In this way, SQLite also achieves 100% MC/DC in addition to 100% branch coverage.

7.5 Measuring branch coverage

Branch coverage in SQLite is currently measured using gcov with the "-b" option. First the test program is compiled using options "-g -fprofile-arcs -ftest-coverage" and then the test program is run. Then "gcov -b" is run to generate a coverage report. The coverage report is verbose and inconvenient to read, so the gcov-generated report is processed using some simple scripts to put it into a more human-friendly format. This entire process is automated using scripts, of course.

Note that running SQLite with gcov is not a test of SQLite — it is a test of the test suite. The gcov run does not test SQLite because the -fprofile-args and -ftest-coverage options cause the compiler to generate different code. The gcov run merely verifies that the test suite provides 100% branch test coverage. The gcov run is a test of the test - a meta-test.

After gcov has been run to verify 100% branch test coverage, then the test program is recompiled using delivery compiler options (without the special -fprofile-arcs and -ftest-coverage options) and the test program is rerun. This second run is the actual test of SQLite.

It is important to verify that the gcov test run and the second real test run both give the same output. Any differences in output indicate either the use of undefined or indeterminate behavior in the SQLite code (and hence a bug), or a bug in the compiler. Note that SQLite has, over the previous decade, encountered bugs in each of GCC, Clang, and MSVC. Compiler bugs, while rare, do happen, which is why it is so important to test the code in an as-delivered configuration.

7.6 Experience with full test coverage

The developers of SQLite have found that full coverage testing is an extremely effective method for locating and preventing bugs. Because every single branch instruction in SQLite core code is covered by test cases, the developers can be confident that changes made in one part of the code do not have unintended consequences in other parts of the code. The many new features and performance improvements that have been added to SQLite in recent years would not have been possible without the availability full-coverage testing.

Maintaining 100% MC/DC is laborious and time-consuming. The level of effort needed to maintain full-coverage testing is probably not cost effective for a typical application. However, we think that full-coverage testing is justified for a very widely deployed infrastructure library like SQLite, and especially for a database library which by its very nature "remembers" past mistakes.

8.0 Dynamic Analysis

Dynamic analysis refers to internal and external checks on the SQLite code which are performed while the code is live and running. Dynamic analysis has proven to be a great help in maintaining the quality of SQLite.

8.1 Assert

The SQLite core contains 4955 `assert()` statements that verify function preconditions and postconditions and loop invariants. Assert() is a macro which is a standard part of ANSI-C. The argument is a boolean value that is assumed to always be true. If the assertion is false, the program prints an error message and halts.

Assert() macros are disabled by compiling with the NDEBUG macro defined. In most systems, asserts are enabled by default. But in SQLite, the asserts are so numerous and are in such performance critical places, that the database engine runs about three times slower when asserts are enabled. Hence, the default (production) build of SQLite disables asserts. Assert statements are only enabled when SQLite is compiled with the SQLITE_DEBUG preprocessor macro defined.

8.2 Valgrind

Valgrind is perhaps the most amazing and useful developer tool in the world. Valgrind is a simulator - it simulates an x86 running a Linux binary. (Ports of Valgrind for platforms other than Linux are in development, but as of this writing, Valgrind only works reliably on Linux, which in the opinion of the SQLite developers means that Linux should be the preferred platform for all software development.) As Valgrind runs a Linux binary, it looks for all kinds of interesting errors such as array overruns, reading from uninitialized memory, stack overflows, memory leaks, and so forth. Valgrind finds problems that can easily slip through all of the other tests run against SQLite. And, when Valgrind does find an error, it can dump the developer directly into a symbolic debugger at the exact point where the error occur, to facilitate a quick fix.

Because it is a simulator, running a binary in Valgrind is slower than running it on native hardware. (To a first approximation, an application running in Valgrind on a workstation will perform about the same as it would running natively on a smartphone.) So it is impractical to run the full SQLite test suite through Valgrind. However, the veryquick tests and the coverage of the TH3 tests are run through Valgrind prior to every release.

8.3 Memsys2

SQLite contains a pluggable memory allocation subsystem. The default implementation uses system malloc() and free(). However, if SQLite is compiled with SQLITE_MEMDEBUG, an alternative memory allocation wrapper (memsys2) is inserted that looks for memory allocation errors at run-time. The memsys2 wrapper checks for memory leaks, of course, but also looks for buffer overruns, uses of uninitialized memory, and attempts to use memory after it has been freed. These same checks are also done by valgrind (and, indeed, Valgrind does them better) but memsys2 has the advantage of being much faster than Valgrind, which means the checks can be done more often and for longer tests.

8.4 Mutex Asserts

SQLite contains a pluggable mutex subsystem. Depending on compile-time options, the default mutex system contains interfaces sqlite3_mutex_held() and sqlite3_mutex_notheld() that detect whether or not a particular mutex is held by the calling thread. These two interfaces are used extensively within assert() statements in SQLite to verify mutexes are held and released at all the right moments, in order to double-check that SQLite does work correctly in multi-threaded applications.

8.5 Journal Tests

One of the things that SQLite does to ensure that transactions are atomic across system crashes and power failures is to write all changes into the rollback journal file prior to changing the database. The TCL test harness contains an alternative OS backend implementation that helps to verify this is occurring correctly. The "journal-test VFS" monitors all disk I/O traffic between the database file and rollback journal, checking to make sure that nothing is written into the database file which has not first been written and synced to the rollback journal. If any discrepancies are found, an assertion fault is raised.

The journal tests are an additional double-check over and above the crash tests to make sure that SQLite transactions will be atomic across system crashes and power failures.

8.6 Undefined Behavior Checks

In the C programming language, it is very easy to write code that has "undefined" or "implementation defined" behavior. That means that the code might work during development, but then give a different answer on a different system, or when recompiled using different compiler options. Examples of undefined and implementation-defined behavior in ANSI C include:

- Signed integer overflow. (Signed integer overflow does not necessarily wrap around, as most people expect.)
- Shifting an N-bit integer by more than N bits.
- Shifting by a negative amount.
- Shifting a negative number.
- Using the memcpy() function on overlapping buffers.
- The order of evaluation of function arguments.
- Whether or not "char" variables are signed or unsigned.
- And so forth....

Since undefined and implementation-defined behavior is non-portable and can easily lead to incorrect answers, SQLite works very hard to avoid it. For example, when adding two integer column values together as part of an SQL statement, SQLite does not simple add them together using the C-language "+" operator. Instead, it first checks to make sure the addition will not overflow, and if it will, it does the addition using floating point instead.

To help ensure that SQLite does not make use of undefined or implementation defined behavior, the test suites are rerun using instrumented builds that try to detect undefined behavior. For example, test suites are run using the "-ftrapv" option of GCC. And they are run again using the "-fsanitize=undefined" option on Clang. And again using the "/RTC1" option in MSVC. Then the test suites are rerun using options like "-funsigned-char" and "-fsigned-char" to make sure that implementation differences do not matter either. Tests are then repeated on 32-bit and 64-bit systems and on big-endian and little-endian systems, using a variety of CPU architectures. Furthermore, the test suites are augmented with many test cases that are deliberately designed to provoke undefined behavior. For example: "**SELECT -1*(-9223372036854775808);**".

9.0 Disabled Optimization Tests

The sqlite3_test_control(SQLITE_TESTCTRL_OPTIMIZATIONS, ...) interface allows selected SQL statement optimizations to be disabled at run-time. SQLite should always generate exactly the same answer with optimizations enabled and with optimizations disabled; the answer simply arrives quicker with the optimizations turned on. So in a production environment, one always leaves the optimizations turned on (the default setting).

One verification technique used on SQLite is to run an entire test suite twice, once with optimizations left on and a second time with optimizations turned off, and verify that the same output is obtained both times. This shows that the optimizations do not introduce errors.

Not all test cases can be handled this way. Some test cases check to verify that the optimizations really are reducing the amount of computation by counting the number of disk accesses, sort operations, full-scan steps, or other processing steps that occur during queries. Those test cases will appear to fail when optimizations are disabled. But the majority of test cases simply check that the correct answer was obtained, and all of those cases can be run successfully with and without the optimizations, in order to show that the optimizations do not cause malfunctions.

10.0 Checklists

The SQLite developers use an on-line checklist to coordinate testing activity and to verify that all tests pass prior each SQLite release. Past checklists are retained for historical reference. (The checklists are read-only for anonymous internet viewers, but developers can log in and update checklist items in their web browsers.) The use of checklists for SQLite testing and other development activities is inspired by *The Checklist Manifesto* .

The latest checklists contain approximately 200 items that are individually verified for each release. Some checklist items only take a few seconds to verify and mark off. Others involve test suites that run for many hours.

The release checklist is not automated: developers run each item on the checklist manually. We find that it is important to keep a human in the loop. Sometimes problems are found while running a checklist item even though the test itself passed. It is important to have a human reviewing the test output at the highest level, and constantly asking "Is this really right?"

The release checklist is continuously evolving. As new problems or potential problems are discovered, new checklist items are added to make sure those problems do not appear in subsequent releases. The release checklist has proven to be an invaluable tool in helping to ensure that nothing is overlooked during the release process.

11.0 Static Analysis

Static analysis means analyzing code at or before compile-time to check for correctness. Static analysis includes looking at compiler warning messages and running the code through more in-depth analysis engines such as the Clang Static Analyzer. SQLite compiles without warnings on GCC and Clang using the -Wall and -Wextra flags on Linux and Mac and on MSVC on Windows. No valid warnings are generated by the Clang Static Analyzer tool "scan-build" either (though recent versions of clang seem to generate many false-positives.) Nevertheless, some warnings might be generated by other static analyzers. Users are encouraged not to stress over these warnings and to instead take solace in the intense testing of SQLite described above.

Static analysis has not proven to be especially helpful in finding bugs in SQLite. Static analysis has found a few bugs in SQLite, but those are the exceptions. More bugs have been introduced into SQLite while trying to get it to compile without warnings than have been found by static analysis.

12.0 Summary

SQLite is open source. This gives many people the idea that it is not well tested as commercial software and is perhaps unreliable. But that impression is false. SQLite has exhibited very high reliability in the field and a very low defect rate, especially considering how rapidly it is evolving. The quality of SQLite is achieved in part by careful code design and implementation. But extensive testing also plays a vital role in maintaining and improving the quality of SQLite. This document has summarized the testing procedures that every release of SQLite undergoes with the hope of inspiring confidence that SQLite is suitable for use in mission-critical applications.

SQLite Is Public Domain

SQLite is in the Public Domain

All of the code and documentation in SQLite has been dedicated to the public domain by the authors. All code authors, and representatives of the companies they work for, have signed affidavits dedicating their contributions to the public domain and originals of those signed affidavits are stored in a firesafe at the main offices of Hwaci. Anyone is free to copy, modify, publish, use, compile, sell, or distribute the original SQLite code, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

The previous paragraph applies to the deliverable code and documentation in SQLite - those parts of the SQLite library that you actually bundle and ship with a larger application. Some scripts used as part of the build process (for example the "configure" scripts generated by autoconf) might fall under other open-source licenses. Nothing from these build scripts ever reaches the final deliverable SQLite library, however, and so the licenses associated with those scripts should not be a factor in assessing your rights to copy and use the SQLite library.

All of the deliverable code in SQLite has been written from scratch. No code has been taken from other projects or from the open internet. Every line of code can be traced back to its original author, and all of those authors have public domain dedications on file. So the SQLite code base is clean and is uncontaminated with licensed code from other projects.

Obtaining An License To Use SQLite

Even though SQLite is in the public domain and does not require a license, some users want to obtain a license anyway. Some reasons for obtaining a license include:

- Your company desires warranty of title and indemnity against claims of copyright infringement.

- You are using SQLite in a jurisdiction that does not recognize the public domain.
- You are using SQLite in a jurisdiction that does not recognize the right of an author to dedicate their work to the public domain.
- You want to hold a tangible legal document as evidence that you have the legal right to use and distribute SQLite.
- Your legal department tells you that you have to purchase a license.

If you feel like you really need to purchase a license for SQLite, Hwaci, the company that employs the architect and principal developers of SQLite, will sell you one. All proceeds from the sale of SQLite licenses are used to fund further improvements to SQLite.

Contributed Code

In order to keep SQLite completely free and unencumbered by copyright, all new contributors to the SQLite code base are asked to dedicate their contributions to the public domain. If you want to send a patch or enhancement for possible inclusion in the SQLite source tree, please accompany the patch with the following statement:

> *The author or authors of this code dedicate any and all copyright interest in this code to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this code under copyright law.*

We are not able to accept patches or changes to SQLite that are not accompanied by a statement such as the above. In addition, if you make changes or enhancements as an employee, then a simple statement such as the above is insufficient. You must also send by surface mail a copyright release signed by a company officer. A signed original of the copyright release should be mailed to:

> Hwaci 6200 Maple Cove Lane Charlotte, NC 28269 USA

A template copyright release is available in PDF or HTML. You can use this release to make future changes.

Frequently Asked Questions

1. How do I create an AUTOINCREMENT field.
2. What datatypes does SQLite support?
3. SQLite lets me insert a string into a database column of type integer!
4. Why doesn't SQLite allow me to use '0' and '0.0' as the primary key on two different rows of the same table?
5. Can multiple applications or multiple instances of the same application access a single database file at the same time?
6. Is SQLite threadsafe?
7. How do I list all tables/indices contained in an SQLite database
8. Are there any known size limits to SQLite databases?
9. What is the maximum size of a VARCHAR in SQLite?
10. Does SQLite support a BLOB type?
11. How do I add or delete columns from an existing table in SQLite.
12. I deleted a lot of data but the database file did not get any smaller. Is this a bug?
13. Can I use SQLite in my commercial product without paying royalties?
14. How do I use a string literal that contains an embedded single-quote (') character?
15. What is an SQLITE_SCHEMA error, and why am I getting one?
16. Why does ROUND(9.95,1) return 9.9 instead of 10.0? Shouldn't 9.95 round up?
17. I get some compiler warnings when I compile SQLite. Isn't this a problem? Doesn't it indicate poor code quality?
18. Case-insensitive matching of Unicode characters does not work.
19. INSERT is really slow - I can only do few dozen INSERTs per second
20. I accidentally deleted some important information from my SQLite database. How can I recover it?
21. What is an SQLITE_CORRUPT error? What does it mean for the database to be "malformed"? Why am I getting this error?
22. Does SQLite support foreign keys?
23. I get a compiler error if I use the SQLITE*OMIT*... compile-time options when building SQLite.
24. My WHERE clause expression `column1="column1"` does not work. It causes every row of the table to be returned, not just the rows where column1 has the value "column1".
25. How are the syntax diagrams (a.k.a. "railroad" diagrams) for SQLite generated?
26. The SQL standard requires that a UNIQUE constraint be enforced even if one or more of the columns in the constraint are NULL, but SQLite does not do this. Isn't that a bug?
27. What is the Export Control Classification Number (ECCN) for SQLite?
28. My query does not return the column name that I expect. Is this a bug?

(1) How do I create an AUTOINCREMENT field.

> Short answer: A column declared INTEGER PRIMARY KEY will autoincrement.
>
> Longer answer: If you declare a column of a table to be INTEGER PRIMARY KEY, then whenever you insert a NULL into that column of the table, the NULL is automatically converted into an integer which is one greater than the largest value of that column over all other rows in the table, or 1 if the table is empty. Or, if the largest existing integer key 9223372036854775807 is in use then an unused key value is chosen at random. For example, suppose you have a table like this:
>
> ```
> CREATE TABLE t1(
>   a INTEGER PRIMARY KEY,
>   b INTEGER
> );
> ```
>
> With this table, the statement
>
> ```
> INSERT INTO t1 VALUES(NULL,123);
> ```
>
> is logically equivalent to saying:
>
> ```
> INSERT INTO t1 VALUES((SELECT max(a) FROM t1)+1,123);
> ```
>
> There is a function named sqlite3_last_insert_rowid() which will return the integer key for the most recent insert operation.
>
> Note that the integer key is one greater than the largest key that was in the table just prior to the insert. The new key will be unique over all keys currently in the table, but it might overlap with keys that have been previously deleted from the table. To create keys that are unique over the lifetime of the table, add the AUTOINCREMENT keyword to the INTEGER PRIMARY KEY declaration. Then the key chosen will be one more than the largest key that has ever existed in that table. If the largest possible key has previously existed in that table, then the INSERT will fail with an SQLITE_FULL error code.

(2) What datatypes does SQLite support?

> SQLite uses dynamic typing. Content can be stored as INTEGER, REAL, TEXT, BLOB, or as NULL.

(3) SQLite lets me insert a string into a database column of type integer!

> This is a feature, not a bug. SQLite uses dynamic typing. It does not enforce data type constraints. Data of any type can (usually) be inserted into any column. You can put arbitrary length strings into integer columns, floating point numbers in boolean columns, or dates in character columns. The datatype you assign to a column in the CREATE TABLE command does not restrict what data can be put into that column. Every column is able to hold an arbitrary length string. (There is one exception: Columns of type INTEGER PRIMARY KEY may only hold a 64-bit signed integer. An error will result if you try to put anything other than an integer into an INTEGER PRIMARY KEY column.)
>
> But SQLite does use the declared type of a column as a hint that you prefer values in that format. So, for example, if a column is of type INTEGER and you try to insert a string into that column, SQLite will attempt to convert the string into an integer. If it can, it inserts the integer instead. If not, it inserts the string. This feature is called type affinity.

(4) Why doesn't SQLite allow me to use '0' and '0.0' as the primary key on two different rows of the same table?

> This problem occurs when your primary key is a numeric type. Change the datatype of your primary key to TEXT and it should work.
>
> Every row must have a unique primary key. For a column with a numeric type, SQLite thinks that **'0'** and **'0.0'** are the same value because they compare equal to one another numerically. (See the previous question.) Hence the values are not unique.

(5) Can multiple applications or multiple instances of the same application access a single database file at the same time?

> Multiple processes can have the same database open at the same time. Multiple processes can be doing a SELECT at the same time. But only one process can be making changes to the database at any moment in time, however.
>
> SQLite uses reader/writer locks to control access to the database. (Under Win95/98/ME which lacks support for reader/writer locks, a probabilistic simulation is used instead.) But use caution: this locking mechanism might not work correctly if the database file is kept on an NFS filesystem. This is because fcntl() file locking is broken on many NFS implementations. You should avoid putting SQLite database files on NFS if multiple processes might try to access the file at the same time. On Windows, Microsoft's documentation says that locking may not work under FAT filesystems if you are not running the Share.exe daemon. People who have a lot of experience with Windows tell me that file locking of network files is very buggy and is not dependable. If what they say is true, sharing an SQLite database between two or more Windows machines might cause unexpected problems.
>
> We are aware of no other *embedded* SQL database engine that supports as much concurrency as SQLite. SQLite allows multiple processes to have the database file open at once, and for multiple processes to read the database at once. When any process wants to write, it must lock the entire database file for the duration of its update. But that normally only takes a few milliseconds. Other processes just wait on the writer to finish then continue about their business. Other embedded SQL database engines typically only allow a single process to connect to the database at once.
>
> However, client/server database engines (such as PostgreSQL, MySQL, or Oracle) usually support a higher level of concurrency and allow multiple processes to be writing to the same database at the same time. This is possible in a client/server database because there is always a single well-controlled server process available to coordinate access. If your application has a need for a lot of concurrency, then you should consider using a client/server database. But experience suggests that most applications need much less concurrency than their designers imagine.
>
> When SQLite tries to access a file that is locked by another process, the default behavior is to return SQLITE_BUSY. You can adjust this behavior from C code using the sqlite3_busy_handler() or sqlite3_busy_timeout() API functions.

(6) Is SQLite threadsafe?

> Threads are evil. Avoid them.
>
> SQLite is threadsafe. We make this concession since many users choose to ignore the advice given in the previous paragraph. But in order to be thread-safe, SQLite must be compiled with the SQLITE_THREADSAFE preprocessor macro set to 1. Both the Windows and Linux precompiled binaries in the distribution are compiled this way. If you are unsure if the SQLite library you are linking against is compiled to be threadsafe you can call the sqlite3_threadsafe() interface to find out.
>
> SQLite is threadsafe because it uses mutexes to serialize access to common data structures. However, the work of acquiring and releasing these mutexes will slow SQLite down slightly. Hence, if you do not need SQLite to be threadsafe, you should disable the mutexes for maximum performance. See the threading mode documentation for additional information.
>
> Under Unix, you should not carry an open SQLite database across a fork() system call into the child process.

(7) How do I list all tables/indices contained in an SQLite database

> If you are running the **sqlite3** command-line access program you can type "**.tables**" to get a list of all tables. Or you can type "**.schema**" to see the complete database schema including all tables and indices. Either of these commands can be followed by a LIKE pattern that will restrict the tables that are displayed.
>
> From within a C/C++ program (or a script using Tcl/Ruby/Perl/Python bindings) you can get access to table and index names by doing a SELECT on a special table named "**SQLITE_MASTER**". Every SQLite database has an SQLITE_MASTER table that defines the schema for the database. The SQLITE_MASTER table looks like this:
>
> ```
> CREATE TABLE sqlite_master (
>   type TEXT,
>   name TEXT,
>   tbl_name TEXT,
>   rootpage INTEGER,
>   sql TEXT
> );
> ```
>
> For tables, the **type** field will always be **'table'** and the **name** field will be the name of the table. So to get a list of all tables in the database, use the following SELECT command:
>
> ```
> SELECT name FROM sqlite_master
> WHERE type='table'
> ORDER BY name;
> ```

> For indices, **type** is equal to **'index'**, **name** is the name of the index and **tbl_name** is the name of the table to which the index belongs. For both tables and indices, the **sql** field is the text of the original CREATE TABLE or CREATE INDEX statement that created the table or index. For automatically created indices (used to implement the PRIMARY KEY or UNIQUE constraints) the **sql** field is NULL.
>
> The SQLITE_MASTER table is read-only. You cannot change this table using UPDATE, INSERT, or DELETE. The table is automatically updated by CREATE TABLE, CREATE INDEX, DROP TABLE, and DROP INDEX commands.
>
> Temporary tables do not appear in the SQLITE_MASTER table. Temporary tables and their indices and triggers occur in another special table named SQLITE_TEMP_MASTER. SQLITE_TEMP_MASTER works just like SQLITE_MASTER except that it is only visible to the application that created the temporary tables. To get a list of all tables, both permanent and temporary, one can use a command similar to the following:
>
> ```
> SELECT name FROM
>    (SELECT * FROM sqlite_master UNION ALL
>     SELECT * FROM sqlite_temp_master)
> WHERE type='table'
> ORDER BY name
> ```

(8) Are there any known size limits to SQLite databases?

> See limits.html for a full discussion of the limits of SQLite.

(9) What is the maximum size of a VARCHAR in SQLite?

> SQLite does not enforce the length of a VARCHAR. You can declare a VARCHAR(10) and SQLite will be happy to store a 500-million character string there. And it will keep all 500-million characters intact. Your content is never truncated. SQLite understands the column type of "VARCHAR(*N*)" to be the same as "TEXT", regardless of the value of *N*.

(10) Does SQLite support a BLOB type?

> SQLite allows you to store BLOB data in any column, even columns that are declared to hold some other type. BLOBs can even be used as PRIMARY KEYs.

(11) How do I add or delete columns from an existing table in SQLite.

> SQLite has limited ALTER TABLE support that you can use to add a column to the end of a table or to change the name of a table. If you want to make more complex changes in the structure of a table, you will have to recreate the table. You can save existing data to a temporary table, drop the old table, create the new table, then copy the data back in from the temporary table.
>
> For example, suppose you have a table named "t1" with columns names "a", "b", and "c" and that you want to delete column "c" from this table. The following steps illustrate how this could be done:
>
> ```
> BEGIN TRANSACTION;
> CREATE TEMPORARY TABLE t1_backup(a,b);
> INSERT INTO t1_backup SELECT a,b FROM t1;
> DROP TABLE t1;
> CREATE TABLE t1(a,b);
> INSERT INTO t1 SELECT a,b FROM t1_backup;
> DROP TABLE t1_backup;
> COMMIT;
> ```

(12) I deleted a lot of data but the database file did not get any smaller. Is this a bug?

> No. When you delete information from an SQLite database, the unused disk space is added to an internal "free-list" and is reused the next time you insert data. The disk space is not lost. But neither is it returned to the operating system.
>
> If you delete a lot of data and want to shrink the database file, run the VACUUM command. VACUUM will reconstruct the database from scratch. This will leave the database with an empty free-list and a file that is minimal in size. Note, however, that the VACUUM can take some time to run and it can use up to twice as much temporary disk space as the original file while it is running.
>
> An alternative to using the VACUUM command is auto-vacuum mode, enabled using the auto_vacuum pragma.

(13) Can I use SQLite in my commercial product without paying royalties?

> Yes. SQLite is in the public domain. No claim of ownership is made to any part of the code. You can do anything you want with it.

(14) How do I use a string literal that contains an embedded single-quote (') character?

> The SQL standard specifies that single-quotes in strings are escaped by putting two single quotes in a row. SQL works like the Pascal programming language in this regard. Example:
>
> ```
>     INSERT INTO xyz VALUES('5 O''clock');
> ```

(15) What is an SQLITE_SCHEMA error, and why am I getting one?

> An SQLITE_SCHEMA error is returned when a prepared SQL statement is no longer valid and cannot be executed. When this occurs, the statement must be recompiled from SQL using the sqlite3_prepare() API. An SQLITE_SCHEMA error can only occur when using the sqlite3_prepare(), and sqlite3_step() interfaces to run SQL. You will never receive an SQLITE_SCHEMA error from sqlite3_exec(). Nor will you receive an error if you prepare statements using sqlite3_prepare_v2() instead of sqlite3_prepare().
>
> The sqlite3_prepare_v2() interface creates a prepared statement that will automatically recompile itself if the schema changes. The easiest way to deal with SQLITE_SCHEMA errors is to always use sqlite3_prepare_v2() instead of sqlite3_prepare().

(16) Why does ROUND(9.95,1) return 9.9 instead of 10.0? Shouldn't 9.95 round up?

> SQLite uses binary arithmetic and in binary, there is no way to write 9.95 in a finite number of bits. The closest to you can get to 9.95 in a 64-bit IEEE float (which is what SQLite uses) is 9.949999999999999289457264239899814128875732421875. So when you type "9.95", SQLite really understands the number to be the much longer value shown above. And that value rounds down.
>
> This kind of problem comes up all the time when dealing with floating point binary numbers. The general rule to remember is that most fractional numbers that have a finite representation in decimal (a.k.a "base-10") do not have a finite representation in binary (a.k.a "base-2"). And so they are approximated using the closest binary number available. That approximation is usually very close, but it will be slightly off and in some cases can cause your results to be a little different from what you might expect.

(17) I get some compiler warnings when I compile SQLite. Isn't this a problem? Doesn't it indicate poor code quality?

> Quality assurance in SQLite is done using full-coverage testing, not by compiler warnings or other static code analysis tools. In other words, we verify that SQLite actually gets the correct answer, not that it merely satisfies stylistic constraints. Most of the SQLite code base is devoted purely to testing. The SQLite test suite runs tens of thousands of separate test cases and many of those test cases are parameterized so that hundreds of millions of tests involving billions of SQL statements are run and evaluated for correctness prior to every release. The developers use code coverage tools to verify that all paths through the code are tested. Whenever a bug is found in SQLite, new test cases are written to exhibit the bug so that the bug cannot recur undetected in the future.
>
> During testing, the SQLite library is compiled with special instrumentation that allows the test scripts to simulate a wide variety of failures in order to verify that SQLite recovers correctly. Memory allocation is carefully tracked and no memory leaks occur, even following memory allocation failures. A custom VFS layer is used to simulate operating system crashes and power failures in order to ensure that transactions are atomic across these events. A mechanism for deliberately injecting I/O errors shows that SQLite is resilient to such malfunctions. (As an experiment, try inducing these kinds of errors on other SQL database engines and see what happens!)
>
> We also run SQLite using Valgrind on Linux and verify that it detects no problems.
>
> Some people say that we should eliminate all warnings because benign warnings mask real warnings that might arise in future changes. This is true enough. But in reply, the developers observe that all warnings have already been fixed in the builds used for SQLite development (various versions of GCC, MSVC, and clang). Compiler warnings usually only arise from compilers or compile-time options that the SQLite developers do not use themselves.

(18) Case-insensitive matching of Unicode characters does not work.

> The default configuration of SQLite only supports case-insensitive comparisons of ASCII characters. The reason for this is that doing full Unicode case-insensitive comparisons and case conversions requires tables and logic that would nearly double the size of the SQLite library. The SQLite developers reason that any application that needs full Unicode case support probably already has the necessary tables and functions and so SQLite should not take up space to duplicate this ability.
>
> Instead of providing full Unicode case support by default, SQLite provides the ability to link against external Unicode comparison and conversion routines. The application can overload the built-in NOCASE collating sequence (using sqlite3_create_collation()) and the built-in like(), upper(), and lower() functions (using sqlite3_create_function()). The SQLite source code includes an "ICU" extension that does these overloads. Or, developers can write their own overloads based on their own Unicode-aware comparison routines already contained within their project.

(19) INSERT is really slow - I can only do few dozen INSERTs per second

> Actually, SQLite will easily do 50,000 or more INSERT statements per second on an average desktop computer. But it will only do a few dozen transactions per second. Transaction speed is limited by the rotational speed of your disk drive. A transaction normally requires two complete rotations of the disk platter, which on a 7200RPM disk drive limits you to about 60 transactions per second.
>
> Transaction speed is limited by disk drive speed because (by default) SQLite actually waits until the data really is safely stored on the disk surface before the transaction is complete. That way, if you suddenly lose power or if your OS crashes, your data is still safe. For details, read about atomic commit in SQLite..
>
> By default, each INSERT statement is its own transaction. But if you surround multiple INSERT statements with BEGIN...COMMIT then all the inserts are grouped into a single transaction. The time needed to commit the transaction is amortized over all the enclosed insert statements and so the time per insert statement is greatly reduced.
>
> Another option is to run PRAGMA synchronous=OFF. This command will cause SQLite to not wait on data to reach the disk surface, which will make write operations appear to be much faster. But if you lose power in the middle of a transaction, your database file might go corrupt.

(20) I accidentally deleted some important information from my SQLite database. How can I recover it?

> If you have a backup copy of your database file, recover the information from your backup.
>
> If you do not have a backup, recovery is very difficult. You might be able to find partial string data in a binary dump of the raw database file. Recovering numeric data might also be possible given special tools, though to our knowledge no such tools exist. SQLite is sometimes compiled with the SQLITE_SECURE_DELETE option which overwrites all deleted content with zeros. If that is the case then recovery is clearly impossible. Recovery is also impossible if you have run VACUUM since the data was deleted. If SQLITE_SECURE_DELETE is not used and VACUUM has not been run, then some of the deleted content might still be in the database file, in areas marked for reuse. But, again, there exist no procedures or tools that we know of to help you recover that data.

(21) What is an SQLITE_CORRUPT error? What does it mean for the database to be "malformed"? Why am I getting this error?

> An SQLITE_CORRUPT error is returned when SQLite detects an error in the structure, format, or other control elements of the database file.
>
> SQLite does not corrupt database files, except in the case of very rare bugs (see DatabaseCorruption) and even then the bugs are normally difficult to reproduce. Even if your application crashes in the middle of an update, your database is safe. The database is safe even if your OS crashes or takes a power loss. The crash-resistance of SQLite has been extensively studied and tested and is attested by years of real-world experience by billions of users.
>
> That said, there are a number of things that external programs or bugs in your hardware or OS can do to corrupt a database file. See How To Corrupt An SQLite Database File for further information.
>
> You can use PRAGMA integrity_check to do a thorough but time intensive test of the database integrity.
>
> You can use PRAGMA quick_check to do a faster but less thorough test of the database integrity.
>
> Depending how badly your database is corrupted, you may be able to recover some of the data by using the CLI to dump the schema and contents to a file and then recreate. Unfortunately, once humpty-dumpty falls off the wall, it is generally not possible to put him back together again.

(22) Does SQLite support foreign keys?

> As of version 3.6.19, SQLite supports foreign key constraints. But enforcement of foreign key constraints is turned off by default (for backwards compatibility). To enable foreign key constraint enforcement, run PRAGMA foreign_keys=ON or compile with -DSQLITE_DEFAULT_FOREIGN_KEYS=1.

(23) I get a compiler error if I use the SQLITE*OMIT*... compile-time options when building SQLite.

> The SQLITE*OMIT*... compile-time options only work when building from canonical source files. They do not work when you build from the SQLite amalgamation or from the pre-processed source files.
>
> It is possible to build a special amalgamation that will work with a predetermined set of SQLITE*OMIT*... options. Instructions for doing so can be found with the SQLITE*OMIT*... documentation.

(24) My WHERE clause expression `column1="column1"` does not work. It causes every row of the table to be returned, not just the rows where column1 has the value "column1".

> Use single-quotes, not double-quotes, around string literals in SQL. This is what the SQL standard requires. Your WHERE clause expression should read: `column1='column1'`
>
> SQL uses double-quotes around identifiers (column or table names) that contains special characters or which are keywords. So double-quotes are a way of escaping identifier names. Hence, when you say `column1="column1"` that is equivalent to `column1=column1` which is obviously always true.

(25) How are the syntax diagrams (a.k.a. "railroad" diagrams) for SQLite generated?

> The process is explained at http://wiki.tcl-lang.org/21708.

(26) The SQL standard requires that a UNIQUE constraint be enforced even if one or more of the columns in the constraint are NULL, but SQLite does not do this. Isn't that a bug?

> Perhaps you are referring to the following statement from SQL92:
>
> > A unique constraint is satisfied if and only if no two rows in a table have the same non-null values in the unique columns.
>
> That statement is ambiguous, having at least two possible interpretations:
>
> 1. A unique constraint is satisfied if and only if no two rows in a table have the same values and have non-null values in the unique columns.
> 2. A unique constraint is satisfied if and only if no two rows in a table have the same values in the subset of unique columns that are not null.
>
> SQLite follows interpretation (1), as does PostgreSQL, MySQL, Oracle, and Firebird. It is true that Informix and Microsoft SQL Server use interpretation (2), however we the SQLite developers hold that interpretation (1) is the most natural reading of the requirement and we also want to maximize compatibility with other SQL database engines, and most other database engines also go with (1), so that is what SQLite does.

(27) What is the Export Control Classification Number (ECCN) for SQLite?

> After careful review of the Commerce Control List (CCL), we are convinced that the core public-domain SQLite source code is not described by any ECCN, hence the ECCN should be reported as **EAR99**.
>
> The above is true for the core public-domain SQLite. If you extend SQLite by adding new code, or if you statically link SQLite with your application, that might change the ECCN in your particular case.

(28) My query does not return the column name that I expect. Is this a bug?

> If the columns of your result set are named by AS clauses, then SQLite is guaranteed to use the identifier to the right of the AS keyword as the column name. If the result set does not use an AS clause, then SQLite is free to name the column anything it wants. See the sqlite3_column_name() documentation for further information.

Books About SQLite

SQLite Database System Design and Implementation (2015)

Author: Sibsankar Haldar Publisher: https://books.google.com/ This book provides a comprehensive description of SQLite database system. It describes design principles, engineering trade-offs, implementation issues, and operations of SQLite. |

Android SQLite Essentials (2014)

Authors: Sunny Kumar Aditya and Vikash Kumar Karn Publisher: Packt Publishing Android SQLite Essentials focuses on the core concepts behind building database-driven applications. This book covers the basic and advanced topics with equivalent simplicity and detail, in order to enable readers to quickly grasp and implement the concepts to build an application database.This book takes a hands-on, example-based approach to help readers understand the core topics of SQLite and Android database-driven applications. This book focuses on providing you with latent as well as widespread knowledge about practices and approaches towards development in an easily understandable manner. |

SQLite for Mobile Apps Simplified (2014)

Author: Sribatsa Das Publisher: Amazon AmazonSQLite for Mobile Apps Simplified is devoted to presenting approach and implementation methodology for using SQLite database in mobile apps. It presents step-by-step examples to create schema, execute transactions and access data from Android, BlackBerry and iOS applications. In addition, it presents ADB

Shell and SQLite command-line shell from ADB Shell to access the SQLite Database created by the Android apps. For BlackBerry and iOS application, the book presents ways to access the data using the command line shell. |

The Definitive Guide to SQLite (2nd edition, 2010)

Authors: Mike Owens and Grant Allen Publisher: Apress AmazonOutside of the world of enterprise computing, there is one database that enables a huge range of software and hardware to flex relational database capabilities, without the baggage and cost of traditional database management systems. That database is SQLite - an embeddable database with an amazingly small footprint, yet able to handle databases of enormous size. SQLite comes equipped with an array of powerful features available through a host of programming and development environments. It is supported by languages such as C, Java, Perl, PHP, Python, Ruby, TCL, and more.*The Definitive Guide to SQLite, Second Edition* is devoted to complete coverage of the latest version of this powerful database. It offers a thorough overview of SQLite\u2019s capabilities and APIs. The book also uses SQLite as the basis for helping newcomers make their first foray into database development. In only a short time you can be writing programs as diverse as a server-side browser plug-in or the next great iPhone or Android application! |

Using SQLite (2010)

Author: Jay A. Kreibich Publisher: O'Reilly Media O'ReillyDevelopers, take note: databases aren't just for the IS group any more. You can build database-backed applications for the desktop, Web, embedded systems, or operating systems without linking to heavy-duty client-server databases such as Oracle and MySQL. This book shows how you to use SQLite, a small and lightweight database that you can build right into your application during development. Applications that handle data have an enormous advantage today, and with SQLite, you'll discover how to develop a database-backed application that remains manageable in size and complexity. This book guides you every step of the way. You'll get a crash course in data modeling, become familiar with SQLite's dialect of the SQL database language, and learn how you to work with SQLite using either a scripting language or a C-based language, such as C# or Objective C.Now, even relatively small and nimble applications can be a part of the data revolution. Using SQLite shows you how. |

SQLite 3 - Einstieg in die Datenbankwelt (2010)

Author: Key Droessler Publisher: Lulu.com AmazonDie Datenbanksprache SQL (Structured Query Language) wird in Datenbanken zur Definition, Manipulation, Sicherung, aber hauptsaechlich zur Abfrage von Daten aus der Datenbank eingesetzt. Unabhaengig vom Betriebssystem oder aufwendigen, benutzerfreundlichen, graphischen Oberflaechen bleibt die Logik aber immer gleich.SQLite ist eine freie Desktop-Datenbank, sie kostet nichts, ist fuer viele Betriebssysteme verfuegbar, schnell heruntergeladen und installiert und auf das Notwendigste reduziert. Fuer den Einsteiger sind das die besten Voraussetzungen, ohne viel Aufwand schnell in die Welt der Datenbanken und Datenbanksprache reinzuschnuppern.Wer nach den Uebungen aber auf den Geschmack gekommen ist, hat schon den groessten Teil an Datenbanken und SQL gelernt, denn alles Besprochene ist Wissen, welches auch auf jedes andere der vielen Datenbanken grundlegend anwendbar ist. Nun koennen Sie auf die richtig Grossen zugehen, vom grossen Fachbuch bis zum riesigen Datenbanksystem. |

Inside Symbian SQL (2010)

Authors: Ivan Litovski & Richard Maynard Publisher: Wiley wiley.comThis is the definitive reference book on the Symbian SQL database which is based on SQLite. The authors (both members of the Symbian engineering team responsible for the implementation of the code) show you how to design code and ease migration from an internal and external point of view, plus they reveal the dos and don'ts of writing high-performance database applications. Packed with resources and sample code, this timely book reveals how to design and tune applications that use the Symbian SQL framework to ultimately improve performance.With its sample code and insider expertise, this text has everything you need to keep you ahead of the curve. |

The SQL Guide to SQLite (2009)

Author: Rick F. van der Lans Publisher: Lulu.com AmazonSQLite is a small, fast, embeddable, SQL-based database server. It is easy to install, needs no management, and is open source. This book describes SQLite in detail. With hundreds of examples, plus a proven approach and structure, the book teaches you how to use SQLite efficiently and effectively. It contains a complete description of the SQL dialect as implemented in SQLite version 3.6. The book can be seen as a tutorial and a reference book. Source code for the numerous SQL examples and exercises included in this book can be downloaded from www.r20.nl. |

An Introduction to SQLite - 2nd Edition (2009)

Author: Naoki Nishizawa Publisher: Shoeisha Amazon.jpThis text is written in fluent Japanese specifically for a Japanese audience. This is the second edition of the book - the first edition was published in 2005. |

Inside SQLite (2007)

Author: Sibsankar Haldar Publisher: O'Reilly Media O'ReillySQLite is a small, zero-configuration, custom-tailored, embeddable, thread-safe, easily maintainable, transaction-oriented, SQL-based, relational database management system. There is no separate install or setup procedure to initialize SQLite before using it. There is no configuration file. SQLite is open source, and is available in the public domain (for more information on open source, visit http://opensource.org). You can download SQLite source code from its homepage http://www.sqlite.org, compile it using your favorite C compiler, and start using the compiled library. SQLite runs on Linux, Windows, Mac OS X, and a few other operating systems. It has been widely used in low-to-medium tier database applications. This Short Cut discusses design principles, engineering trade-offs, implementation issues, and operations of SQLite. It presents a comprehensive description of all important components of the SQLite engine. |

SQLite (2004)

Author: Chris Newman Publisher: Sams AmazonSQLite is a small, fast, embeddable database. What makes it popular is the combination of the database engine and interface into a single library as well as the ability to store all the data in a single file. Its functionality lies between MySQL and PostgreSQL, however it is faster than both databases.In *SQLite*, author Chris Newman provides a thorough, practical guide to using, administering and programming this up-and-coming database. If you want to learn about SQLite or about its use in conjunction with PHP this is the book for you. |

Alphabetical List Of SQLite Documents

See Also:

- Categorical Document List
- Books About SQLite
- Permuted Title Index
- Website Keyword Index

1. 8+3 Filenames
2. About SQLite
3. Alphabetical List Of SQLite Documents
4. An Asynchronous I/O Module For SQLite
5. An Introduction To The SQLite C/C++ Interface
6. Appropriate Uses For SQLite
7. Architecture of SQLite
8. Atomic Commit In SQLite
9. Automatic Undo/Redo With SQLite
10. Benefits of SQLite As A File Format
11. Books About SQLite
12. C/C++ Interface For SQLite Version 3
13. C/C++ Interface For SQLite Version 3 (old)
14. Change in Default Page Size in SQLite Version 3.12.0
15. Chronology Of SQLite Releases
16. Command Line Shell For SQLite
17. Compilation Options For SQLite
18. Constraint Conflict Resolution in SQLite
19. Custom Builds Of SQLite
20. Datatypes In SQLite version 2
21. Datatypes In SQLite Version 3
22. Distinctive Features Of SQLite
23. Dynamic Memory Allocation In SQLite
24. EXPLAIN QUERY PLAN
25. Features Of SQLite
26. File Format Changes in SQLite
27. File Format For SQLite Databases
28. File Locking And Concurrency In SQLite Version 3
29. How SQLite Is Tested
30. How To Compile SQLite

11. Website Keyword Index
12. Well-Known Users Of SQLite
13. What If OpenDocument Used SQLite?
14. Write-Ahead Logging
15. Zero-Configuration

Website Keyword Index

Other Documentation Indices:

- Categorical Document List
- Books About SQLite
- Alphabetical List Of Documents
- Permuted Document Title Index

- -DSQLITE_ENABLE_STMT_SCANSTATUS
- -DSQLITE_ENABLE_TREE_EXPLAIN
- -DSQLITE_ENABLE_UNLOCK_NOTIFY
- -DSQLITE_ENABLE_UPDATE_DELETE_LIMIT
- -DSQLITE_EXTRA_DURABLE
- -DSQLITE_FTS3_MAX_EXPR_DEPTH
- -DSQLITE_HAVE_ISNAN
- -DSQLITE_LIKE_DOESNT_MATCH_BLOBS
- -DSQLITE_MAX_MMAP_SIZE
- -DSQLITE_MAX_SCHEMA_RETRY
- -DSQLITE_MAX_WORKER_THREADS
- -DSQLITE_MEMDEBUG
- -DSQLITE_MINIMUM_FILE_DESCRIPTOR
- -DSQLITE_OMIT_ALTERTABLE
- -DSQLITE_OMIT_ANALYZE
- -DSQLITE_OMIT_ATTACH
- -DSQLITE_OMIT_AUTHORIZATION
- -DSQLITE_OMIT_AUTOINCREMENT
- -DSQLITE_OMIT_AUTOINIT
- -DSQLITE_OMIT_AUTOMATIC_INDEX
- -DSQLITE_OMIT_AUTORESET
- -DSQLITE_OMIT_AUTOVACUUM
- -DSQLITE_OMIT_BETWEEN_OPTIMIZATION
- -DSQLITE_OMIT_BLOB_LITERAL
- -DSQLITE_OMIT_BTREECOUNT
- -DSQLITE_OMIT_BUILTIN_TEST
- -DSQLITE_OMIT_CAST
- -DSQLITE_OMIT_CHECK
- -DSQLITE_OMIT_COMPILEOPTION_DIAGS
- -DSQLITE_OMIT_COMPLETE
- -DSQLITE_OMIT_COMPOUND_SELECT
- -DSQLITE_OMIT_CTE
- -DSQLITE_OMIT_DATETIME_FUNCS
- -DSQLITE_OMIT_DECLTYPE
- -DSQLITE_OMIT_DEPRECATED
- -DSQLITE_OMIT_DISKIO
- -DSQLITE_OMIT_EXPLAIN
- -DSQLITE_OMIT_FLAG_PRAGMAS
- -DSQLITE_OMIT_FLOATING_POINT
- -DSQLITE_OMIT_FOREIGN_KEY

- -DSQLITE_OMIT_GET_TABLE
- -DSQLITE_OMIT_INCRBLOB
- -DSQLITE_OMIT_INTEGRITY_CHECK
- -DSQLITE_OMIT_LIKE_OPTIMIZATION
- -DSQLITE_OMIT_LOAD_EXTENSION
- -DSQLITE_OMIT_LOCALTIME
- -DSQLITE_OMIT_LOOKASIDE
- -DSQLITE_OMIT_MEMORYDB
- -DSQLITE_OMIT_OR_OPTIMIZATION
- -DSQLITE_OMIT_PAGER_PRAGMAS
- -DSQLITE_OMIT_PRAGMA
- -DSQLITE_OMIT_PROGRESS_CALLBACK
- -DSQLITE_OMIT_QUICKBALANCE
- -DSQLITE_OMIT_REINDEX
- -DSQLITE_OMIT_SCHEMA_PRAGMAS
- -DSQLITE_OMIT_SCHEMA_VERSION_PRAGMAS
- -DSQLITE_OMIT_SHARED_CACHE
- -DSQLITE_OMIT_SUBQUERY
- -DSQLITE_OMIT_TCL_VARIABLE
- -DSQLITE_OMIT_TEMPDB
- -DSQLITE_OMIT_TRACE
- -DSQLITE_OMIT_TRIGGER
- -DSQLITE_OMIT_TRUNCATE_OPTIMIZATION
- -DSQLITE_OMIT_UTF16
- -DSQLITE_OMIT_VACUUM
- -DSQLITE_OMIT_VIEW
- -DSQLITE_OMIT_VIRTUALTABLE
- -DSQLITE_OMIT_WAL
- -DSQLITE_OMIT_WSD
- -DSQLITE_OMIT_XFER_OPT
- -DSQLITE_OS_OTHER
- -DSQLITE_POWERSAFE_OVERWRITE
- -DSQLITE_REVERSE_UNORDERED_SELECTS
- -DSQLITE_RTREE_INT_ONLY
- -DSQLITE_SECURE_DELETE
- -DSQLITE_SORTER_PMASZ
- -DSQLITE_SOUNDEX
- -DSQLITE_STMTJRNL_SPILL
- -DSQLITE_TEMP_STORE
- -DSQLITE_THREADSAFE

- sqlite3_clear_bindings
- sqlite3_close
- sqlite3_close_v2
- sqlite3_collation_needed
- sqlite3_collation_needed16
- sqlite3_column_blob
- sqlite3_column_bytes
- sqlite3_column_bytes16
- sqlite3_column_count
- sqlite3_column_database_name
- sqlite3_column_database_name16
- sqlite3_column_decltype
- sqlite3_column_decltype16
- sqlite3_column_double
- sqlite3_column_int
- sqlite3_column_int64
- sqlite3_column_name
- sqlite3_column_name16
- sqlite3_column_origin_name
- sqlite3_column_origin_name16
- sqlite3_column_table_name
- sqlite3_column_table_name16
- sqlite3_column_text
- sqlite3_column_text16
- sqlite3_column_type
- sqlite3_column_value
- sqlite3_commit_hook
- sqlite3_compileoption_get
- sqlite3_compileoption_used
- sqlite3_complete
- sqlite3_complete16
- sqlite3_config
- sqlite3_context
- sqlite3_context_db_handle
- sqlite3_create_collation
- sqlite3_create_collation16
- sqlite3_create_collation_v2
- sqlite3_create_function
- sqlite3_create_function16
- sqlite3_create_function_v2

- sqlite3_next_stmt
- sqlite3_open
- sqlite3_open16
- sqlite3_open_v2
- sqlite3_os_end
- sqlite3_os_init
- sqlite3_overload_function
- sqlite3_pcache
- sqlite3_pcache_methods2
- sqlite3_pcache_page
- sqlite3_prepare
- sqlite3_prepare16
- sqlite3_prepare16_v2
- sqlite3_prepare_v2
- sqlite3_profile
- sqlite3_progress_handler
- sqlite3_randomness
- sqlite3_realloc
- sqlite3_realloc64
- sqlite3_release_memory
- sqlite3_reset
- sqlite3_reset_auto_extension
- sqlite3_result_blob
- sqlite3_result_blob64
- sqlite3_result_double
- sqlite3_result_error
- sqlite3_result_error16
- sqlite3_result_error_code
- sqlite3_result_error_nomem
- sqlite3_result_error_toobig
- sqlite3_result_int
- sqlite3_result_int64
- sqlite3_result_null
- sqlite3_result_subtype
- sqlite3_result_text
- sqlite3_result_text16
- sqlite3_result_text16be
- sqlite3_result_text16le
- sqlite3_result_text64
- sqlite3_result_value

- SQLITE_ABORT
- SQLITE_ABORT_ROLLBACK
- SQLITE_ACCESS_EXISTS
- SQLITE_ACCESS_READ
- SQLITE_ACCESS_READWRITE
- SQLITE_ALLOW_COVERING_INDEX_SCAN
- SQLITE_ALLOW_URI_AUTHORITY
- SQLITE_ALTER_TABLE
- SQLITE_ANALYZE
- SQLITE_ANY
- SQLITE_ATTACH
- SQLITE_AUTH
- SQLITE_AUTH_USER
- SQLITE_BLOB
- SQLITE_BUSY
- SQLITE_BUSY_RECOVERY
- SQLITE_BUSY_SNAPSHOT
- SQLITE_CANTOPEN
- SQLITE_CANTOPEN_CONVPATH
- SQLITE_CANTOPEN_FULLPATH
- SQLITE_CANTOPEN_ISDIR
- SQLITE_CANTOPEN_NOTEMPDIR
- SQLITE_CASE_SENSITIVE_LIKE
- SQLITE_CHECKPOINT_FULL
- SQLITE_CHECKPOINT_PASSIVE
- SQLITE_CHECKPOINT_RESTART
- SQLITE_CHECKPOINT_TRUNCATE
- sqlite_compileoption_get() SQL function
- sqlite_compileoption_used() SQL function
- SQLITE_CONFIG_COVERING_INDEX_SCAN
- SQLITE_CONFIG_GETMALLOC
- SQLITE_CONFIG_GETMUTEX
- SQLITE_CONFIG_GETPCACHE
- SQLITE_CONFIG_GETPCACHE2
- SQLITE_CONFIG_HEAP
- SQLITE_CONFIG_LOG
- SQLITE_CONFIG_LOOKASIDE
- SQLITE_CONFIG_MALLOC
- SQLITE_CONFIG_MEMSTATUS
- SQLITE_CONFIG_MMAP_SIZE

- SQLITE_OMIT_SCHEMA_VERSION_PRAGMAS
- SQLITE_OMIT_SHARED_CACHE
- SQLITE_OMIT_SUBQUERY
- SQLITE_OMIT_TCL_VARIABLE
- SQLITE_OMIT_TEMPDB
- SQLITE_OMIT_TRACE
- SQLITE_OMIT_TRIGGER
- SQLITE_OMIT_TRUNCATE_OPTIMIZATION
- SQLITE_OMIT_UTF16
- SQLITE_OMIT_VACUUM
- SQLITE_OMIT_VIEW
- SQLITE_OMIT_VIRTUALTABLE
- SQLITE_OMIT_WAL
- SQLITE_OMIT_WSD
- SQLITE_OMIT_XFER_OPT
- SQLITE_OPEN_AUTOPROXY
- SQLITE_OPEN_CREATE
- SQLITE_OPEN_DELETEONCLOSE
- SQLITE_OPEN_EXCLUSIVE
- SQLITE_OPEN_FULLMUTEX
- SQLITE_OPEN_MAIN_DB
- SQLITE_OPEN_MAIN_JOURNAL
- SQLITE_OPEN_MASTER_JOURNAL
- SQLITE_OPEN_MEMORY
- SQLITE_OPEN_NOMUTEX
- SQLITE_OPEN_PRIVATECACHE
- SQLITE_OPEN_READONLY
- SQLITE_OPEN_READWRITE
- SQLITE_OPEN_SHAREDCACHE
- SQLITE_OPEN_SUBJOURNAL
- SQLITE_OPEN_TEMP_DB
- SQLITE_OPEN_TEMP_JOURNAL
- SQLITE_OPEN_TRANSIENT_DB
- SQLITE_OPEN_URI
- SQLITE_OPEN_WAL
- SQLITE_OS_OTHER
- SQLITE_PERM
- SQLITE_POWERSAFE_OVERWRITE
- SQLITE_PRAGMA
- SQLITE_PROTOCOL

- Version 3.6.13
- Version 3.6.14
- Version 3.6.14.1
- Version 3.6.14.2
- Version 3.6.15
- Version 3.6.16
- Version 3.6.16.1
- Version 3.6.17
- Version 3.6.18
- Version 3.6.19
- Version 3.6.2
- Version 3.6.20
- Version 3.6.21
- Version 3.6.22
- Version 3.6.23
- Version 3.6.23.1
- Version 3.6.3
- Version 3.6.4
- Version 3.6.5
- Version 3.6.6
- Version 3.6.6.1
- Version 3.6.6.2
- Version 3.6.7
- Version 3.6.8
- Version 3.6.9
- Version 3.7.0
- Version 3.7.0.1
- Version 3.7.1
- Version 3.7.10
- Version 3.7.11
- Version 3.7.12
- Version 3.7.12.1
- Version 3.7.13
- Version 3.7.14
- Version 3.7.14.1
- Version 3.7.15
- Version 3.7.15.1
- Version 3.7.15.2
- Version 3.7.16
- Version 3.7.16.1

- Version 3.7.16.2
- Version 3.7.17
- Version 3.7.2
- Version 3.7.3
- Version 3.7.4
- Version 3.7.5
- Version 3.7.6
- Version 3.7.6.1
- Version 3.7.6.2
- Version 3.7.6.3
- Version 3.7.7
- Version 3.7.7.1
- Version 3.7.8
- Version 3.7.9
- Version 3.8.0
- Version 3.8.0.1
- Version 3.8.0.2
- Version 3.8.1
- Version 3.8.10
- Version 3.8.10.1
- Version 3.8.10.2
- Version 3.8.11
- Version 3.8.11.1
- Version 3.8.2
- Version 3.8.3
- Version 3.8.3.1
- Version 3.8.4
- Version 3.8.4.1
- Version 3.8.4.2
- Version 3.8.4.3
- Version 3.8.5
- Version 3.8.6
- Version 3.8.7
- Version 3.8.7.1
- Version 3.8.7.2
- Version 3.8.7.3
- Version 3.8.7.4
- Version 3.8.8
- Version 3.8.8.1
- Version 3.8.8.2

Permuted Title Index

Popular Pages:

- Home
- Features
- Frequently Asked Questions
- Well-known Users
- Getting Started
- When To Use SQLite
- Distinctive Features
- Alphabetical list of docs
- Books About SQLite
- Website Keyword Index

- Copyright
- SQL Syntax
    - Pragmas
    - SQL functions
    - Date & time functions
    - Aggregate functions
- C/C++ Interface Spec
    - Introduction
    - List of C-language APIs
- The TCL Interface Spec

- Mailing Lists
- News
- Report a Bug
- Version control
    - Source code timeline
    - Documentation timeline

Permuted Index: <small>(what is this?)</small>

Other Documentation Indices:

- Categorical Document List
- Books About SQLite

- Alphabetical List Of Documents
- Website Keyword Index

SQLite Programming Interfaces

SQLite In 5 Minutes Or Less

Here is what you do to start experimenting with SQLite without having to do a lot of tedious reading and configuration:

Download The Code

- Get a copy of the prebuilt binaries for your machine, or get a copy of the sources and compile them yourself. Visit the download page for more information.

Create A New Database

- At a shell or DOS prompt, enter: "**sqlite3 test.db**". This will create a new database named "test.db". (You can use a different name if you like.)
- Enter SQL commands at the prompt to create and populate the new database.
- Additional documentation is available here

Write Programs That Use SQLite

- Below is a simple TCL program that demonstrates how to use the TCL interface to SQLite. The program executes the SQL statements given as the second argument on the database defined by the first argument. The commands to watch for are the **sqlite3** command on line 7 which opens an SQLite database and creates a new object named "**db**" to access that database, the use of the eval method on the **db** object on line 8 to run SQL commands against the database, and the closing of the database connection on the last line of the script.

```
01  #!/usr/bin/tclsh
02  if {$argc!=2} {
03    puts stderr "Usage: %s DATABASE SQL-STATEMENT"
04    exit 1
05  }
06  package require sqlite3
07  sqlite3 db [lindex $argv 0]
08  db eval [lindex $argv 1] x {
09    foreach v $x(*) {
10      puts "$v = $x($v)"
11    }
12    puts ""
13  }
14  db close
```

- Below is a simple C program that demonstrates how to use the C/C++ interface to SQLite. The name of a database is given by the first argument and the second argument is one or more SQL statements to execute against the database. The function calls to pay attention to here are the call to sqlite3_open() on line 22 which opens the database, sqlite3_exec() on line 28 that executes SQL commands against the database, and sqlite3_close() on line 33 that closes the database connection.

  See also the Introduction To The SQLite C/C++ Interface for an introductory overview and roadmap to the dozens of SQLite interface functions.

  ```
  01  #include &lt;stdio.h&gt;
  02  #include &lt;sqlite3.h&gt;
  03
  04  static int callback(void *NotUsed, int argc, char **argv, char **azColName){
  05    int i;
  06    for(i=0; i&lt;argc; i++){
  07      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
  08    }
  09    printf("\n");
  10    return 0;
  11  }
  12
  13  int main(int argc, char **argv){
  14    sqlite3 *db;
  15    char *zErrMsg = 0;
  16    int rc;
  17
  18    if( argc!=3 ){
  19      fprintf(stderr, "Usage: %s DATABASE SQL-STATEMENT\n", argv[0]);
  20      return(1);
  21    }
  22    rc = sqlite3_open(argv[1], &db);
  23    if( rc ){
  24      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
  25      sqlite3_close(db);
  26      return(1);
  27    }
  28    rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
  29    if( rc!=SQLITE_OK ){
  30      fprintf(stderr, "SQL error: %s\n", zErrMsg);
  31      sqlite3_free(zErrMsg);
  32    }
  33    sqlite3_close(db);
  34    return 0;
  35  }
  ```

  See the How To Compile SQLite document for instructions and hints on how to compile the program shown above.

An Introduction To The SQLite C/C++ Interface

1.0 Executive Summary

The following two objects and eight methods comprise the essential elements of the SQLite interface:

sqlite3	The database connection object. Created by sqlite3_open() and destroyed by sqlite3_close().
sqlite3_stmt	The prepared statement object. Created by sqlite3_prepare() and destroyed by sqlite3_finalize().
sqlite3_open()	Open a connection to a new or existing SQLite database. The constructor for sqlite3.
sqlite3_prepare()	Compile SQL text into byte-code that will do the work of querying or updating the database. The constructor for sqlite3_stmt.
sqlite3_bind()	Store application data into parameters of the original SQL.
sqlite3_step()	Advance an sqlite3_stmt to the next result row or to completion.
sqlite3_column()	Column values in the current result row for an sqlite3_stmt.
sqlite3_finalize()	Destructor for sqlite3_stmt.
sqlite3_close()	Destructor for sqlite3.
sqlite3_exec()	A wrapper function that does sqlite3_prepare(), sqlite3_step(), sqlite3_column(), and sqlite3_finalize() for a string of one or more SQL statements.

2.0 Introduction

SQLite currently has over 200 distinct APIs. This can be overwhelming to a new programmer. Fortunately, most of the interfaces are very specialized and need not be considered by beginners. The core API is small, simple, and easy to learn. This article summarizes the core API.

A separate document, The SQLite C/C++ Interface, provides detailed specifications for all C/C++ APIs for SQLite. Once the reader understands the basic principles of operation for SQLite, that document should be used as a reference guide. This article is intended as introduction only and is neither a complete nor authoritative reference for the SQLite API.

3.0 Core Objects And Interfaces

The principal task of an SQL database engine is to evaluate SQL statements of SQL. To accomplish this, the developer needs two objects:

- The database connection object: sqlite3
- The prepared statement object: sqlite3_stmt

Strictly speaking, the prepared statement object is not required since the convenience wrapper interfaces, sqlite3_exec or sqlite3_get_table, can be used and these convenience wrappers encapsulate and hide the prepared statement object. Nevertheless, an understanding of prepared statements is needed to make full use of SQLite.

The database connection and prepared statement objects are controlled by a small set of C/C++ interface routine listed below.

- sqlite3_open()
- sqlite3_prepare()
- sqlite3_step()
- sqlite3_column()
- sqlite3_finalize()
- sqlite3_close()

Note that the list of routines above is conceptual rather than actual. Many of these routines come in multiple versions. For example, the list above shows a single routine named sqlite3_open() when in fact there are three separate routines that accomplish the same thing in slightly different ways: sqlite3_open(), sqlite3_open16() and sqlite3_open_v2(). The list mentions sqlite3_column() when in fact no such routine exists. The "sqlite3_column()" shown in the list is place holders for an entire family of routines to be used for extracting column data in various datatypes.

Here is a summary of what the core interfaces do:

sqlite3_open()	This routine opens a connection to an SQLite database file and returns a database connection object. This is often the first SQLite API call that an application makes and is a prerequisite for most other SQLite APIs. Many SQLite interfaces require a pointer to the database connection object as their first parameter and can be thought of as methods on the database connection object. This routine is the constructor for the database connection object.
	This routine converts SQL text into a prepared statement object and returns a pointer to that object. This interface requires a database connection pointer created by a prior call to sqlite3_open() and a text string containing the SQL

sqlite3_prepare()	statement to be prepared. This API does not actually evaluate the SQL statement. It merely prepares the SQL statement for evaluation.Think of each SQL statement as a small computer program. The purpose of sqlite3_prepare() is to compile that program into object code. The prepared statement is the object code. The sqlite3_step() interface then runs the object code to get a result.New applications should always invoke sqlite3_prepare_v2() instead of sqlite3_prepare(). The older sqlite3_prepare() is retained for backwards compatibility. But sqlite3_prepare_v2() provides a much better interface.
sqlite3_step()	This routine is used to evaluate a prepared statement that has been previously created by the sqlite3_prepare() interface. The statement is evaluated up to the point where the first row of results are available. To advance to the second row of results, invoke sqlite3_step() again. Continue invoking sqlite3_step() until the statement is complete. Statements that do not return results (ex: INSERT, UPDATE, or DELETE statements) run to completion on a single call to sqlite3_step().
sqlite3_column()	This routine returns a single column from the current row of a result set for a prepared statement that is being evaluated by sqlite3_step(). Each time sqlite3_step() stops with a new result set row, this routine can be called multiple times to find the values of all columns in that row.As noted above, there really is no such thing as a "sqlite3_column()" function in the SQLite API. Instead, what we here call "sqlite3_column()" is a place-holder for an entire family of functions that return a value from the result set in various data types. There are also routines in this family that return the size of the result (if it is a string or BLOB) and the number of columns in the result set.
sqlite3_column_blob()	
sqlite3_column_bytes()	
sqlite3_column_bytes16()	
sqlite3_column_count()	
sqlite3_column_double()	
sqlite3_column_int()	
sqlite3_column_int64()	
sqlite3_column_text()	
sqlite3_column_text16()	
sqlite3_column_type()	
sqlite3_column_value()	

sqlite3_finalize()	This routine destroys a prepared statement created by a prior call to sqlite3_prepare(). Every prepared statement must be destroyed using a call to this routine in order to avoid memory leaks.
sqlite3_close()	This routine closes a database connection previously opened by a call to sqlite3_open(). All prepared statements associated with the connection should be finalized prior to closing the connection.

3.1 Typical Usage Of Core Routines And Objects

An application will typically use sqlite3_open() to create a single database connection during initialization. Note that sqlite3_open() can be used to either open existing database files or to create and open new database files. While many applications use only a single database connection, there is no reason why an application cannot call sqlite3_open() multiple times in order to open multiple database connections - either to the same database or to different databases. Sometimes a multi-threaded application will create separate database connections for each threads. Note that a single database connection can access two or more databases using the ATTACH SQL command, so it is not necessary to have a separate database connection for each database file.

Many applications destroy their database connections using calls to sqlite3_close() at shutdown. Or, for example, an application that uses SQLite as its application file format might open database connections in response to a File/Open menu action and then destroy the corresponding database connection in response to the File/Close menu.

To run an SQL statement, the application follows these steps:

1. Create a prepared statement using sqlite3_prepare().
2. Evaluate the prepared statement by calling sqlite3_step() one or more times.
3. For queries, extract results by calling sqlite3_column() in between two calls to sqlite3_step().
4. Destroy the prepared statement using sqlite3_finalize().

The foregoing is all one really needs to know in order to use SQLite effectively. All the rest is optimization and detail.

4.0 Convenience Wrappers Around Core Routines

The sqlite3_exec() interface is a convenience wrapper that carries out all four of the above steps with a single function call. A callback function passed into sqlite3_exec() is used to process each row of the result set. The sqlite3_get_table() is another convenience wrapper that does all four of the above steps. The sqlite3_get_table() interface differs from sqlite3_exec() in that it stores the results of queries in heap memory rather than invoking a callback.

It is important to realize that neither sqlite3_exec() nor sqlite3_get_table() do anything that cannot be accomplished using the core routines. In fact, these wrappers are implemented purely in terms of the core routines.

5.0 Binding Parameters and Reusing Prepared Statements

In prior discussion, it was assumed that each SQL statement is prepared once, evaluated, then destroyed. However, SQLite allows the same prepared statement to be evaluated multiple times. This is accomplished using the following routines:

- sqlite3_reset()
- sqlite3_bind()

After a prepared statement has been evaluated by one or more calls to sqlite3_step(), it can be reset in order to be evaluated again by a call to sqlite3_reset(). Think of sqlite3_reset() as rewinding the prepared statement program back to the beginning. Using sqlite3_reset() on an existing prepared statement rather than creating a new prepared statement avoids unnecessary calls to sqlite3_prepare(). For many SQL statements, the time needed to run sqlite3_prepare() equals or exceeds the time needed by sqlite3_step(). So avoiding calls to sqlite3_prepare() can give a significant performance improvement.

It is not commonly useful to evaluate the *exact* same SQL statement more than once. More often, one wants to evaluate similar statements. For example, you might want to evaluate an INSERT statement multiple times with different values. Or you might want to evaluate the same query multiple times using a different key in the WHERE clause. To accommodate this, SQLite allows SQL statements to contain parameters which are "bound" to values prior to being evaluated. These values can later be changed and the same prepared statement can be evaluated a second time using the new values.

SQLite allows a parameter wherever a string literal, numeric constant, or NULL is allowed. (Parameters may not be used for column or table names.) A parameter takes one of the following forms:

- **?**
- **?***NNN*
- **:***AAA*
- **$***AAA*

- *@AAA*

In the examples above, *NNN* is an integer value and *AAA* is an identifier. A parameter initially has a value of NULL. Prior to calling sqlite3_step() for the first time or immediately after sqlite3_reset(), the application can invoke the sqlite3_bind() interfaces to attach values to the parameters. Each call to sqlite3_bind() overrides prior bindings on the same parameter.

An application is allowed to prepare multiple SQL statements in advance and evaluate them as needed. There is no arbitrary limit to the number of outstanding prepared statements. Some applications call sqlite3_prepare() multiple times at start-up to create all of the prepared statements they will ever need. Other applications keep a cache of the most recently used prepared statements and then reuse prepared statements out of the cache when available. Another approach is to only reuse prepared statements when they are inside of a loop.

6.0 Configuring SQLite

The default configuration for SQLite works great for most applications. But sometimes developers want to tweak the setup to try to squeeze out a little more performance, or take advantage of some obscure feature.

The sqlite3_config() interface is used to make global, process-wide configuration changes for SQLite. The sqlite3_config() interface must be called before any database connections are created. The sqlite3_config() interface allows the programmer to do things like:

- Adjust how SQLite does memory allocation, including setting up alternative memory allocators appropriate for safety-critical real-time embedded systems and application-defined memory allocators.
- Set up a process-wide error log.
- Specify an application-defined page cache.
- Adjust the use of mutexes so that they are appropriate for various threading models, or substitute an application-defined mutex system.

After process-wide configuration is complete and database connections have been created, individual database connections can be configured using calls to sqlite3_limit() and sqlite3_db_config().

7.0 Extending SQLite

SQLite includes interfaces that can be used to extend its functionality. Such routines include:

- sqlite3_create_collation()
- sqlite3_create_function()
- sqlite3_create_module()
- sqlite3_vfs_register()

The sqlite3_create_collation() interface is used to create new collating sequences for sorting text. The sqlite3_create_module() interface is used to register new virtual table implementations. The sqlite3_vfs_register() interface creates new VFSes.

The sqlite3_create_function() interface creates new SQL functions - either scalar or aggregate. The new function implementation typically makes use of the following additional interfaces:

- sqlite3_aggregate_context()
- sqlite3_result()
- sqlite3_user_data()
- sqlite3_value()

All of the built-in SQL functions of SQLite are created using exactly these same interfaces. Refer to the SQLite source code, and in particular the date.c and func.c source files for examples.

Shared libraries or DLLs can be used as loadable extensions to SQLite.

8.0 Other Interfaces

This article only mentions the foundational SQLite interfaces. The SQLite library includes many other APIs implementing useful features that are not described here. A complete list of functions that form the SQLite application programming interface is found at the C/C++ Interface Specification. Refer to that document for complete and authoritative information about all SQLite interfaces.

How To Compile SQLite

SQLite is ANSI-C source code. It must be compiled into machine code before it is useful. This article is a guide to the various ways of compiling SQLite.

This article does not contain a step-by-step recipe for compiling SQLite. That would be difficult since each development situation is different. Rather, this article describes and illustrates the principals behind the compilation of SQLite. Typical compilation commands are provided as examples with the expectation that application developers can use these examples as guidance for developing their own custom compilation procedures. In other words, this article provides ideas and insights, not turnkey solutions.

Amalgamation Versus Individual Source Files

SQLite is built from over one hundred files of C code and script spread across multiple directories. The implementation of SQLite is pure ANSI-C, but many of the C-language source code files are either generated or transformed by auxiliary C programs and AWK, SED, and TCL scripts prior to being incorporated into the finished SQLite library. Building the necessary C programs and transforming and/or creating the C-language source code for SQLite is a complex process.

To simplify matters, SQLite is also available as a pre-packaged amalgamation source code file: **sqlite3.c**. The amalgamation is a single file of ANSI-C code that implements the entire SQLite library. The amalgamation is much easier to deal with. Everything is contained within a single code file, so it is easy to drop into the source tree of a larger C or C++ program. All the code generation and transformation steps have already been carried out so there are no auxiliary C programs to configure and compile and no scripts to run. And, because the entire library is contained in a single translation unit, compilers are able to do more advanced optimizations resulting in a 5% to 10% performance improvement. For these reasons, the amalgamation source file ("**sqlite3.c**") is recommended for all applications.

> *The use of the amalgamation is recommended for all applications.*

Building SQLite directly from individual source code files is certainly possible, but it is not recommended. For some specialized applications, it might be necessary to modify the build process in ways that cannot be done using just the prebuilt amalgamation source file downloaded from the website. For those situations, it is recommended that a customized amalgamation be built (as described below) and used. In other words, even if a project requires building SQLite beginning with individual source files, it is still recommended that an amalgamation source file be used as an intermediate step.

Compiling The Command-Line Interface

A build of the command-line interface requires three source files:

- **sqlite3.c**: The SQLite amalgamation source file
- **sqlite3.h**: The header files that accompanies sqlite3.c and defines the C-language interfaces to SQLite.
- **shell.c**: The command-line interface program itself. This is the C source code file that contains the definition of the **main()** routine and the loop that prompts for user input and passes that input into the SQLite database engine for processing.

All three of the above source files are contained in the amalgamation tarball available on the download page.

To build the CLI, simply put these three files in the same directory and compile them together. Using MSVC:

```
cl shell.c sqlite3.c -Fesqlite3.exe
```

On unix systems (or on Windows using cygwin or mingw+msys) the command typically looks something like this:

```
gcc shell.c sqlite3.c -lpthread -ldl
```

The pthreads library is needed to make SQLite threadsafe. But since the CLI is single threaded, we could instruct SQLite to build in a non-threadsafe mode and thereby omit the pthreads library:

```
gcc -DSQLITE_THREADSAFE=0 shell.c sqlite3.c -ldl
```

The -ldl library is needed to support dynamic loading, the sqlite3_load_extension() interface and the load_extension() SQL function. If these features are not required, then they can be omitted using SQLITE_OMIT_LOAD_EXTENSION compile-time option:

```
gcc -DSQLITE_THREADSAFE=0 -DSQLITE_OMIT_LOAD_EXTENSION shell.c sqlite3.c
```

One might want to provide other compile-time options such as -DSQLITE_ENABLE_FTS4 or -DSQLITE_ENABLE_FTS5 for full-text search, -DSQLITE_ENABLE_RTREE for the R*Tree search engine extension, -DSQLITE_ENABLE_JSON1 to include JSON SQL functions, or -DSQLITE_ENABLE_DBSTAT_VTAB for the dbstat virtual table. In order to see extra commentary in EXPLAIN listings, add the -DSQLITE_ENABLE_EXPLAIN_COMMENTS option. On unix systems, add -DHAVE_USLEEP=1 if the host machine supports the usleep()

system call. Add -DHAVE_READLINE and the -lreadline and -lncurses libraries to get command-line editing support. One might also want to specify some compiler optimization switches. (The precompiled CLI available for download from the SQLite website uses "-Os".) There are countless possible variations here. A command to compile a full-featured shell might look something like this:

```
gcc -Os -I. -DSQLITE_THREADSAFE=0 -DSQLITE_ENABLE_FTS4 \
   -DSQLITE_ENABLE_FTS5 -DSQLITE_ENABLE_JSON1 \
   -DSQLITE_ENABLE_RTREE -DSQLITE_ENABLE_EXPLAIN_COMMENTS \
   -DHAVE_USLEEP -DHAVE_READLINE \
   shell.c sqlite3.c -ldl -lreadline -lncurses -o sqlite3
```

The key point is this: Building the CLI consists of compiling together two C-language files. The **shell.c** file contains the definition of the entry point and the user input loop and the SQLite amalgamation **sqlite3.c** contains the complete implementation of the SQLite library.

Compiling The TCL Interface

The TCL interface for SQLite is a small module that is added into the regular amalgamation. The result is a new amalgamated source file called "**tclsqlite3.c**". This single source file is all that is needed to generate a shared library that can be loaded into a standard tclsh or wish using the TCL load command, or to generate a standalone tclsh that comes with SQLite built in. A copy of the tcl amalgamation is included on the download page as a file in the TEA tarball.

To generate a TCL-loadable library for SQLite on Linux, the following command will suffice:

```
gcc -o libtclsqlite3.so -shared tclsqlite3.c -lpthread -ldl -ltcl
```

Building shared libraries for Mac OS X and Windows is not nearly so simple, unfortunately. For those platforms it is best to use the configure script and makefile that is included with the TEA tarball.

To generate a standalone tclsh that is statically linked with SQLite, use this compiler invocation:

```
gcc -DTCLSH=1 tclsqlite3.c -ltcl -lpthread -ldl -lz -lm
```

The trick here is the -DTCLSH=1 option. The TCL interface module for SQLite includes a **main()** procedure that initializes a TCL interpreter and enters a command-line loop when it is compiled with -DTCLSH=1. The command above works on both Linux and Mac OS X, though one may need to adjust the library options depending on the platform and which version of TCL one is linking against.

Building The Amalgamation

The versions of the SQLite amalgamation that are supplied on the download page are normally adequate for most users. However, some projects may want or need to build their own amalgamations. A common reason for building a custom amalgamation is in order to use certain compile-time options to customize the SQLite library. Recall that the SQLite amalgamation contains a lot of C-code that is generated by auxiliary programs and scripts. Many of the compile-time options effect this generated code and must be supplied to the code generators before the amalgamation is assembled. The set of compile-time options that must be passed into the code generators can vary from one release of SQLite to the next, but at the time of this writing (circa SQLite 3.6.20, 2009-11-04) the set of options that must be known by the code generators includes:

- SQLITE_ENABLE_UPDATE_DELETE_LIMIT
- SQLITE_OMIT_ALTERTABLE
- SQLITE_OMIT_ANALYZE
- SQLITE_OMIT_ATTACH
- SQLITE_OMIT_AUTOINCREMENT
- SQLITE_OMIT_CAST
- SQLITE_OMIT_COMPOUND_SELECT
- SQLITE_OMIT_EXPLAIN
- SQLITE_OMIT_FOREIGN_KEY
- SQLITE_OMIT_PRAGMA
- SQLITE_OMIT_REINDEX
- SQLITE_OMIT_SUBQUERY
- SQLITE_OMIT_TEMPDB
- SQLITE_OMIT_TRIGGER
- SQLITE_OMIT_VACUUM
- SQLITE_OMIT_VIEW
- SQLITE_OMIT_VIRTUALTABLE

To build a custom amalgamation, first download the original individual source files onto a unix or unix-like development platform. Be sure to get the original source files not the "preprocessed source files". One can obtain the complete set of original source files either from the download page or directly from the configuration management system.

Suppose the SQLite source tree is stored in a directory named "sqlite". Plan to construct the amalgamation in a parallel directory named (for example) "bld". First construct an appropriate Makefile by either running the configure script at the top of the SQLite source tree, or by making a copy of one of the template Makefiles at the top of the source tree. Then hand edit this Makefile to include the desired compile-time options. Finally run:

```
make sqlite3.c
```

Or on Windows with MSVC:

```
nmake /f Makefile.msc sqlite3.c
```

The "sqlite3.c" make target will automatically construct the regular "**sqlite3.c**" amalgamation source file, its header file "**sqlite3.h**", and the "**tclsqlite3.c**" amalgamation source file that includes the TCL interface. Afterwards, the needed files can be copied into project directories and compiled according to the procedures outlined above.

Building A Windows DLL

To build a DLL of SQLite for use in Windows, first acquire the appropriate amalgamated source code files, sqlite3.c and sqlite3.h. These can either be downloaded from the SQLite website or custom generated from sources as shown above.

With source code files in the working directory, a DLL can be generated using MSVC with the following command:

```
cl sqlite3.c -link -dll -out:sqlite3.dll
```

The above command should be run from the MSVC Native Tools Command Prompt. If you have MSVC installed on your machine, you probably have multiple versions of this Command Prompt, for native builds for x86 and x64, and possibly also for cross-compiling to ARM. Use the appropriate Command Prompt depending on the desired DLL.

If using the MinGW compiler, the command-line is this:

```
gcc -shared sqlite3.c -o sqlite3.dll
```

Note that MinGW generates 32-bit DLLs only. There is a separate MinGW64 project that can be used to generate 64-bit DLLs. Presumably the command-line syntax is similar. Also note that recent versions of MSVC generate DLLs that will not work on WinXP and earlier versions of Windows. So for maximum compatibility of your generated DLL, MinGW is recommended. A good rule-of-thumb is to generate 32-bit DLLs using MinGW and 64-bit DLLs using MSVC.

In most cases, you will want to supplement the basic commands above with compile-time options appropriate for your application. Commonly used compile-time options include:

- **-Os** - Optimize for size. Make the DLL as small as possible.
- **-O2** - Optimize for speed. This will make the DLL larger by unrolling loops and inlining functions.
- **-DSQLITE_ENABLE_FTS4** - Include the full-text search engine code in SQLite.
- **-DSQLITE_ENABLE_RTREE** - Include the R-Tree extension.
- **-DSQLITE_ENABLE_COLUMN_METADATA** - This enables some extra APIs that are required by some common systems, including Ruby-on-Rails.

C-language Interface Specification for SQLite

These pages are intended to be precise and detailed specification. For a tutorial introduction, see instead:

- SQLite In 3 Minutes Or Less and/or
- the Introduction To The SQLite C/C++ Interface.

This same content is also available as a single large HTML file.

The SQLite interface elements can be grouped into three categories:

1. **List Of Objects.** This is a list of all abstract objects and datatypes used by the SQLite library. There are couple dozen objects in total, but the two most important objects are: A database connection object sqlite3, and the prepared statement object sqlite3_stmt.

2. **List Of Constants.** This is a list of numeric constants used by SQLite and represented by #defines in the sqlite3.h header file. These constants are things such as numeric result codes from various interfaces (ex: SQLITE_OK) or flags passed into functions to control behavior (ex: SQLITE_OPEN_READONLY).

3. **List Of Functions.** This is a list of all functions and methods operating on the objects and using and/or returning constants. There are many functions, but most applications only use a handful.

SQLite Result Codes

Many of the routines in the SQLite C-language Interface return numeric result codes indicating either success or failure, and in the event of a failure, providing some idea of the cause of the failure. This document strives to explain what each of those numeric result codes means.

Result Codes versus Error Codes

"Error codes" are a subset of "result codes" that indicate that something has gone wrong. There are only a few non-error result codes: SQLITE_OK, SQLITE_ROW, and SQLITE_DONE. The term "error code" means any result code other than these three.

Primary Result Codes versus Extended Result Codes

Result codes are signed 32-bit integers. The least significant 8 bits of the result code define a broad category and are called the "primary result code". More significant bits provide more detailed information about the error and are called the "extended result code"

Note that the primary result code is always a part of the extended result code. Given a full 32-bit extended result code, the application can always find the corresponding primary result code merely by extracting the least significant 8 bits of the extended result code.

All extended result codes are also error codes. Hence the terms "extended result code" and "extended error code" are interchangeable.

For historic compatibility, the C-language interfaces return primary result codes by default. The extended result code for the most recent error can be retrieved using the sqlite3_extended_errcode() interface. The sqlite3_extended_result_codes() interface can be used to put a database connection into a mode where it returns the extended result codes instead of the primary result codes.

Definitions

All result codes are integers. Symbolic names for all result codes are created using "#define" macros in the sqlite3.h header file. There are separate sections in the sqlite3.h header file for the result code definitions and the extended result code definitions.

Primary result code symbolic names are of the form "SQLITE_XXXXXX" where XXXXXX is a sequence of uppercase alphabetic characters. Extended result code names are of the form "SQLITE_XXXXXX_YYYYYYY" where the XXXXXX part is the corresponding primary result code and the YYYYYYY is an extension that further classifies the result code.

The names and numeric values for existing result codes are fixed and unchanging. However, new result codes, and especially new extended result codes, might appear in future releases of SQLite.

Primary Result Code List

The 31 result codes are defined in sqlite3.h and are listed in alphabetical order below:

- SQLITE_ABORT (4)
- SQLITE_AUTH (23)
- SQLITE_BUSY (5)
- SQLITE_CANTOPEN (14)
- SQLITE_CONSTRAINT (19)
- SQLITE_CORRUPT (11)
- SQLITE_DONE (101)
- SQLITE_EMPTY (16)
- SQLITE_ERROR (1)
- SQLITE_FORMAT (24)
- SQLITE_FULL (13)

- SQLITE_INTERNAL (2)
- SQLITE_INTERRUPT (9)
- SQLITE_IOERR (10)
- SQLITE_LOCKED (6)
- SQLITE_MISMATCH (20)
- SQLITE_MISUSE (21)
- SQLITE_NOLFS (22)
- SQLITE_NOMEM (7)
- SQLITE_NOTADB (26)
- SQLITE_NOTFOUND (12)
- SQLITE_NOTICE (27)

- SQLITE_OK (0)
- SQLITE_PERM (3)
- SQLITE_PROTOCOL (15)
- SQLITE_RANGE (25)
- SQLITE_READONLY (8)

- SQLITE_ROW (100)
- SQLITE_SCHEMA (17)
- SQLITE_TOOBIG (18)
- SQLITE_WARNING (28)

Extended Result Code List

The 52 extended result codes are defined in sqlite3.h and are listed in alphabetical order below:

- SQLITE_ABORT_ROLLBACK (516)
- SQLITE_BUSY_RECOVERY (261)
- SQLITE_BUSY_SNAPSHOT (517)
- SQLITE_CANTOPEN_CONVPATH (1038)
- SQLITE_CANTOPEN_FULLPATH (782)
- SQLITE_CANTOPEN_ISDIR (526)
- SQLITE_CANTOPEN_NOTEMPDIR (270)
- SQLITE_CONSTRAINT_CHECK (275)
- SQLITE_CONSTRAINT_COMMITHOOK (531)
- SQLITE_CONSTRAINT_FOREIGNKEY (787)
- SQLITE_CONSTRAINT_FUNCTION (1043)
- SQLITE_CONSTRAINT_NOTNULL (1299)
- SQLITE_CONSTRAINT_PRIMARYKEY (1555)
- SQLITE_CONSTRAINT_ROWID (2579)
- SQLITE_CONSTRAINT_TRIGGER (1811)
- SQLITE_CONSTRAINT_UNIQUE (2067)
- SQLITE_CONSTRAINT_VTAB (2323)
- SQLITE_CORRUPT_VTAB (267)
- SQLITE_IOERR_ACCESS (3338)
- SQLITE_IOERR_BLOCKED (2826)
- SQLITE_IOERR_CHECKRESERVEDLOCK (3594)
- SQLITE_IOERR_CLOSE (4106)
- SQLITE_IOERR_CONVPATH (6666)
- SQLITE_IOERR_DELETE (2570)
- SQLITE_IOERR_DELETE_NOENT (5898)
- SQLITE_IOERR_DIR_CLOSE (4362)
- SQLITE_IOERR_DIR_FSYNC (1290)
- SQLITE_IOERR_FSTAT (1802)
- SQLITE_IOERR_FSYNC (1034)
- SQLITE_IOERR_GETTEMPPATH (6410)

- SQLITE_IOERR_LOCK (3850)
- SQLITE_IOERR_MMAP (6154)
- SQLITE_IOERR_NOMEM (3082)
- SQLITE_IOERR_RDLOCK (2314)
- SQLITE_IOERR_READ (266)
- SQLITE_IOERR_SEEK (5642)
- SQLITE_IOERR_SHMLOCK (5130)
- SQLITE_IOERR_SHMMAP (5386)
- SQLITE_IOERR_SHMOPEN (4618)
- SQLITE_IOERR_SHMSIZE (4874)
- SQLITE_IOERR_SHORT_READ (522)
- SQLITE_IOERR_TRUNCATE (1546)
- SQLITE_IOERR_UNLOCK (2058)
- SQLITE_IOERR_WRITE (778)
- SQLITE_LOCKED_SHAREDCACHE (262)
- SQLITE_NOTICE_RECOVER_ROLLBACK (539)
- SQLITE_NOTICE_RECOVER_WAL (283)
- SQLITE_READONLY_CANTLOCK (520)
- SQLITE_READONLY_DBMOVED (1032)
- SQLITE_READONLY_RECOVERY (264)
- SQLITE_READONLY_ROLLBACK (776)
- SQLITE_WARNING_AUTOINDEX (284)

Result Code Meanings

The meanings for all 83 result code values are shown below, in numeric order.

(0) SQLITE_OK

The SQLITE_OK result code means that the operation was successful and that there were no errors. Most other result codes indicate an error.

(1) SQLITE_ERROR

The SQLITE_ERROR result code is a generic error code that is used when no other more specific error code is available.

(2) SQLITE_INTERNAL

The SQLITE_INTERNAL result code indicates an internal malfunction. In a working version of SQLite, an application should never see this result code. If application does encounter this result code, it shows that there is a bug in the database engine.

SQLite does not currently generate this result code. However, application-defined SQL functions or virtual tables, or VFSes, or other extensions might cause this result code to be returned.

(3) SQLITE_PERM

The SQLITE_PERM result code indicates that the requested access mode for a newly created database could not be provided.

(4) SQLITE_ABORT

The SQLITE_ABORT result code indicates that an operation was aborted prior to completion, usually be application request. See also: SQLITE_INTERRUPT.

If the callback function to sqlite3_exec() returns non-zero, then sqlite3_exec() will return SQLITE_ABORT.

If a ROLLBACK operation occurs on the same database connection as a pending read or write, then the pending read or write may fail with an SQLITE_ABORT or SQLITE_ABORT_ROLLBACK error.

In addition to being a result code, the SQLITE_ABORT value is also used as a conflict resolution mode returned from the sqlite3_vtab_on_conflict() interface.

(5) SQLITE_BUSY

The SQLITE_BUSY result code indicates that the database file could not be written (or in some cases read) because of concurrent activity by some other database connection, usually a database connection in a separate process.

For example, if process A is in the middle of a large write transaction and at the same time process B attempts to start a new write transaction, process B will get back an SQLITE_BUSY result because SQLite only supports one writer at a time. Process B will need to wait for process A to finish its transaction before starting a new transaction. The sqlite3_busy_timeout() and sqlite3_busy_handler() interfaces and the busy_timeout pragma are available to process B to help it deal with SQLITE_BUSY errors.

An SQLITE_BUSY error can occur at any point in a transaction: when the transaction is first started, during any write or update operations, or when the transaction commits. To avoid encountering SQLITE_BUSY errors in the middle of a transaction, the application can use BEGIN IMMEDIATE instead of just BEGIN to start a transaction. The BEGIN IMMEDIATE command might itself return SQLITE_BUSY, but if it succeeds, then SQLite guarantees that no subsequent operations on the same database through the next COMMIT will return SQLITE_BUSY.

See also: SQLITE_BUSY_RECOVERY and SQLITE_BUSY_SNAPSHOT.

The SQLITE_BUSY result code differs from SQLITE_LOCKED in that SQLITE_BUSY indicates a conflict with a separate database connection, probably in a separate process, whereas SQLITE_LOCKED indicates a conflict within the same database connection (or sometimes a database connection with a shared cache).

(6) SQLITE_LOCKED

The SQLITE_LOCKED result code indicates that a write operation could not continue because of a conflict within the same database connection or a conflict with a different database connection that uses a shared cache.

For example, a DROP TABLE statement cannot be run while another thread is reading from that table on the same database connection because dropping the table would delete the table out from under the concurrent reader.

The SQLITE_LOCKED result code differs from SQLITE_BUSY in that SQLITE_LOCKED indicates a conflict on the same database connection (or on a connection with a shared cache) whereas SQLITE_BUSY indicates a conflict with a different database connection, probably in a different process.

(7) SQLITE_NOMEM

The SQLITE_NOMEM result code indicates that SQLite was unable to allocate all the memory it needed to complete the operation. In other words, an internal call to sqlite3_malloc() or sqlite3_realloc() has failed in a case where the memory being allocated was required in order to continue the operation.

(8) SQLITE_READONLY

The SQLITE_READONLY result code is returned when an attempt is made to alter some data for which the current database connection does not have write permission.

(9) SQLITE_INTERRUPT

The SQLITE_INTERRUPT result code indicates that an operation was interrupted by the sqlite3_interrupt() interface. See also: SQLITE_ABORT

(10) SQLITE_IOERR

The SQLITE_IOERR result code says that the operation could not finish because the operating system reported an I/O error.

A full disk drive will normally give an SQLITE_FULL error rather than an SQLITE_IOERR error.

There are many different extended result codes for I/O errors that identify the specific I/O operation that failed.

(11) SQLITE_CORRUPT

The SQLITE_CORRUPT result code indicates that the database file has been corrupted. See the How To Corrupt Your Database Files for further discussion on how corruption can occur.

(12) SQLITE_NOTFOUND

The SQLITE_NOTFOUND result code is used in two contexts. SQLITE_NOTFOUND can be returned by the sqlite3_file_control() interface to indicate that the file control opcode passed as the third argument was not recognized by the underlying VFS. SQLITE_NOTFOUND can also be returned by the xSetSystemCall() method of an sqlite3_vfs object.

The SQLITE_NOTFOUND result code is also used internally by the SQLite implementation, but those internal uses are not exposed to the application.

(13) SQLITE_FULL

The SQLITE_FULL result code indicates that a write could not complete because the disk is full. Note that this error can occur when trying to write information into the main database file, or it can also occur when writing into temporary disk files.

Sometimes applications encounter this error even though there is an abundance of primary disk space because the error occurs when writing into temporary disk files on a system where temporary files are stored on a separate partition with much less space that the primary disk.

(14) SQLITE_CANTOPEN

The SQLITE_CANTOPEN result code indicates that SQLite was unable to open a file. The file in question might be a primary database file or on of several temporary disk files.

(15) SQLITE_PROTOCOL

The SQLITE_PROTOCOL result code indicates a problem with the file locking protocol used by SQLite. The SQLITE_PROTOCOL error is currently only returned when using WAL mode and attempting to start a new transaction. There is a race condition that can occur when two separate database connections both try to start a transaction at the same time in WAL mode. The loser of the race backs off and tries again, after a brief delay. If the same connection loses the locking race dozens of times over a span of multiple seconds, it will eventually give up and return SQLITE_PROTOCOL. The SQLITE_PROTOCOL error should appear in practice very, very rarely, and only when there are many separate processes all competing intensely to write to the same database.

(16) SQLITE_EMPTY

The SQLITE_EMPTY result code is not currently used.

(17) SQLITE_SCHEMA

The SQLITE_SCHEMA result code indicates that the database schema has changed. This result code can be returned from sqlite3_step() for a prepared statement that was generated using sqlite3_prepare() or sqlite3_prepare16(). If the database schema was changed by some other process in between the time that the statement was prepared and the time the statement was run, this error can result.

If a prepared statement is generated from sqlite3_prepare_v2() then the statement is automatically re-prepared if the schema changes, up to SQLITE_MAX_SCHEMA_RETRY times (default: 50). The sqlite3_step() interface will only return SQLITE_SCHEMA back to the application if the failure persists after these many retries.

(18) SQLITE_TOOBIG

The SQLITE_TOOBIG error code indicates that a string or BLOB was too large. The default maximum length of a string or BLOB in SQLite is 1,000,000,000 bytes. This maximum length can be changed at compile-time using the SQLITE_MAX_LENGTH compile-time option, or at run-time using the sqlite3_limit(db,SQLITE_LIMIT_LENGTH,...) interface. The SQLITE_TOOBIG error results when SQLite encounters a string or BLOB that exceeds the compile-time or run-time limit.

The SQLITE_TOOBIG error code can also result when an oversized SQL statement is passed into one of the sqlite3_prepare_v2() interfaces. The maximum length of an SQL statement defaults to a much smaller value of 1,000,000 bytes. The maximum SQL statement length can be set at compile-time using SQLITE_MAX_SQL_LENGTH or at run-time using sqlite3_limit(db,SQLITE_LIMIT_SQL_LENGTH,...).

(19) SQLITE_CONSTRAINT

The SQLITE_CONSTRAINT error code means that an SQL constraint violation occurred while trying to process an SQL statement. Additional information about the failed constraint can be found by consulting the accompanying error message (returned via sqlite3_errmsg() or sqlite3_errmsg16()) or by looking at the extended error code.

(20) SQLITE_MISMATCH

The SQLITE_MISMATCH error code indicates a datatype mismatch.

SQLite is normally very forgiving about mismatches between the type of a value and the declared type of the container in which that value is to be stored. For example, SQLite allows the application to store a large BLOB in a column with a declared type of BOOLEAN. But in a few cases, SQLite is strict about types. The SQLITE_MISMATCH error is returned in those few cases when the types do not match.

The rowid of a table must be an integer. Attempt to set the rowid to anything other than an integer (or a NULL which will be automatically converted into the next available integer rowid) results in an SQLITE_MISMATCH error.

(21) SQLITE_MISUSE

The SQLITE_MISUSE return code might be returned if the application uses any SQLite interface in a way that is undefined or unsupported. For example, using a prepared statement after that prepared statement has been finalized might result in an SQLITE_MISUSE error.

SQLite tries to detect misuse and report the misuse using this result code. However, there is no guarantee that the detection of misuse will be successful. Misuse detection is probabilistic. Applications should never depend on an SQLITE_MISUSE return value.

If SQLite ever returns SQLITE_MISUSE from any interface, that means that the application is incorrectly coded and needs to be fixed. Do not ship an application that sometimes returns SQLITE_MISUSE from a standard SQLite interface because that application contains potentially serious bugs.

(22) SQLITE_NOLFS

The SQLITE_NOLFS error can be returned on systems that do not support large files when the database grows to be larger than what the filesystem can handle. "NOLFS" stands for "NO Large File Support".

(23) SQLITE_AUTH

The SQLITE_AUTH error is returned when the authorizer callback indicates that an SQL statement being prepared is not authorized.

(24) SQLITE_FORMAT

The SQLITE_FORMAT error code is not currently used by SQLite.

(25) SQLITE_RANGE

The SQLITE_RANGE error indices that the parameter number argument to one of the sqlite3_bind routines or the column number in one of the sqlite3_column routines is out of range.

(26) SQLITE_NOTADB

When attempting to open a file, the SQLITE_NOTADB error indicates that the file being opened does not appear to be an SQLite database file.

(27) SQLITE_NOTICE

The SQLITE_NOTICE result code is not returned by any C/C++ interface. However, SQLITE_NOTICE (or rather one of its extended error codes) is sometimes used as the first argument in an sqlite3_log() callback to indicate that an unusual operation is taking place.

(28) SQLITE_WARNING

The SQLITE_WARNING result code is not returned by any C/C++ interface. However, SQLITE_WARNING (or rather one of its extended error codes) is sometimes used as the first argument in an sqlite3_log() callback to indicate that an unusual and possibly ill-advised operation is taking place.

(100) SQLITE_ROW

The SQLITE_ROW result code returned by sqlite3_step() indicates that another row of output is available.

(101) SQLITE_DONE

The SQLITE_DONE result code indicates that an operation has completed. The SQLITE_DONE result code is most commonly seen as a return value from sqlite3_step() indicating that the SQL statement has run to completion. But SQLITE_DONE can also be returned by other multi-step interfaces such as sqlite3_backup_step().

(261) SQLITE_BUSY_RECOVERY

The SQLITE_BUSY_RECOVERY error code is an extended error code for SQLITE_BUSY that indicates that an operation could not continue because another process is busy recovering a WAL mode database file following a crash. The SQLITE_BUSY_RECOVERY error code only occurs on WAL mode databases.

(262) SQLITE_LOCKED_SHAREDCACHE

The SQLITE_LOCKED_SHAREDCACHE error code is an extended error code for SQLITE_LOCKED indicating that the locking conflict has occurred due to contention with a different database connection that happens to hold a shared cache with the database connection to which the error was returned. For example, if the other database connection is holding an exclusive lock on the database, then the database connection that receives this error will be unable to read or write any part of the database file unless it has the read_uncommitted pragma enabled.

The SQLITE_LOCKED_SHARECACHE error code works very much like the SQLITE_BUSY error code except that SQLITE_LOCKED_SHARECACHE is for separate database connections that share a cache whereas SQLITE_BUSY is for the much more common case of separate database connections that do not share the same cache. Also, the sqlite3_busy_handler() and sqlite3_busy_timeout() interfaces do not help in resolving SQLITE_LOCKED_SHAREDCACHE conflicts.

(264) SQLITE_READONLY_RECOVERY

The SQLITE_READONLY_RECOVERY error code is an extended error code for SQLITE_READONLY. The SQLITE_READONLY_RECOVERY error code indicates that a WAL mode database cannot be opened because the database file needs to be recovered and recovery requires write access but only read access is available.

(266) SQLITE_IOERR_READ

The SQLITE_IOERR_READ error code is an extended error code for SQLITE_IOERR indicating an I/O error in the VFS layer while trying to read from a file on disk. This error might result from a hardware malfunction or because a filesystem came unmounted while the file was open.

(267) SQLITE_CORRUPT_VTAB

The SQLITE_CORRUPT_VTAB error code is an extended error code for SQLITE_CORRUPT used by virtual tables. A virtual table might return SQLITE_CORRUPT_VTAB to indicate that content in the virtual table is corrupt.

(270) SQLITE_CANTOPEN_NOTEMPDIR

The SQLITE_CANTOPEN_NOTEMPDIR error code is no longer used.

(275) SQLITE_CONSTRAINT_CHECK

The SQLITE_CONSTRAINT_CHECK error code is an extended error code for SQLITE_CONSTRAINT indicating that a CHECK constraint failed.

(283) SQLITE_NOTICE_RECOVER_WAL

The SQLITE_NOTICE_RECOVER_WAL result code is passed to the callback of sqlite3_log() when a WAL mode database file is recovered.

(284) SQLITE_WARNING_AUTOINDEX

The SQLITE_WARNING_AUTOINDEX result code is passed to the callback of sqlite3_log() whenever automatic indexing is used. This can serve as a warning to application designers that the database might benefit from additional indexes.

(516) SQLITE_ABORT_ROLLBACK

The SQLITE_ABORT_ROLLBACK error code is an extended error code for SQLITE_ABORT indicating that an SQL statement aborted because the transaction that was active when the SQL statement first started was rolled back. Pending write operations always fail with this error when a rollback occurs. A ROLLBACK will cause a pending read operation to fail only if the schema was changed within the transaction being rolled back.

(517) SQLITE_BUSY_SNAPSHOT

The SQLITE_BUSY_SNAPSHOT error code is an extended error code for SQLITE_BUSY that occurs on WAL mode databases when a database connection tries to promote a read transaction into a write transaction but finds that another database connection has already written to the database and thus invalidated prior reads.

The following scenario illustrates how an SQLITE_BUSY_SNAPSHOT error might arise:

1. Process A starts a read transaction on the database and does one or more SELECT statement. Process A keeps the transaction open.
2. Process B updates the database, changing values previous read by process A.
3. Process A now tries to write to the database. But process A's view of the database content is now obsolete because process B has modified the database file after process A read from it. Hence process A gets an SQLITE_BUSY_SNAPSHOT error.

(520) SQLITE_READONLY_CANTLOCK

The SQLITE_READONLY_CANTLOCK error code is an extended error code for SQLITE_READONLY. The SQLITE_READONLY_CANTLOCK error code indicates that SQLite is unable to obtain a read lock on a WAL mode database because the shared-memory file associated with that database is read-only.

(522) SQLITE_IOERR_SHORT_READ

The SQLITE_IOERR_SHORT_READ error code is an extended error code for SQLITE_IOERR indicating that a read attempt in the VFS layer was unable to obtain as many bytes as was requested. This might be due to a truncated file.

(526) SQLITE_CANTOPEN_ISDIR

The SQLITE_CANTOPEN_ISDIR error code is an extended error code for SQLITE_CANTOPEN indicating that a file open operation failed because the file is really a directory.

(531) SQLITE_CONSTRAINT_COMMITHOOK

The SQLITE_CONSTRAINT_COMMITHOOK error code is an extended error code for SQLITE_CONSTRAINT indicating that a commit hook callback returned non-zero that thus caused the SQL statement to be rolled back.

(539) SQLITE_NOTICE_RECOVER_ROLLBACK

The SQLITE_NOTICE_RECOVER_ROLLBACK result code is passed to the callback of sqlite3_log() when a hot journal is rolled back.

(776) SQLITE_READONLY_ROLLBACK

The SQLITE_READONLY_ROLLBACK error code is an extended error code for SQLITE_READONLY. The SQLITE_READONLY_ROLLBACK error code indicates that a database cannot be opened because it has a hot journal that needs to be rolled back but cannot because the database is readonly.

(778) SQLITE_IOERR_WRITE

The SQLITE_IOERR_WRITE error code is an extended error code for SQLITE_IOERR indicating an I/O error in the VFS layer while trying to write into a file on disk. This error might result from a hardware malfunction or because a filesystem came unmounted while the file was open. This error should not occur if the filesystem is full as there is a separate error code (SQLITE_FULL) for that purpose.

(782) SQLITE_CANTOPEN_FULLPATH

The SQLITE_CANTOPEN_FULLPATH error code is an extended error code for SQLITE_CANTOPEN indicating that a file open operation failed because the operating system was unable to convert the filename into a full pathname.

(787) SQLITE_CONSTRAINT_FOREIGNKEY

The SQLITE_CONSTRAINT_FOREIGNKEY error code is an extended error code for SQLITE_CONSTRAINT indicating that a foreign key constraint failed.

(1032) SQLITE_READONLY_DBMOVED

The SQLITE_READONLY_DBMOVED error code is an extended error code for SQLITE_READONLY. The SQLITE_READONLY_DBMOVED error code indicates that a database cannot be modified because the database file has been moved since it was opened, and so any attempt to modify the database might result in database corruption if the processes crashes because the rollback journal would not be correctly named.

(1034) SQLITE_IOERR_FSYNC

The SQLITE_IOERR_FSYNC error code is an extended error code for SQLITE_IOERR indicating an I/O error in the VFS layer while trying to flush previously written content out of OS and/or disk-control buffers and into persistent storage. In other words, this code indicates a problem with the fsync() system call in unix or the FlushFileBuffers() system call in windows.

(1038) SQLITE_CANTOPEN_CONVPATH

The SQLITE_CANTOPEN_CONVPATH error code is an extended error code for SQLITE_CANTOPEN used only by Cygwin VFS and indicating that the cygwin_conv_path() system call failed while trying to open a file. See also: SQLITE_IOERR_CONVPATH

(1043) SQLITE_CONSTRAINT_FUNCTION

The SQLITE_CONSTRAINT_FUNCTION error code is not currently used by the SQLite core. However, this error code is available for use by extension functions.

(1290) SQLITE_IOERR_DIR_FSYNC

The SQLITE_IOERR_DIR_FSYNC error code is an extended error code for SQLITE_IOERR indicating an I/O error in the VFS layer while trying to invoke fsync() on a directory. The unix VFS attempts to fsync() directories after creating or deleting certain files to ensure that those files will still appear in the filesystem following a power loss or system crash. This error code indicates a problem attempting to perform that fsync().

(1299) SQLITE_CONSTRAINT_NOTNULL

The SQLITE_CONSTRAINT_NOTNULL error code is an extended error code for SQLITE_CONSTRAINT indicating that a NOT NULL constraint failed.

(1546) SQLITE_IOERR_TRUNCATE

The SQLITE_IOERR_TRUNCATE error code is an extended error code for SQLITE_IOERR indicating an I/O error in the VFS layer while trying to truncate a file to a smaller size.

(1555) SQLITE_CONSTRAINT_PRIMARYKEY

The SQLITE_CONSTRAINT_PRIMARYKEY error code is an extended error code for SQLITE_CONSTRAINT indicating that a PRIMARY KEY constraint failed.

(1802) SQLITE_IOERR_FSTAT

The SQLITE_IOERR_FSTAT error code is an extended error code for SQLITE_IOERR indicating an I/O error in the VFS layer while trying to invoke fstat() (or the equivalent) on a file in order to determine information such as the file size or access permissions.

(1811) SQLITE_CONSTRAINT_TRIGGER

The SQLITE_CONSTRAINT_TRIGGER error code is an extended error code for SQLITE_CONSTRAINT indicating that a RAISE function within a trigger fired, causing the SQL statement to abort.

(2058) SQLITE_IOERR_UNLOCK

The SQLITE_IOERR_UNLOCK error code is an extended error code for SQLITE_IOERR indicating an I/O error within xUnlock method on the sqlite3_io_methods object.

(2067) SQLITE_CONSTRAINT_UNIQUE

The SQLITE_CONSTRAINT_UNIQUE error code is an extended error code for SQLITE_CONSTRAINT indicating that a UNIQUE constraint failed.

(2314) SQLITE_IOERR_RDLOCK

The SQLITE_IOERR_UNLOCK error code is an extended error code for SQLITE_IOERR indicating an I/O error within xLock method on the sqlite3_io_methods object while trying to obtain a read lock.

(2323) SQLITE_CONSTRAINT_VTAB

The SQLITE_CONSTRAINT_VTAB error code is not currently used by the SQLite core. However, this error code is available for use by application-defined virtual tables.

(2570) SQLITE_IOERR_DELETE

The SQLITE_IOERR_UNLOCK error code is an extended error code for SQLITE_IOERR indicating an I/O error within xDelete method on the sqlite3_vfs object.

(2579) SQLITE_CONSTRAINT_ROWID

The SQLITE_CONSTRAINT_ROWID error code is an extended error code for SQLITE_CONSTRAINT indicating that a rowid is not unique.

(2826) SQLITE_IOERR_BLOCKED

The SQLITE_IOERR_BLOCKED error code is no longer used.

(3082) SQLITE_IOERR_NOMEM

The SQLITE_IOERR_NOMEM error code is sometimes returned by the VFS layer to indicate that an operation could not be completed due to the inability to allocate sufficient memory. This error code is normally converted into SQLITE_NOMEM by the higher layers of SQLite before being returned to the application.

(3338) SQLITE_IOERR_ACCESS

The SQLITE_IOERR_ACCESS error code is an extended error code for SQLITE_IOERR indicating an I/O error within the xAccess method on the sqlite3_vfs object.

(3594) SQLITE_IOERR_CHECKRESERVEDLOCK

The SQLITE_IOERR_CHECKRESERVEDLOCK error code is an extended error code for SQLITE_IOERR indicating an I/O error within the xCheckReservedLock method on the sqlite3_io_methods object.

(3850) SQLITE_IOERR_LOCK

The SQLITE_IOERR_LOCK error code is an extended error code for SQLITE_IOERR indicating an I/O error in the advisory file locking logic. Usually an SQLITE_IOERR_LOCK error indicates a problem obtaining a PENDING lock. However it can also indicate miscellaneous locking errors on some of the specialized VFSes used on Macs.

(4106) SQLITE_IOERR_CLOSE

The SQLITE_IOERR_ACCESS error code is an extended error code for SQLITE_IOERR indicating an I/O error within the xClose method on the sqlite3_io_methods object.

(4362) SQLITE_IOERR_DIR_CLOSE

The SQLITE_IOERR_DIR_CLOSE error code is no longer used.

(4618) SQLITE_IOERR_SHMOPEN

The SQLITE_IOERR_SHMOPEN error code is an extended error code for SQLITE_IOERR indicating an I/O error within the xShmMap method on the sqlite3_io_methods object while trying to open a new shared memory segment.

(4874) SQLITE_IOERR_SHMSIZE

The SQLITE_IOERR_SHMSIZE error code is an extended error code for SQLITE_IOERR indicating an I/O error within the xShmMap method on the sqlite3_io_methods object while trying to resize an existing shared memory segment.

(5130) SQLITE_IOERR_SHMLOCK

The SQLITE_IOERR_SHMLOCK error code is no longer used.

(5386) SQLITE_IOERR_SHMMAP

The SQLITE_IOERR_SHMMAP error code is an extended error code for SQLITE_IOERR indicating an I/O error within the xShmMap method on the sqlite3_io_methods object while trying to map a shared memory segment into the process address space.

(5642) SQLITE_IOERR_SEEK

The SQLITE_IOERR_SEEK error code is an extended error code for SQLITE_IOERR indicating an I/O error within the xRead or xWrite methods on the sqlite3_io_methods object while trying to seek a file descriptor to the beginning point of the file where the read or write is to occur.

(5898) SQLITE_IOERR_DELETE_NOENT

The SQLITE_IOERR_DELETE_NOENT error code is an extended error code for SQLITE_IOERR indicating that the xDelete method on the sqlite3_vfs object failed because the file being deleted does not exist.

(6154) SQLITE_IOERR_MMAP

The SQLITE_IOERR_MMAP error code is an extended error code for SQLITE_IOERR indicating an I/O error within the xFetch or xUnfetch methods on the sqlite3_io_methods object while trying to map or unmap part of the database file into the process address space.

(6410) SQLITE_IOERR_GETTEMPPATH

The SQLITE_IOERR_GETTEMPPATH error code is an extended error code for SQLITE_IOERR indicating that the VFS is unable to determine a suitable directory in which to place temporary files.

(6666) SQLITE_IOERR_CONVPATH

The SQLITE_IOERR_CONVPATH error code is an extended error code for SQLITE_IOERR used only by Cygwin VFS and indicating that the cygwin_conv_path() system call failed. See also: SQLITE_CANTOPEN_CONVPATH

The Tcl interface to the SQLite library

The SQLite library is designed to be very easy to use from a Tcl or Tcl/Tk script. SQLite began as a Tcl extension and the primary test suite for SQLite is written in TCL. SQLite can be used with any programming language, but its connections to TCL run deep.

This document gives an overview of the Tcl programming interface for SQLite.

The API

The interface to the SQLite library consists of single tcl command named **sqlite3** Because there is only this one command, the interface is not placed in a separate namespace.

The **sqlite3** command is used as follows:

> **sqlite3** *dbcmd database-name ?options?*

The **sqlite3** command opens the database named in the second argument. If the database does not already exist, the default behavior is for it to be created automatically (though this can be changed by using the "**-create false**" option). The **sqlite3** command always creates a new Tcl command to control the database. The name of the new Tcl command is given by the first argument. This approach is similar to the way widgets are created in Tk.

The name of the database is usually just the name of a disk file in which the database is stored. If the name of the database is the special name ":memory:" then a new database is created in memory. If the name of the database is an empty string, then the database is created in an empty file that is automatically deleted when the database connection closes. URI filenames can be used if the "**-uri yes**" option is supplied on the **sqlite3** command.

Options understood by the **sqlite3** command include:

> **-create** *BOOLEAN*
>
> If true, then a new database is created if one does not already exist. If false, then an attempt to open a database file that does not previously exist raises an error. The default behavior is "true".
>
> **-nomutex** *BOOLEAN*
>
> If true, then all mutexes for the database connection are disabled. This provides a small performance boost in single-threaded applications.
>
> **-readonly** *BOOLEAN*
>
> If true, then open the database file read-only. If false, then the database is opened for both reading and writing if filesystem permissions allow, or for reading only if filesystem write permission is denied by the operating system. The default setting is "false". Note that if the previous process to have the database did not exit cleanly and left behind a hot journal, then the write permission is required to recover the database after opening, and the database cannot be opened read-only.
>
> **-uri** *BOOLEAN*
>
> If true, then interpret the filename argument as a URI filename. If false, then the argument is a literal filename. The default value is "false".
>
> **-vfs** *VFSNAME*
>
> Use an alternative VFS named by the argument.

Once an SQLite database is open, it can be controlled using methods of the *dbcmd*. There are currently 33 methods defined.

- authorizer
- backup
- busy
- cache
- changes
- close
- collate
- collation_needed
- commit_hook
- complete
- copy

- enable_load_extension
- errorcode

- eval
- exists
- function
- incrblob
- last_insert_rowid
- nullvalue
- onecolumn
- profile
- progress

- restore
- rollback_hook
- status
- timeout
- total_changes
- trace
- transaction
- unlock_notify
- update_hook
- version
- wal_hook

The use of each of these methods will be explained in the sequel, though not in the order shown above.

The "eval" method

The most useful *dbcmd* method is "eval". The eval method is used to execute SQL on the database. The syntax of the eval method looks like this:

> *dbcmd* **eval** *sql* ?*array-name*? ?*script*?

The job of the eval method is to execute the SQL statement or statements given in the second argument. For example, to create a new table in a database, you can do this:

> **sqlite3 db1 ./testdb db1 eval {CREATE TABLE t1(a int, b text)}**

The above code creates a new table named **t1** with columns **a** and **b**. What could be simpler?

Query results are returned as a list of column values. If a query requests 2 columns and there are 3 rows matching the query, then the returned list will contain 6 elements. For example:

```
db1 eval {INSERT INTO t1 VALUES(1,'hello')} db1 eval {INSERT INTO t1 VALUES(2,'goodbye')} db1 eval {INSERT INTO t1 VALUES(3,'howdy!')} set x [db1 eval {SELECT * FROM t1 ORDER BY a}]
```

The variable **$x** is set by the above code to

```
1 hello 2 goodbye 3 howdy!
```

You can also process the results of a query one row at a time by specifying the name of an array variable and a script following the SQL code. For each row of the query result, the values of all columns will be inserted into the array variable and the script will be executed. For instance:

```
db1 eval {SELECT * FROM t1 ORDER BY a} values { parray values puts "" }
```

This last code will give the following output:

```
*values() = a b values(a) = 1 values(b) = hello

values(*) = a b values(a) = 2 values(b) = goodbye

values(*) = a b values(a) = 3 values(b) = howdy!

**
```

For each column in a row of the result, the name of that column is used as an index in to array and the value of the column is stored in the corresponding array entry. (Caution: If two or more columns in the result set of a query have the same name, then the last column with that name will overwrite prior values and earlier columns with the same name will be inaccessible.) The special array index * is used to store a list of column names in the order that they appear.

If the array variable name is omitted or is the empty string, then the value of each column is stored in a variable with the same name as the column itself. For example:

```
db1 eval {SELECT * FROM t1 ORDER BY a} { puts "a=$a b=$b" }
```

From this we get the following output

```
a=1 b=hello a=2 b=goodbye a=3 b=howdy!
```

Tcl variable names can appear in the SQL statement of the second argument in any position where it is legal to put a string or number literal. The value of the variable is substituted for the variable name. If the variable does not exist a NULL values is used. For example:

```
db1 eval {INSERT INTO t1 VALUES(5,$bigstring)}
```

Note that it is not necessary to quote the $bigstring value. That happens automatically. If $bigstring is a large string or binary object, this technique is not only easier to write, it is also much more efficient since it avoids making a copy of the content of $bigstring.

If the $bigstring variable has both a string and a "bytearray" representation, then TCL inserts the value as a string. If it has only a "bytearray" representation, then the value is inserted as a BLOB. To force a value to be inserted as a BLOB even if it also has a text representation, use a "@" character to in place of the "$". Like this:

```
db1 eval {INSERT INTO t1 VALUES(5,@bigstring)}
```

If the variable does not have a bytearray representation, then "@" works just like "$". Note that ":" works like "$" in all cases so the following is another way to express the same statement:

```
db1 eval {INSERT INTO t1 VALUES(5,:bigstring)}
```

The use of ":" instead of "$" before the name of a variable can sometimes be useful if the SQL text is enclosed in double-quotes "..." instead of curly-braces {...}. When the SQL is contained within double-quotes "..." then TCL will do the substitution of $-variables, which can lead to SQL injection if extreme care is not used. But TCL will never substitute a :-variable regardless of whether double-quotes "..." or curly-braces {...} are used to enclose the SQL, so the use of :-variables adds an extra measure of defense against SQL injection.

The "close" method

As its name suggests, the "close" method to an SQLite database just closes the database. This has the side-effect of deleting the *dbcmd* Tcl command. Here is an example of opening and then immediately closing a database:

```
sqlite3 db1 ./testdb db1 close
```

If you delete the *dbcmd* directly, that has the same effect as invoking the "close" method. So the following code is equivalent to the previous:

```
sqlite3 db1 ./testdb rename db1 {}
```

The "transaction" method

The "transaction" method is used to execute a TCL script inside an SQLite database transaction. The transaction is committed when the script completes, or it rolls back if the script fails. If the transaction occurs within another transaction (even one that is started manually using BEGIN) it is a no-op.

The transaction command can be used to group together several SQLite commands in a safe way. You can always start transactions manually using BEGIN, of course. But if an error occurs so that the COMMIT or ROLLBACK are never run, then the database will remain locked indefinitely. Also, BEGIN does not nest, so you have to make sure no other transactions are active before starting a new one. The "transaction" method takes care of all of these details automatically.

The syntax looks like this:

> *dbcmd* **transaction** *?transaction-type? script*

The *transaction-type* can be one of **deferred**, **exclusive** or **immediate**. The default is deferred.

The "cache" method

The "eval" method described above keeps a cache of prepared statements for recently evaluated SQL commands. The "cache" method is used to control this cache. The first form of this command is:

> *dbcmd* **cache size** *N*

This sets the maximum number of statements that can be cached. The upper limit is 100. The default is 10. If you set the cache size to 0, no caching is done.

The second form of the command is this:

> *dbcmd* **cache flush**

The cache-flush method finalizes all prepared statements currently in the cache.

The "complete" method

The "complete" method takes a string of supposed SQL as its only argument. It returns TRUE if the string is a complete statement of SQL and FALSE if there is more to be entered.

The "complete" method is useful when building interactive applications in order to know when the user has finished entering a line of SQL code. This is really just an interface to the **sqlite3_complete()** C function.

The "copy" method

The "copy" method copies data from a file into a table. It returns the number of rows processed successfully from the file. The syntax of the copy method looks like this:

> *dbcmd* **copy** *conflict-algorithm table-name file-name* ?*column-separator*? ?*null-indicator*?

Conflict-algorithm must be one of the SQLite conflict algorithms for the INSERT statement: *rollback*, *abort*, *fail*,*ignore*, or *replace*. See the SQLite Language section for ON CONFLICT for more information. The conflict-algorithm must be specified in lower case.

Table-name must already exists as a table. File-name must exist, and each row must contain the same number of columns as defined in the table. If a line in the file contains more or less than the number of columns defined, the copy method rollbacks any inserts, and returns an error.

Column-separator is an optional column separator string. The default is the ASCII tab character \t.

Null-indicator is an optional string that indicates a column value is null. The default is an empty string. Note that column-separator and null-indicator are optional positional arguments; if null-indicator is specified, a column-separator argument must be specified and precede the null-indicator argument.

The copy method implements similar functionality to the **.import** SQLite shell command.

The "timeout" method

The "timeout" method is used to control how long the SQLite library will wait for locks to clear before giving up on a database transaction. The default timeout is 0 millisecond. (In other words, the default behavior is not to wait at all.)

The SQLite database allows multiple simultaneous readers or a single writer but not both. If any process is writing to the database no other process is allows to read or write. If any process is reading the database other processes are allowed to read but not write. The entire database shared a single lock.

When SQLite tries to open a database and finds that it is locked, it can optionally delay for a short while and try to open the file again. This process repeats until the query times out and SQLite returns a failure. The timeout is adjustable. It is set to 0 by default so that if the database is locked, the SQL statement fails immediately. But you can use the "timeout" method to change the timeout value to a positive number. For example:

> **db1 timeout 2000**

The argument to the timeout method is the maximum number of milliseconds to wait for the lock to clear. So in the example above, the maximum delay would be 2 seconds.

The "busy" method

The "busy" method, like "timeout", only comes into play when the database is locked. But the "busy" method gives the programmer much more control over what action to take. The "busy" method specifies a callback Tcl procedure that is invoked whenever SQLite tries to open a locked database. This callback can do whatever is desired. Presumably, the callback will do some other useful work for a short while (such as service GUI events) then return so that the lock can be tried again. The callback procedure should return "0" if it wants SQLite to try again to open the database and should return "1" if it wants SQLite to abandon the current operation.

The "enable_load_extension" method

The extension loading mechanism of SQLite (accessed using the load_extension() SQL function) is turned off by default. This is a security precaution. If an application wants to make use of the load_extension() function it must first turn the capability on using this method.

This method takes a single boolean argument which will turn the extension loading functionality on or off.

This method maps to the sqlite3_enable_load_extension() C/C++ interface.

The "exists" method

The "exists" method is similar to "onecolumn" and "eval" in that it executes SQL statements. The difference is that the "exists" method always returns a boolean value which is TRUE if a query in the SQL statement it executes returns one or more rows and FALSE if the SQL returns an empty set.

The "exists" method is often used to test for the existence of rows in a table. For example:

> **if {[db exists {SELECT 1 FROM table1 WHERE user=$user}]} {
>
> # Processing if $user exists
>
> } else {
>
> # Processing if $user does not exist
>
> }**

The "last_insert_rowid" method

The "last_insert_rowid" method returns an integer which is the ROWID of the most recently inserted database row.

The "function" method

The "function" method registers new SQL functions with the SQLite engine. The arguments are the name of the new SQL function and a TCL command that implements that function. Arguments to the function are appended to the TCL command before it is invoked.

The following example creates a new SQL function named "hex" that converts its numeric argument in to a hexadecimal encoded string:

> **db function hex {format 0x%X}**

The "function" method accepts the following options:

> **-argcount** *INTEGER*
>
> Specify the number of arguments that the SQL function accepts. The default value of -1 means any number of arguments.
>
> **-deterministic**
>
> This option indicates that the function will always return the same answer given the same argument values. The SQLite query optimizer uses this information to cache answers from function calls with constant inputs and reuse the result rather than invoke the function repeatedly.

The "nullvalue" method

The "nullvalue" method changes the representation for NULL returned as result of the "eval" method.

> **db1 nullvalue NULL**

The "nullvalue" method is useful to differ between NULL and empty column values as Tcl lacks a NULL representation. The default representation for NULL values is an empty string.

The "onecolumn" method

The "onecolumn" method works like "eval" in that it evaluates the SQL query statement given as its argument. The difference is that "onecolumn" returns a single element which is the first column of the first row of the query result.

This is a convenience method. It saves the user from having to do a " `[lindex ... 0]` " on the results of an "eval" in order to extract a single column result.

The "changes" method

The "changes" method returns an integer which is the number of rows in the database that were inserted, deleted, and/or modified by the most recent "eval" method.

The "total_changes" method

The "total_changes" method returns an integer which is the number of rows in the database that were inserted, deleted, and/or modified since the current database connection was first opened.

The "authorizer" method

The "authorizer" method provides access to the sqlite3_set_authorizer C/C++ interface. The argument to authorizer is the name of a procedure that is called when SQL statements are being compiled in order to authorize certain operations. The callback procedure takes 5 arguments which describe the operation being coded. If the callback returns the text string "SQLITE_OK", then the operation is allowed. If it returns "SQLITE_IGNORE", then the operation is silently disabled. If the return is "SQLITE_DENY" then the compilation fails with an error.

If the argument is an empty string then the authorizer is disabled. If the argument is omitted, then the current authorizer is returned.

The "progress" method

This method registers a callback that is invoked periodically during query processing. There are two arguments: the number of SQLite virtual machine opcodes between invocations, and the TCL command to invoke. Setting the progress callback to an empty string disables it.

The progress callback can be used to display the status of a lengthy query or to process GUI events during a lengthy query.

The "collate" method

This method registers new text collating sequences. There are two arguments: the name of the collating sequence and the name of a TCL procedure that implements a comparison function for the collating sequence.

For example, the following code implements a collating sequence called "NOCASE" that sorts in text order without regard to case:

```
proc nocase_compare {a b} { return [string compare [string tolower $a] [string
tolower $b]] } db collate NOCASE nocase_compare
```

The "collation_needed" method

This method registers a callback routine that is invoked when the SQLite engine needs a particular collating sequence but does not have that collating sequence registered. The callback can register the collating sequence. The callback is invoked with a single parameter which is the name of the needed collating sequence.

The "commit_hook" method

This method registers a callback routine that is invoked just before SQLite tries to commit changes to a database. If the callback throws an exception or returns a non-zero result, then the transaction rolls back rather than commit.

The "rollback_hook" method

This method registers a callback routine that is invoked just before SQLite tries to do a rollback. The script argument is run without change.

The "status" method

This method returns status information from the most recently evaluated SQL statement. The status method takes a single argument which should be either "steps" or "sorts". If the argument is "steps", then the method returns the number of full table scan steps that the previous SQL statement evaluated. If the argument is "sorts", the method returns the number of sort operations. This information can be used to detect queries that are not using indices to speed search or sorting.

The status method is basically a wrapper on the sqlite3_stmt_status() C-language interface.

The "update_hook" method

This method registers a callback routine that is invoked just before each row is modified by an UPDATE, INSERT, or DELETE statement. Four arguments are appended to the callback before it is invoked:

- The keyword "INSERT", "UPDATE", or "DELETE", as appropriate

- The name of the database which is being changed
- The table that is being changed
- The rowid of the row in the table being changed

The "wal_hook" method

This method registers a callback routine that is invoked after transaction commit when the database is in WAL mode. Two arguments are appended to the callback command before it is invoked:

- The name of the database on which the transaction was committed
- The number of entries in the write-ahead log (WAL) file for that database

This method might decide to run a checkpoint either itself or as a subsequent idle callback. Note that SQLite only allows a single WAL hook. By default this single WAL hook is used for the auto-checkpointing. If you set up an explicit WAL hook, then that one WAL hook must ensure that checkpoints are occurring since the auto-checkpointing mechanism will be disabled.

The "incrblob" method

This method opens a TCL channel that can be used to read or write into a preexisting BLOB in the database. The syntax is like this:

> *dbcmd* **incrblob ?-readonly?** *?DB? TABLE COLUMN ROWID*

The command returns a new TCL channel for reading or writing to the BLOB. The channel is opened using the underlying sqlite3_blob_open() C-language interface. Close the channel using the **close** command of TCL.

The "errorcode" method

This method returns the numeric error code that resulted from the most recent SQLite operation.

The "trace" method

The "trace" method registers a callback that is invoked as each SQL statement is compiled. The text of the SQL is appended as a single string to the command before it is invoked. This can be used (for example) to keep a log of all SQL operations that an application performs.

The "backup" method

The "backup" method makes a backup copy of a live database. The command syntax is like this:

> *dbcmd* **backup** ?*source-database*? *backup-filename*

The optional *source-database* argument tells which database in the current connection should be backed up. The default value is **main** (or, in other words, the primary database file). To back up TEMP tables use **temp**. To backup an auxiliary database added to the connection using the ATTACH command, use the name of that database as it was assigned in the ATTACH command.

The *backup-filename* is the name of a file into which the backup is written. *Backup-filename* does not have to exist ahead of time, but if it does, it must be a well-formed SQLite database.

The "restore" method

The "restore" method copies the content from a separate database file into the current database connection, overwriting any preexisting content. The command syntax is like this:

> *dbcmd* **restore** ?*target-database*? *source-filename*

The optional *target-database* argument tells which database in the current connection should be overwritten with new content. The default value is **main** (or, in other words, the primary database file). To repopulate the TEMP tables use **temp**. To overwrite an auxiliary database added to the connection using the ATTACH command, use the name of that database as it was assigned in the ATTACH command.

The *source-filename* is the name of an existing well-formed SQLite database file from which the content is extracted.

The "version" method

Return the current library version. For example, "3.6.17".

The "profile" method

This method is used to profile the execution of SQL statements run by the application. The syntax is as follows:

> *dbcmd* **profile** ?*script*?

Unless *script* is an empty string, this method arranges for the *script* to be evaluated after the execution of each SQL statement. Two arguments are appended to *script* before it is invoked: the text of the SQL statement executed and the time elapsed while executing the statement, in nanoseconds.

A database handle may only have a single profile script registered at any time. If there is already a script registered when the profile method is invoked, the previous profile script is replaced by the new one. If the *script* argument is an empty string, any previously registered profile callback is canceled but no new profile script is registered.

SQL As Understood By SQLite

SQLite understands most of the standard SQL language. But it does omit some features while at the same time adding a few features of its own. This document attempts to describe precisely what parts of the SQL language SQLite does and does not support. A list of SQL keywords is also provided. The SQL language syntax is described by syntax diagrams.

The following syntax documentation topics are available:

- aggregate functions
- ALTER TABLE
- ANALYZE
- ATTACH DATABASE
- BEGIN TRANSACTION
- comment
- COMMIT TRANSACTION
- core functions
- CREATE INDEX
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- CREATE VIRTUAL TABLE

- date and time functions
- DELETE
- DETACH DATABASE
- DROP INDEX
- DROP TABLE
- DROP TRIGGER
- DROP VIEW
- END TRANSACTION
- EXPLAIN
- expression
- INDEXED BY
- INSERT
- keywords

- ON CONFLICT clause
- PRAGMA
- REINDEX

- RELEASE SAVEPOINT
- REPLACE
- ROLLBACK TRANSACTION
- SAVEPOINT
- SELECT
- UPDATE
- VACUUM
- WITH clause

The routines sqlite3_prepare_v2(), sqlite3_prepare(), sqlite3_prepare16(), sqlite3_prepare16_v2(), sqlite3_exec(), and sqlite3_get_table() accept an SQL statement list (sql-stmt-list) which is a semicolon-separated list of statements.

sql-stmt-list:

Each SQL statement in the statement list is an instance of the following:

sql-stmt:

PRAGMA Statements

The PRAGMA statement is an SQL extension specific to SQLite and used to modify the operation of the SQLite library or to query the SQLite library for internal (non-table) data. The PRAGMA statement is issued using the same interface as other SQLite commands (e.g. SELECT, INSERT) but is different in the following important respects:

- Specific pragma statements may be removed and others added in future releases of SQLite. There is no guarantee of backwards compatibility.
- No error messages are generated if an unknown pragma is issued. Unknown pragmas are simply ignored. This means if there is a typo in a pragma statement the library does not inform the user of the fact.
- Some pragmas take effect during the SQL compilation stage, not the execution stage. This means if using the C-language sqlite3_prepare(), sqlite3_step(), sqlite3_finalize() API (or similar in a wrapper interface), the pragma may run during the sqlite3_prepare() call, not during the sqlite3_step() call as normal SQL statements do. Or the pragma might run during sqlite3_step() just like normal SQL statements. Whether or not the pragma runs during sqlite3_prepare() or sqlite3_step() depends on the pragma and on the specific release of SQLite.
- The pragma command is specific to SQLite and is very unlikely to be compatible with any other SQL database engine.

The C-language API for SQLite provides the SQLITE_FCNTL_PRAGMA file control which gives VFS implementations the opportunity to add new PRAGMA statements or to override the meaning of built-in PRAGMA statements.

PRAGMA command syntax

pragma-stmt: <button id="x1469" onclick="hideorshow("x1469","x1470")">hide</button>

> **pragma-value:** <button id="x1471" onclick="hideorshow("x1471","x1472")">hide</button>
>
> > **signed-number:** <button id="x1473" onclick="hideorshow("x1473","x1474")">show</button>

A pragma can take either zero or one argument. The argument is may be either in parentheses or it may be separated from the pragma name by an equal sign. The two syntaxes yield identical results. In many pragmas, the argument is a boolean. The boolean can be one of:

1 yes true on 0 no false off

Keyword arguments can optionally appear in quotes. (Example: `'yes' [FALSE]`.) Some pragmas takes a string literal as their argument. When pragma takes a keyword argument, it will usually also take a numeric equivalent as well. For example, "0" and "no" mean the same thing, as does "1" and "yes". When querying the value of a setting, many pragmas return the number rather than the keyword.

A pragma may have an optional schema-name before the pragma name. The schema-name is the name of an ATTACH-ed database or "main" or "temp" for the main and the TEMP databases. If the optional schema name is omitted, "main" is assumed. In some pragmas, the schema name is meaningless and is simply ignored. In the documentation below, pragmas for which the schema name is meaningful are shown with a "*schema.*" prefix.

List Of PRAGMAs

- application_id
- auto_vacuum
- automatic_index
- busy_timeout
- cache_size
- cache_spill
- case_sensitive_like
- cell_size_check
- checkpoint_fullfsync
- collation_list

- compile_options
- ~~count_changes~~[1]
- ~~data_store_directory~~[1]
- data_version
- database_list
- ~~default_cache_size~~[1]
- defer_foreign_keys
- ~~empty_result_callbacks~~[1]
- encoding
- foreign_key_check
- foreign_key_list
- foreign_keys

- freelist_count
- ~~full_column_names~~[1]
- fullfsync
- ignore_check_constraints
- incremental_vacuum
- index_info
- index_list
- index_xinfo
- integrity_check
- journal_mode
- journal_size_limit
- legacy_file_format
- locking_mode
- max_page_count
- mmap_size
- page_count
- page_size
- *parser_trace*[2]
- query_only
- quick_check
- read_uncommitted
- recursive_triggers

- reverse_unordered_selects
- schema_version
- secure_delete
- ~~short_column_names~~[1]
- shrink_memory

- soft_heap_limit
- *stats*[3]
- synchronous
- table_info
- temp_store
- ~~temp_store_directory~~[1]
- threads
- user_version
- *vdbe_addoptrace*[2]
- *vdbe_debug*[2]
- *vdbe_listing*[2]
- *vdbe_trace*[2]
- wal_autocheckpoint
- wal_checkpoint
- writable_schema

Notes:

1. Pragmas whose names are marked through in the list above are deprecated. They are not maintained. They continue to exist for historical compatibility only. Do not use the deprecated pragmas in new applications. Remove deprecated pragmas from existing applications at your earliest opportunity.
2. These pragmas are used for debugging SQLite and are only available when SQLite is compiled using SQLITE_DEBUG.
3. These pragmas are used for testing SQLite and are not recommended for use in application programs.

PRAGMA *schema.***application_id; PRAGMA** *schema.***application_id =** *integer* **;**

The application_id PRAGMA is used to query or set the 32-bit unsigned big-endian "Application ID" integer located at offset 68 into the database header. Applications that use SQLite as their application file-format should set the Application ID integer to a unique integer so that utilities such as file(1) can determine the specific file type rather than just reporting "SQLite3 Database". A list of assigned application IDs can be seen by consulting the magic.txt file in the SQLite source repository.

PRAGMA *schema.***auto_vacuum; PRAGMA** *schema.***auto_vacuum =** *0* | *NONE* | *1* | *FULL* | *2* | *INCREMENTAL***;**

Query or set the auto-vacuum status in the database.

The default setting for auto-vacuum is 0 or "none", unless the SQLITE_DEFAULT_AUTOVACUUM compile-time option is used. The "none" setting means that auto-vacuum is disabled. When auto-vacuum is disabled and data is deleted data from a database, the database file remains the same size. Unused database file pages are added to a "freelist" and reused for subsequent inserts. So no database file space is lost. However, the database file does not shrink. In this mode the VACUUM command can be used to rebuild the entire database file and thus reclaim unused disk space.

When the auto-vacuum mode is 1 or "full", the freelist pages are moved to the end of the database file and the database file is truncated to remove the freelist pages at every transaction commit. Note, however, that auto-vacuum only truncates the freelist pages from the file. Auto-vacuum does not defragment the database nor repack individual database pages the way that the VACUUM command does. In fact, because it moves pages around within the file, auto-vacuum can actually make fragmentation worse.

Auto-vacuuming is only possible if the database stores some additional information that allows each database page to be traced backwards to its referrer. Therefore, auto-vacuuming must be turned on before any tables are created. It is not possible to enable or disable auto-vacuum after a table has been created.

When the value of auto-vacuum is 2 or "incremental" then the additional information needed to do auto-vacuuming is stored in the database file but auto-vacuuming does not occur automatically at each commit as it does with auto_vacuum=full. In incremental mode, the separate incremental_vacuum pragma must be invoked to cause the auto-vacuum to occur.

The database connection can be changed between full and incremental autovacuum mode at any time. However, changing from "none" to "full" or "incremental" can only occur when the database is new (no tables have yet been created) or by running the VACUUM command. To change auto-vacuum modes, first use the auto_vacuum pragma to set the new desired mode, then invoke the VACUUM command to reorganize the entire database file. To change from "full" or "incremental" back to "none" always requires running VACUUM even on an empty database.

When the auto_vacuum pragma is invoked with no arguments, it returns the current auto_vacuum mode.

PRAGMA automatic_index; PRAGMA automatic_index = *boolean***;**

Query, set, or clear the automatic indexing capability.

Automatic indexing is enabled by default as of version 3.7.17, but this might change in future releases of SQLite.

PRAGMA busy_timeout; PRAGMA busy_timeout = *milliseconds***;**

Query or change the setting of the busy timeout. This pragma is an alternative to the sqlite3_busy_timeout() C-language interface which is made available as a pragma for use with language bindings that do not provide direct access to sqlite3_busy_timeout().

Each database connection can only have a single busy handler. This PRAGMA sets the busy handler for the process, possibly overwriting any previously set busy handler.

PRAGMA *schema.***cache_size; PRAGMA** *schema.***cache_size =** *pages***; PRAGMA** *schema.***cache_size = -***kibibytes***;**

Query or change the suggested maximum number of database disk pages that SQLite will hold in memory at once per open database file. Whether or not this suggestion is honored is at the discretion of the Application Defined Page Cache. The default page cache that is built into SQLite honors the request, however alternative application-defined page cache implementations may choose to interpret the suggested cache size in different ways or to ignore it all together. The default suggested cache size is -2000, which means the cache size is limited to 2048000 bytes of memory. The default suggested cache size can be altered using the SQLITE_DEFAULT_CACHE_SIZE compile-time options. The TEMP database has a default suggested cache size of 0 pages.

If the argument N is positive then the suggested cache size is set to N. If the argument N is negative, then the number of cache pages is adjusted to use approximately abs(N*1024) bytes of memory. *Backwards compatibility note:* The behavior of cache_size with a negative N was different in SQLite versions prior to 3.7.10. In version 3.7.9 and earlier, the number of pages in the cache was set to the absolute value of N.

When you change the cache size using the cache_size pragma, the change only endures for the current session. The cache size reverts to the default value when the database is closed and reopened.

PRAGMA cache_spill; PRAGMA cache_spill=*boolean***; PRAGMA** *schema.***cache*spill=_N*;**

The cache_spill pragma enables or disables the ability of the pager to spill dirty cache pages to the database file in the middle of a transaction. Cache_spill is enabled by default and most applications should leave it that way as cache spilling is usually advantageous. However, a cache spill has the side-effect of acquiring an EXCLUSIVE lock on the database file. Hence, some applications that have large long-running transactions may want to disable cache spilling in order to prevent the application from acquiring an exclusive lock on the database until the moment that the transaction COMMITs.

The "PRAGMA cache*spill=_N*" form of this pragma sets a minimum cache size threshold required for spilling to occur. The number of pages in cache must exceed both the cache_spill threshold and the maximum cache size set by the PRAGMA cache_size statement in order for spilling to occur.

The "PRAGMA cache*spill=_boolean*" form of this pragma applies across all databases attached to the database connection. But the "PRAGMA cache*spill=_N*" form of this statement only applies to the "main" schema or whatever other schema is specified as part of the statement.

PRAGMA case_sensitive_like = *boolean***;**

The default behavior of the LIKE operator is to ignore case for ASCII characters. Hence, by default **'a' LIKE 'A'** is true. The case_sensitive_like pragma installs a new application-defined LIKE function that is either case sensitive or insensitive depending on the value of the case_sensitive_like pragma. When case_sensitive_like is disabled, the default LIKE behavior is expressed. When case_sensitive_like is enabled, case becomes significant. So, for example, **'a' LIKE 'A'** is false but **'a' LIKE 'a'** is still true.

This pragma uses sqlite3_create_function() to overload the LIKE and GLOB functions, which may override previous implementations of LIKE and GLOB registered by the application. This pragma only changes the behavior of the SQL LIKE operator. It does not change the behavior of the sqlite3_strlike() C-language interface, which is always case insensitive.

PRAGMA cell_size_check PRAGMA cell_size_check = *boolean***;**

The cell_size_check pragma enables or disables additional sanity checking on database b-tree pages as they are initially read from disk. With cell size checking enabled, database corruption is detected earlier and is less likely to "spread". However, there is a small performance hit for doing the extra checks and so cell size checking is turned off by default.

PRAGMA checkpoint_fullfsync PRAGMA checkpoint_fullfsync = *boolean***;**

Query or change the fullfsync flag for checkpoint operations. If this flag is set, then the F_FULLFSYNC syncing method is used during checkpoint operations on systems that support F_FULLFSYNC. The default value of the checkpoint_fullfsync flag is off. Only Mac OS-X supports F_FULLFSYNC.

If the fullfsync flag is set, then the F_FULLFSYNC syncing method is used for all sync operations and the checkpoint_fullfsync setting is irrelevant.

PRAGMA collation_list;

Return a list of the collating sequences defined for the current database connection.

PRAGMA compile_options;

This pragma returns the names of compile-time options used when building SQLite, one option per row. The "SQLITE_" prefix is omitted from the returned option names. See also the sqlite3_compileoption_get() C/C++ interface and the sqlite_compileoption_get() SQL functions.

PRAGMA count_changes; PRAGMA count_changes = boolean;

Query or change the count-changes flag. Normally, when the count-changes flag is not set, INSERT, UPDATE and DELETE statements return no data. When count-changes is set, each of these commands returns a single row of data consisting of one integer value - the number of rows inserted, modified or deleted by the command. The returned change count does not include any insertions, modifications or deletions performed by triggers, or any changes made automatically by foreign key actions.

Another way to get the row change counts is to use the sqlite3_changes() or sqlite3_total_changes() interfaces. There is a subtle different, though. When an INSERT, UPDATE, or DELETE is run against a view using an INSTEAD OF trigger, the count_changes pragma reports the number of rows in the view that fired the trigger, whereas sqlite3_changes() and sqlite3_total_changes() do not.

This pragma is deprecated and exists for backwards compatibility only. New applications should avoid using this pragma. Older applications should discontinue use of this pragma at the earliest opportunity. This pragma may be omitted from the build when SQLite is compiled using SQLITE_OMIT_DEPRECATED.

PRAGMA data_store_directory; PRAGMA data_store_directory = '*directory-name*';

Query or change the value of the sqlite3_data_directory global variable, which windows operating-system interface backends use to determine where to store database files specified using a relative pathname.

Changing the data_store_directory setting is not threadsafe. Never change the data_store_directory setting if another thread within the application is running any SQLite interface at the same time. Doing so results in undefined behavior. Changing the data_store_directory setting writes to the sqlite3_data_directory global variable and that global variable is not protected by a mutex.

This facility is provided for WinRT which does not have an OS mechanism for reading or changing the current working directory. The use of this pragma in any other context is discouraged and may be disallowed in future releases.

This pragma is deprecated and exists for backwards compatibility only. New applications should avoid using this pragma. Older applications should discontinue use of this pragma at the earliest opportunity. This pragma may be omitted from the build when SQLite is compiled using SQLITE_OMIT_DEPRECATED.

PRAGMA *schema.*data_version;

The "PRAGMA data_version" command provides an indication that the database file has been modified. Interactive programs that hold database content in memory or that display database content on-screen can use the PRAGMA data_version command to determine if they need to flush and reload their memory or update the screen display.

The integer values returned by two invocations of "PRAGMA data_version" from the same connection will be different if changes were committed to the database by any other connection in the interim. The "PRAGMA data_version" value is unchanged for commits made on the same database connection. The behavior of "PRAGMA data_version" is the same for all database connections, including database connections in separate processes and shared cache database connections.

The "PRAGMA data_version" value is a local property of each database connection and so values returned by two concurrent invocations of "PRAGMA data_version" on separate database connections are often different even though the underlying database is identical. It is only meaningful to compare the "PRAGMA data_version" values returned by the same database connection at two different points in time.

PRAGMA database_list;

This pragma works like a query to return one row for each database attached to the current database connection. The second column is the "main" for the main database file, "temp" for the database file used to store TEMP objects, or the name of the ATTACHed database for other database files. The third column is the name of the database file itself, or an empty string if the database is not associated with a file.

PRAGMA *schema.***default_cache_size; PRAGMA** *schema.***default_cache_size =** *Number-of-pages***;**

This pragma queries or sets the suggested maximum number of pages of disk cache that will be allocated per open database file. The difference between this pragma and cache_size is that the value set here persists across database connections. The value of the default cache size is stored in the 4-byte big-endian integer located at offset 48 in the header of the database file.

This pragma is deprecated and exists for backwards compatibility only. New applications should avoid using this pragma. Older applications should discontinue use of this pragma at the earliest opportunity. This pragma may be omitted from the build when SQLite is compiled using SQLITE_OMIT_DEPRECATED.

PRAGMA defer_foreign_keys PRAGMA defer_foreign_keys = *boolean***;**

When the defer_foreign_keys PRAGMA is on, enforcement of all foreign key constraints is delayed until the outermost transaction is committed. The defer_foreign_keys pragma defaults to OFF so that foreign key constraints are only deferred if they are created as "DEFERRABLE INITIALLY DEFERRED". The defer_foreign_keys pragma is automatically switched off at each COMMIT or ROLLBACK. Hence, the defer_foreign_keys pragma must be separately enabled for each transaction. This pragma is only meaningful if foreign key constraints are enabled, of course.

The sqlite3_db_status(db,SQLITE_DBSTATUS_DEFERRED_FKS,...) C-language interface can be used during a transaction to determine if there are deferred and unresolved foreign key constraints.

PRAGMA empty_result_callbacks; PRAGMA empty_result_callbacks = *boolean***;**

Query or change the empty-result-callbacks flag.

The empty-result-callbacks flag affects the sqlite3_exec() API only. Normally, when the empty-result-callbacks flag is cleared, the callback function supplied to the sqlite3_exec() is not invoked for commands that return zero rows of data. When empty-result-callbacks is set in this situation, the callback function is invoked exactly once, with the third parameter set to 0 (NULL). This is to enable programs that use the sqlite3_exec() API to retrieve column-names even when a query returns no data.

This pragma is deprecated and exists for backwards compatibility only. New applications should avoid using this pragma. Older applications should discontinue use of this pragma at the earliest opportunity. This pragma may be omitted from the build when SQLite is compiled using SQLITE_OMIT_DEPRECATED.

PRAGMA encoding; PRAGMA encoding = "UTF-8"; PRAGMA encoding = "UTF-16"; PRAGMA encoding = "UTF-16le"; PRAGMA encoding = "UTF-16be";

In first form, if the main database has already been created, then this pragma returns the text encoding used by the main database, one of "UTF-8", "UTF-16le" (little-endian UTF-16 encoding) or "UTF-16be" (big-endian UTF-16 encoding). If the main database has not already been created, then the value returned is the text encoding that will be used to create the main database, if it is created by this session.

The second through fifth forms of this pragma set the encoding that the main database will be created with if it is created by this session. The string "UTF-16" is interpreted as "UTF-16 encoding using native machine byte-ordering". It is not possible to change the text encoding of a database after it has been created and any attempt to do so will be silently ignored.

Once an encoding has been set for a database, it cannot be changed.

Databases created by the ATTACH command always use the same encoding as the main database. An attempt to ATTACH a database with a different text encoding from the "main" database will fail.

PRAGMA *schema.***foreign_key_check; PRAGMA** *schema.***foreign_key_check(***table-name***);**

The foreign*key_check pragma checks the database, or the table called "_table-name*", for foreign key constraints that are violated and returns one row of output for each violation. There are four columns in each result row. The first column is the name of the table that contains the REFERENCES clause. The second column is the rowid of the row that contains the invalid REFERENCES clause. The third column is the name of the table that is referred to. The fourth column is the index of the specific foreign key constraint that failed. The fourth column in the output of the foreign*key_check pragma is the same integer as the first column in the output of the foreign_key_list pragma. When a "_table-name*" is specified, the only foreign key constraints checked are those created by REFERENCES clauses in the CREATE TABLE statement for *table-name*.

PRAGMA foreign_key_list(*table-name***);**

This pragma returns one row for each foreign key constraint created by a REFERENCES clause in the CREATE TABLE statement of table "*table-name*".

PRAGMA foreign_keys; PRAGMA foreign_keys = *boolean***;**

Query, set, or clear the enforcement of foreign key constraints.

This pragma is a no-op within a transaction; foreign key constraint enforcement may only be enabled or disabled when there is no pending BEGIN or SAVEPOINT.

Changing the foreign_keys setting affects the execution of all statements prepared using the database connection, including those prepared before the setting was changed. Any existing statements prepared using the legacy sqlite3_prepare() interface may fail with an SQLITE_SCHEMA error after the foreign_keys setting is changed.

As of SQLite version 3.6.19, the default setting for foreign key enforcement is OFF. However, that might change in a future release of SQLite. The default setting for foreign key enforcement can be specified at compile-time using the SQLITE_DEFAULT_FOREIGN_KEYS preprocessor macro. To minimize future problems, applications should set the foreign key enforcement flag as required by the application and not depend on the default setting.

PRAGMA *schema.***freelist_count;**

Return the number of unused pages in the database file.

PRAGMA full_column_names; PRAGMA full_column_names = *boolean***;**

Query or change the full_column_names flag. This flag together with the short_column_names flag determine the way SQLite assigns names to result columns of SELECT statements. Result columns are named by applying the following rules in order:

1. If there is an AS clause on the result, then the name of the column is the right-hand side of the AS clause.
2. If the result is a general expression, not a just the name of a source table column, then the name of the result is a copy of the expression text.
3. If the short_column_names pragma is ON, then the name of the result is the name of the source table column without the source table name prefix: COLUMN.
4. If both pragmas short_column_names and full_column_names are OFF then case (2) applies.
5. The name of the result column is a combination of the source table and source column name: TABLE.COLUMN

This pragma is deprecated and exists for backwards compatibility only. New applications should avoid using this pragma. Older applications should discontinue use of this pragma at the earliest opportunity. This pragma may be omitted from the build when SQLite is compiled using SQLITE_OMIT_DEPRECATED.

PRAGMA fullfsync PRAGMA fullfsync = *boolean*;

Query or change the fullfsync flag. This flag determines whether or not the F_FULLFSYNC syncing method is used on systems that support it. The default value of the fullfsync flag is off. Only Mac OS X supports F_FULLFSYNC.

See also checkpoint_fullfsync.

PRAGMA ignore_check_constraints = *boolean*;

This pragma enables or disables the enforcement of CHECK constraints. The default setting is off, meaning that CHECK constraints are enforced by default.

PRAGMA *schema.*incremental_vacuum*(N)*;

The incremental*vacuum pragma causes up to _N* pages to be removed from the freelist. The database file is truncated by the same amount. The incremental*vacuum pragma has no effect if the database is not in auto_vacuum=incremental mode or if there are no pages on the freelist. If there are fewer than _N* pages on the freelist, or if *N* is less than 1, or if *N* is omitted entirely, then the entire freelist is cleared.

PRAGMA *schema.***index_info(***index-name***);**

This pragma returns one row for each key column in the named index. A key column is a column that is actually named in the CREATE INDEX index statement or UNIQUE constraint or PRIMARY KEY constraint that created the index. Index entries also usually contain auxiliary columns that point back to the table row being indexed. The auxiliary index-columns are not shown by the index_info pragma, but they are listed by the index_xinfo pragma.

Output columns from the index_info pragma are as follows:

1. The rank of the column within the index. (0 means left-most.)
2. The rank of the column within the table being indexed.
3. The name of the column being indexed.

PRAGMA *schema.***index_list(***table-name***);**

This pragma returns one row for each index associated with the given table.

Output columns from the index_list pragma are as follows:

1. A sequence number assigned to each index for internal tracking purposes.
2. The name of the index.
3. "1" if the index is UNIQUE and "0" if not.
4. "c" if the index was created by a CREATE INDEX statement, "u" if the index was created by a UNIQUE constraint, or "pk" if the index was created by a PRIMARY KEY constraint.
5. "1" if the index is a partial index and "0" if not.

PRAGMA *schema.***index_xinfo(***index-name***);**

This pragma returns information about every column in an index. Unlike this index_info pragma, this pragma returns information about every column in the index, not just the key columns. (A key column is a column that is actually named in the CREATE INDEX index statement or UNIQUE constraint or PRIMARY KEY constraint that created the index. Auxiliary columns are additional columns needed to locate the table entry that corresponds to each index entry.)

Output columns from the index_xinfo pragma are as follows:

1. The rank of the column within the index. (0 means left-most. Key columns come before auxiliary columns.)

2. The rank of the column within the table being indexed, or -1 if the index-column is the rowid of the table being indexed.
3. The name of the column being indexed, or NULL if the index-column is the rowid of the table being indexed.
4. 1 if the index-column is sorted in reverse (DESC) order by the index and 0 otherwise.
5. The name for the collating sequence used to compare values in the index-column.
6. 1 if the index-column is a key column and 0 if the index-column is an auxiliary column.

PRAGMA *schema*.integrity_check; PRAGMA *schema*.integrity_check(*N*)

This pragma does an integrity check of the entire database. The integrity*check pragma looks for out-of-order records, missing pages, malformed records, missing index entries, and UNIQUE and NOT NULL constraint errors. If the integrity_check pragma finds problems, strings are returned (as multiple rows with a single column per row) which describe the problems. Pragma integrity_check will return at most _N* errors before the analysis quits, with N defaulting to 100. If pragma integrity_check finds no errors, a single row with the value 'ok' is returned.

PRAGMA integrity_check does not find FOREIGN KEY errors. Use the PRAGMA foreign_key_check command for to find errors in FOREIGN KEY constraints.

See also the PRAGMA quick_check command which does most of the checking of PRAGMA integrity_check but runs much faster.

PRAGMA *schema*.journal_mode; PRAGMA *schema*.journal*mode = _DELETE* | *TRUNCATE* | *PERSIST* | *MEMORY* | *WAL* | *OFF*

This pragma queries or sets the journal mode for databases associated with the current database connection.

The first form of this pragma queries the current journaling mode for *database*. When *database* is omitted, the "main" database is queried.

The second form changes the journaling mode for "*database*" or for all attached databases if "*database*" is omitted. The new journal mode is returned. If the journal mode could not be changed, the original journal mode is returned.

The DELETE journaling mode is the normal behavior. In the DELETE mode, the rollback journal is deleted at the conclusion of each transaction. Indeed, the delete operation is the action that causes the transaction to commit. (See the document titled Atomic Commit In SQLite for additional detail.)

The TRUNCATE journaling mode commits transactions by truncating the rollback journal to zero-length instead of deleting it. On many systems, truncating a file is much faster than deleting the file since the containing directory does not need to be changed.

The PERSIST journaling mode prevents the rollback journal from being deleted at the end of each transaction. Instead, the header of the journal is overwritten with zeros. This will prevent other database connections from rolling the journal back. The PERSIST journaling mode is useful as an optimization on platforms where deleting or truncating a file is much more expensive than overwriting the first block of a file with zeros. See also: PRAGMA journal_size_limit and SQLITE_DEFAULT_JOURNAL_SIZE_LIMIT.

The MEMORY journaling mode stores the rollback journal in volatile RAM. This saves disk I/O but at the expense of database safety and integrity. If the application using SQLite crashes in the middle of a transaction when the MEMORY journaling mode is set, then the database file will very likely go corrupt.

The WAL journaling mode uses a write-ahead log instead of a rollback journal to implement transactions. The WAL journaling mode is persistent; after being set it stays in effect across multiple database connections and after closing and reopening the database. A database in WAL journaling mode can only be accessed by SQLite version 3.7.0 or later.

The OFF journaling mode disables the rollback journal completely. No rollback journal is ever created and hence there is never a rollback journal to delete. The OFF journaling mode disables the atomic commit and rollback capabilities of SQLite. The ROLLBACK command no longer works; it behaves in an undefined way. Applications must avoid using the ROLLBACK command when the journal mode is OFF. If the application crashes in the middle of a transaction when the OFF journaling mode is set, then the database file will very likely go corrupt.

Note that the journal_mode for an in-memory database is either MEMORY or OFF and can not be changed to a different value. An attempt to change the journal_mode of an in-memory database to any setting other than MEMORY or OFF is ignored. Note also that the journal_mode cannot be changed while a transaction is active.

PRAGMA *schema.***journal_size_limit PRAGMA** *schema.***journal_size_limit =** *N* **;**

If a database connection is operating in exclusive locking mode or in persistent journal mode (PRAGMA journal_mode=persist) then after committing a transaction the rollback journal file may remain in the file-system. This increases performance for subsequent transactions since overwriting an existing file is faster than append to a file, but it also consumes file-system space. After a large transaction (e.g. a VACUUM), the rollback journal file may consume a very large amount of space.

Similarly, in WAL mode, the write-ahead log file is not truncated following a checkpoint. Instead, SQLite reuses the existing file for subsequent WAL entries since overwriting is faster than appending.

The journal_size_limit pragma may be used to limit the size of rollback-journal and WAL files left in the file-system after transactions or checkpoints. Each time a transaction is committed or a WAL file resets, SQLite compares the size of the rollback journal file or WAL file left in the file-system to the size limit set by this pragma and if the journal or WAL file is larger it is truncated to the limit.

The second form of the pragma listed above is used to set a new limit in bytes for the specified database. A negative number implies no limit. To always truncate rollback journals and WAL files to their minimum size, set the journal_size_limit to zero. Both the first and second forms of the pragma listed above return a single result row containing a single integer column - the value of the journal size limit in bytes. The default journal size limit is -1 (no limit). The SQLITE_DEFAULT_JOURNAL_SIZE_LIMIT preprocessor macro can be used to change the default journal size limit at compile-time.

This pragma only operates on the single database specified prior to the pragma name (or on the "main" database if no database is specified.) There is no way to change the journal size limit on all attached databases using a single PRAGMA statement. The size limit must be set separately for each attached database.

PRAGMA legacy*file_format; PRAGMA legacy_file_format = _boolean*

This pragma sets or queries the value of the legacy_file_format flag. When this flag is on, new SQLite databases are created in a file format that is readable and writable by all versions of SQLite going back to 3.0.0. When the flag is off, new databases are created using the latest file format which might not be readable or writable by versions of SQLite prior to 3.3.0.

When the legacy_file_format pragma is issued with no argument, it returns the setting of the flag. This pragma does not tell which file format the current database is using; it tells what format will be used by any newly created databases.

The legacy_file_format pragma is initialized to OFF when an existing database in the newer file format is first opened.

The default file format is set by the SQLITE_DEFAULT_FILE_FORMAT compile-time option.

PRAGMA *schema.***locking_mode; PRAGMA** *schema.***locking***mode = _NORMAL |* ***EXCLUSIVE***

This pragma sets or queries the database connection locking-mode. The locking-mode is either NORMAL or EXCLUSIVE.

In NORMAL locking-mode (the default unless overridden at compile-time using SQLITE_DEFAULT_LOCKING_MODE), a database connection unlocks the database file at the conclusion of each read or write transaction. When the locking-mode is set to EXCLUSIVE, the database connection never releases file-locks. The first time the database is read in EXCLUSIVE mode, a shared lock is obtained and held. The first time the database is written, an exclusive lock is obtained and held.

Database locks obtained by a connection in EXCLUSIVE mode may be released either by closing the database connection, or by setting the locking-mode back to NORMAL using this pragma and then accessing the database file (for read or write). Simply setting the locking-mode to NORMAL is not enough - locks are not released until the next time the database file is accessed.

There are three reasons to set the locking-mode to EXCLUSIVE.

1. The application wants to prevent other processes from accessing the database file.
2. The number of system calls for filesystem operations is reduced, possibly resulting in a small performance increase.
3. WAL databases can be accessed in EXCLUSIVE mode without the use of shared memory. (Additional information)

When the locking_mode pragma specifies a particular database, for example:

> PRAGMA **main.**locking_mode=EXCLUSIVE;

Then the locking mode applies only to the named database. If no database name qualifier precedes the "locking_mode" keyword then the locking mode is applied to all databases, including any new databases added by subsequent ATTACH commands.

The "temp" database (in which TEMP tables and indices are stored) and in-memory databases always uses exclusive locking mode. The locking mode of temp and in-memory databases cannot be changed. All other databases use the normal locking mode by default and are affected by this pragma.

If the locking mode is EXCLUSIVE when first entering WAL journal mode, then the locking mode cannot be changed to NORMAL until after exiting WAL journal mode. If the locking mode is NORMAL when first entering WAL journal mode, then the locking mode can be changed between NORMAL and EXCLUSIVE and back again at any time and without needing to exit WAL journal mode.

PRAGMA *schema.***max_page_count; PRAGMA** *schema.***max_page_count =** *N***;**

Query or set the maximum number of pages in the database file. Both forms of the pragma return the maximum page count. The second form attempts to modify the maximum page count. The maximum page count cannot be reduced below the current database size.

PRAGMA *schema.***mmap_size; PRAGMA** *schema.***mmap_size=***N*

Query or change the maximum number of bytes that are set aside for memory-mapped I/O on a single database. The first form (without an argument) queries the current limit. The second form (with a numeric argument) sets the limit for the specified database, or for all databases if the optional database name is omitted. In the second form, if the database name is omitted, the limit that is set becomes the default limit for all databases that are added to the database connection by subsequent ATTACH statements.

The argument N is the maximum number of bytes of the database file that will be accessed using memory-mapped I/O. If N is zero then memory mapped I/O is disabled. If N is negative, then the limit reverts to the default value determined by the most recent sqlite3_config(SQLITE_CONFIG_MMAP_SIZE), or to the compile time default determined by SQLITE_DEFAULT_MMAP_SIZE if not start-time limit has been set.

The PRAGMA mmap_size statement will never increase the amount of address space used for memory-mapped I/O above the hard limit set by the SQLITE_MAX_MMAP_SIZE compile-time option, nor the hard limit set start-time by the second argument to sqlite3_config(SQLITE_CONFIG_MMAP_SIZE)

The size of the memory-mapped I/O region cannot be changed while the memory-mapped I/O region is in active use, to avoid unmapping memory out from under running SQL statements. For this reason, the mmap_size pragma may be a no-op if the prior mmap_size is non-zero and there are other SQL statements running concurrently on the same database connection.

PRAGMA *schema.***page_count;**

Return the total number of pages in the database file.

PRAGMA *schema.***page_size; PRAGMA** *schema.***page_size =** *bytes***;**

Query or set the page size of the database. The page size must be a power of two between 512 and 65536 inclusive.

When a new database is created, SQLite assigned a page size to the database based on platform and filesystem. For many years, the default page size was almost always 1024 bytes, but beginning with SQLite version 3.12.0 in 2016, the default page size increased to 4096.

The page_size pragma will only cause an immediate change in the page size if it is issued while the database is still empty, prior to the first CREATE TABLE statement. If the page_size pragma is used to specify a new page size just prior to running the VACUUM command and if the database is not in WAL journal mode then VACUUM will change the page size to the new value.

The SQLITE_DEFAULT_PAGE_SIZE compile-time option can be used to change the default page size assigned to new databases.

PRAGMA parser_trace = *boolean***;**

If SQLite has been compiled with the SQLITE_DEBUG compile-time option, then the parser_trace pragma can be used to turn on tracing for the SQL parser used internally by SQLite. This feature is used for debugging SQLite itself.

This pragma is intended for use when debugging SQLite itself. It is only available when the SQLITE_DEBUG compile-time option is used.

PRAGMA query_only; PRAGMA query_only = *boolean***;**

The query_only pragma prevents all changes to database files when enabled.

PRAGMA *schema.***quick_check; PRAGMA** *schema.***quick_check(***N***)**

The pragma is like integrity_check except that it does not verify UNIQUE and NOT NULL constraints and does not verify that index content matches table content. By skipping UNIQUE and NOT NULL and index consistency checks, quick_check is able to run much faster than integrity_check. Otherwise the two pragmas are the same.

PRAGMA read_uncommitted; PRAGMA read_uncommitted = *boolean***;**

Query, set, or clear READ UNCOMMITTED isolation. The default isolation level for SQLite is SERIALIZABLE. Any process or thread can select READ UNCOMMITTED isolation, but SERIALIZABLE will still be used except between connections that share a common page and schema cache. Cache sharing is enabled using the sqlite3_enable_shared_cache() API. Cache sharing is disabled by default.

See SQLite Shared-Cache Mode for additional information.

PRAGMA recursive_triggers; PRAGMA recursive_triggers = ***boolean*****;**

Query, set, or clear the recursive trigger capability.

Changing the recursive_triggers setting affects the execution of all statements prepared using the database connection, including those prepared before the setting was changed. Any existing statements prepared using the legacy sqlite3_prepare() interface may fail with an SQLITE_SCHEMA error after the recursive_triggers setting is changed.

Prior to SQLite version 3.6.18, recursive triggers were not supported. The behavior of SQLite was always as if this pragma was set to OFF. Support for recursive triggers was added in version 3.6.18 but was initially turned OFF by default, for compatibility. Recursive triggers may be turned on by default in future versions of SQLite.

The depth of recursion for triggers has a hard upper limit set by the SQLITE_MAX_TRIGGER_DEPTH compile-time option and a run-time limit set by sqlite3_limit(db,SQLITE_LIMIT_TRIGGER_DEPTH,...).

PRAGMA reverse_unordered_selects; PRAGMA reverse_unordered_selects = ***boolean*****;**

When enabled, this PRAGMA causes many SELECT statements without an ORDER BY clause to emit their results in the reverse order from what they normally would. This can help debug applications that are making invalid assumptions about the result order. The reverse_unordered_selects pragma works for most SELECT statements, however the query planner may sometimes choose an algorithm that is not easily reversed, in which case the output will appear in the same order regardless of the reverse_unordered_selects setting.

SQLite makes no guarantees about the order of results if a SELECT omits the ORDER BY clause. Even so, the order of results does not change from one run to the next, and so many applications mistakenly come to depend on the arbitrary output order whatever that order happens to be. However, sometimes new versions of SQLite will contain optimizer enhancements that will cause the output order of queries without ORDER BY clauses to shift. When that happens, applications that depend on a certain output order might malfunction. By running the application multiple times with this pragma both disabled and enabled, cases where the application makes faulty assumptions about output order can be identified and fixed early, reducing problems that might be caused by linking against a different version of SQLite.

PRAGMA *schema.***schema_version; PRAGMA** *schema.***schema_version =** *integer* **;**
PRAGMA *schema.***user_version; PRAGMA** *schema.***user_version =** *integer* **;**

The pragmas schema_version and user_version are used to set or get the value of the schema-version and user-version, respectively. The schema-version and the user-version are big-endian 32-bit signed integers stored in the database header at offsets 40 and 60, respectively.

The schema-version is usually only manipulated internally by SQLite. It is incremented by SQLite whenever the database schema is modified (by creating or dropping a table or index). The schema version is used by SQLite each time a query is executed to ensure that the internal cache of the schema used when compiling the SQL query matches the schema of the database against which the compiled query is actually executed. Subverting this mechanism by using "PRAGMA schema_version" to modify the schema-version is potentially dangerous and may lead to program crashes or database corruption. Use with caution!

The user-version is not used internally by SQLite. It may be used by applications for any purpose.

PRAGMA *schema.***secure_delete; PRAGMA** *schema.***secure_delete =** *boolean*

Query or change the secure-delete setting. When secure-delete on, SQLite overwrites deleted content with zeros. The default setting is determined by the SQLITE_SECURE_DELETE compile-time option.

When there are attached databases and no database is specified in the pragma, all databases have their secure-delete setting altered. The secure-delete setting for newly attached databases is the setting of the main database at the time the ATTACH command is evaluated.

When multiple database connections share the same cache, changing the secure-delete flag on one database connection changes it for them all.

PRAGMA short_column_names; PRAGMA short_column_names = *boolean***;**

Query or change the short-column-names flag. This flag affects the way SQLite names columns of data returned by SELECT statements. See the full_column_names pragma for full details.

This pragma is deprecated and exists for backwards compatibility only. New applications should avoid using this pragma. Older applications should discontinue use of this pragma at the earliest opportunity. This pragma may be omitted from the build when SQLite is compiled using SQLITE_OMIT_DEPRECATED.

PRAGMA shrink_memory

This pragma causes the database connection on which it is invoked to free up as much memory as it can, by calling sqlite3_db_release_memory().

PRAGMA soft_heap_limit PRAGMA soft_heap_limit=*N*

This pragma invokes the sqlite3_soft_heap_limit64() interface with the argument N, if N is specified and is a non-negative integer. The soft_heap_limit pragma always returns the same integer that would be returned by the sqlite3_soft_heap_limit64(-1) C-language function.

PRAGMA stats;

This pragma returns auxiliary information about tables and indices. The returned information is used during testing to help verify that the query planner is operating correctly. The format and meaning of this pragma will likely change from one release to the next. Because of its volatility, the behavior and output format of this pragma are deliberately undocumented.

The intended use of this pragma is only for testing and validation of SQLite. This pragma is subject to change without notice and is not recommended for use by application programs.

PRAGMA *schema***.synchronous; PRAGMA** *schema***.synchronous =** *0 | OFF | 1 | NORMAL | 2 | FULL | 3 | EXTRA***;**

Query or change the setting of the "synchronous" flag. The first (query) form will return the synchronous setting as an integer. The second form changes the synchronous setting. The meanings of the various synchronous settings are as follows:

EXTRA (3)

EXTRA synchronous is like FULL with the addition that the directory containing a rollback journal is synced after that journal is unlinked to commit a transaction in DELETE mode. EXTRA provides additional durability if the commit is followed closely by a power loss.

FULL (2)

When synchronous is FULL (2), the SQLite database engine will use the xSync method of the VFS to ensure that all content is safely written to the disk surface prior to continuing. This ensures that an operating system crash or power failure will not corrupt the database. FULL synchronous is very safe, but it is also slower. FULL is the most commonly used synchronous setting when not in WAL mode.

NORMAL (1)

When synchronous is NORMAL (1), the SQLite database engine will still sync at the most critical moments, but less often than in FULL mode. There is a very small (though non-zero) chance that a power failure at just the wrong time could corrupt the database in NORMAL mode. But in practice, you are more likely to suffer a catastrophic disk failure or some other unrecoverable hardware fault. Many applications choose NORMAL when in WAL mode.

OFF (0)

With synchronous OFF (0), SQLite continues without syncing as soon as it has handed data off to the operating system. If the application running SQLite crashes, the data will be safe, but the database might become corrupted if the operating system crashes or the computer loses power before that data has been written to the disk surface. On the other hand, commits can be orders of magnitude faster with synchronous OFF.

In WAL mode when synchronous is NORMAL (1), the WAL file is synchronized before each checkpoint and the database file is synchronized after each completed checkpoint and the WAL file header is synchronized when a WAL file begins to be reused after a checkpoint, but no sync operations occur during most transactions. With synchronous=FULL in WAL mode, an additional sync operation of the WAL file happens after each transaction commit. The extra WAL sync following each transaction help ensure that transactions are durable across a power loss, but they do not aid in preserving consistency. If durability is not a concern, then synchronous=NORMAL is normally all one needs in WAL mode.

The default setting is usually synchronous=FULL. The SQLITE_EXTRA_DURABLE compile-time option changes the default to synchronous=EXTRA.

See also the fullfsync and checkpoint_fullfsync pragmas.

PRAGMA *schema.***table_info(***table-name***);**

This pragma returns one row for each column in the named table. Columns in the result set include the column name, data type, whether or not the column can be NULL, and the default value for the column. The "pk" column in the result set is zero for columns that are not part of the primary key, and is the index of the column in the primary key for columns that are part of the primary key.

The table named in the table_info pragma can also be a view.

PRAGMA temp_store; PRAGMA temp_store = *0* | *DEFAULT* | *1* | *FILE* | *2* | *MEMORY***;**

Query or change the setting of the "**temp_store**" parameter. When temp_store is DEFAULT (0), the compile-time C preprocessor macro SQLITE_TEMP_STORE is used to determine where temporary tables and indices are stored. When temp_store is MEMORY (2) temporary tables and indices are kept in as if they were pure in-memory databases memory. When temp_store is FILE (1) temporary tables and indices are stored in a file. The temp_store_directory pragma can be used to specify the directory containing temporary files when **FILE** is specified. When the temp_store setting is changed, all existing temporary tables, indices, triggers, and views are immediately deleted.

It is possible for the library compile-time C preprocessor symbol SQLITE_TEMP_STORE to override this pragma setting. The following table summarizes the interaction of the SQLITE_TEMP_STORE preprocessor macro and the temp_store pragma:

SQLITE_TEMP_STORE	PRAGMA temp_store	Storage used for TEMP tables and indices
0	*any*	file
1	0	file
1	1	file
1	2	memory
2	0	memory
2	1	file
2	2	memory
3	*any*	memory

PRAGMA temp_store_directory; PRAGMA temp_store_directory = '*directory-name***';**

Query or change the value of the sqlite3_temp_directory global variable, which many operating-system interface backends use to determine where to store temporary tables and indices.

When the temp_store_directory setting is changed, all existing temporary tables, indices, triggers, and viewers in the database connection that issued the pragma are immediately deleted. In practice, temp_store_directory should be set immediately after the first database connection for a process is opened. If the temp_store_directory is changed for one database connection while other database connections are open in the same process, then the behavior is undefined and probably undesirable.

Changing the temp_store_directory setting is not threadsafe. Never change the temp_store_directory setting if another thread within the application is running any SQLite interface at the same time. Doing so results in undefined behavior. Changing the temp_store_directory setting writes to the sqlite3_temp_directory global variable and that global variable is not protected by a mutex.

The value *directory-name* should be enclosed in single quotes. To revert the directory to the default, set the *directory-name* to an empty string, e.g., *PRAGMA temp_store_directory = ''*. An error is raised if *directory-name* is not found or is not writable.

The default directory for temporary files depends on the OS. Some OS interfaces may choose to ignore this variable and place temporary files in some other directory different from the directory specified here. In that sense, this pragma is only advisory.

This pragma is deprecated and exists for backwards compatibility only. New applications should avoid using this pragma. Older applications should discontinue use of this pragma at the earliest opportunity. This pragma may be omitted from the build when SQLite is compiled using SQLITE_OMIT_DEPRECATED.

PRAGMA threads; PRAGMA threads = *N*;

Query or change the value of the sqlite3_limit(db,SQLITE_LIMIT_WORKER_THREADS,...) limit for the current database connection. This limit sets an upper bound on the number of auxiliary threads that a prepared statement is allowed to launch to assist with a query. The default limit is 0 unless it is changed using the SQLITE_DEFAULT_WORKER_THREADS compile-time option. When the limit is zero, that means no auxiliary threads will be launched.

This pragma is a thin wrapper around the sqlite3_limit(db,SQLITE_LIMIT_WORKER_THREADS,...) interface.

PRAGMA vdbe_addoptrace = *boolean***;**

If SQLite has been compiled with the SQLITE_DEBUG compile-time option, then the vdbe_addoptrace pragma can be used to cause a complete VDBE opcodes to be displayed as they are created during code generation. This feature is used for debugging SQLite itself. See the VDBE documentation for more information.

This pragma is intended for use when debugging SQLite itself. It is only available when the SQLITE_DEBUG compile-time option is used.

PRAGMA vdbe_debug = *boolean***;**

If SQLite has been compiled with the SQLITE_DEBUG compile-time option, then the vdbe_debug pragma is a shorthand for three other debug-only pragmas: vdbe_addoptrace, vdbe_listing, and vdbe_trace. This feature is used for debugging SQLite itself. See the VDBE documentation for more information.

This pragma is intended for use when debugging SQLite itself. It is only available when the SQLITE_DEBUG compile-time option is used.

PRAGMA vdbe_listing = *boolean***;**

If SQLite has been compiled with the SQLITE_DEBUG compile-time option, then the vdbe_listing pragma can be used to cause a complete listing of the virtual machine opcodes to appear on standard output as each statement is evaluated. With listing is on, the entire content of a program is printed just prior to beginning execution. The statement executes normally after the listing is printed. This feature is used for debugging SQLite itself. See the VDBE documentation for more information.

This pragma is intended for use when debugging SQLite itself. It is only available when the SQLITE_DEBUG compile-time option is used.

PRAGMA vdbe_trace = *boolean***;**

If SQLite has been compiled with the SQLITE_DEBUG compile-time option, then the vdbe_trace pragma can be used to cause virtual machine opcodes to be printed on standard output as they are evaluated. This feature is used for debugging SQLite. See the VDBE documentation for more information.

This pragma is intended for use when debugging SQLite itself. It is only available when the SQLITE_DEBUG compile-time option is used.

PRAGMA wal_autocheckpoint; PRAGMA wal_autocheckpoint=*N*;

This pragma queries or sets the write-ahead log auto-checkpoint interval. When the write-ahead log is enabled (via the journal_mode pragma) a checkpoint will be run automatically whenever the write-ahead log equals or exceeds *N* pages in length. Setting the auto-checkpoint size to zero or a negative value turns auto-checkpointing off.

This pragma is a wrapper around the sqlite3_wal_autocheckpoint() C interface. All automatic checkpoints are PASSIVE.

Autocheckpointing is enabled by default with an interval of 1000 or SQLITE_DEFAULT_WAL_AUTOCHECKPOINT.

PRAGMA *schema.***wal_checkpoint; PRAGMA** *schema.***wal_checkpoint(PASSIVE); PRAGMA** *schema.***wal_checkpoint(FULL); PRAGMA** *schema.***wal_checkpoint(RESTART); PRAGMA** *schema.***wal_checkpoint(TRUNCATE);**

If the write-ahead log is enabled (via the journal_mode pragma), this pragma causes a checkpoint operation to run on database *database*, or on all attached databases if *database* is omitted. If write-ahead log mode is disabled, this pragma is a harmless no-op.

Invoking this pragma without an argument is equivalent to calling the sqlite3_wal_checkpoint() C interface.

Invoking this pragma with an argument is equivalent to calling the sqlite3_wal_checkpoint_v2() C interface with a 3rd parameter corresponding to the argument:

PASSIVE

Checkpoint as many frames as possible without waiting for any database readers or writers to finish. Sync the db file if all frames in the log are checkpointed. This mode is the same as calling the sqlite3_wal_checkpoint() C interface. The busy-handler callback is never invoked in this mode.

FULL

This mode blocks (invokes the busy-handler callback) until there is no database writer and all readers are reading from the most recent database snapshot. It then checkpoints all frames in the log file and syncs the database file. FULL blocks concurrent writers while it is running, but readers can proceed.

RESTART

This mode works the same way as FULL with the addition that after checkpointing the log file it blocks (calls the busy-handler callback) until all readers are finished with the log file. This ensures that the next client to write to the database file restarts the log file from the beginning. RESTART blocks concurrent writers while it is running, but allowed readers to proceed.

TRUNCATE

This mode works the same way as RESTART with the addition that the WAL file is truncated to zero bytes upon successful completion.

The wal_checkpoint pragma returns a single row with three integer columns. The first column is usually 0 but will be 1 if a RESTART or FULL or TRUNCATE checkpoint was blocked from completing, for example because another thread or process was actively using the database. In other words, the first column is 0 if the equivalent call to sqlite3_wal_checkpoint_v2() would have returned SQLITE_OK or 1 if the equivalent call would have returned SQLITE_BUSY. The second column is the number of modified pages that have been written to the write-ahead log file. The third column is the number of pages in the write-ahead log file that have been successfully moved back into the database file at the conclusion of the checkpoint. The second and third column are -1 if there is no write-ahead log, for example if this pragma is invoked on a database connection that is not in WAL mode.

PRAGMA writable_schema = ***boolean*****;**

When this pragma is on, the SQLITE_MASTER tables in which database can be changed using ordinary UPDATE, INSERT, and DELETE statements. Warning: misuse of this pragma can easily result in a corrupt database file.

SQL As Understood By SQLite

[Top]

Core Functions

The core functions shown below are available by default. Date & Time functions, aggregate functions, and JSON functions are documented separately. An application may define additional functions written in C and added to the database engine using the sqlite3_create_function() API.

abs(*X*)	The abs(X) function returns the absolute value of the numeric argument X. Abs(X) returns NULL if X is NULL. Abs(X) returns 0.0 if X is a string or blob that cannot be converted to a numeric value. If X is the integer -9223372036854775808 then abs(X) throws an integer overflow error since there is no equivalent positive 64-bit two complement value.
changes()	The changes() function returns the number of database rows that were changed or inserted or deleted by the most recently completed INSERT, DELETE, or UPDATE statement, exclusive of statements in lower-level triggers. The changes() SQL function is a wrapper around the sqlite3_changes() C/C++ function and hence follows the same rules for counting changes.
char(*X1*,*X2*,...,*XN*)	The char(X1,X2,...,XN) function returns a string composed of characters having the unicode code point values of integers X1 through XN, respectively.
coalesce(*X*,*Y*,...)	The coalesce() function returns a copy of its first non-NULL argument, or NULL if all

	arguments are NULL. Coalesce() must have at least 2 arguments.
glob(*X*,*Y*)	The glob(X,Y) function is equivalent to the expression "**Y GLOB X**". Note that the X and Y arguments are reversed in the glob() function relative to the infix GLOB operator. If the sqlite3_create_function() interface is used to override the glob(X,Y) function with an alternative implementation then the GLOB operator will invoke the alternative implementation.
ifnull(*X*,*Y*)	The ifnull() function returns a copy of its first non-NULL argument, or NULL if both arguments are NULL. Ifnull() must have exactly 2 arguments. The ifnull() function is equivalent to coalesce() with two arguments.
instr(*X*,*Y*)	The instr(X,Y) function finds the first occurrence of string Y within string X and returns the number of prior characters plus 1, or 0 if Y is nowhere found within X. Or, if X and Y are both BLOBs, then instr(X,Y) returns one more than the number bytes prior to the first occurrence of Y, or 0 if Y does not occur anywhere within X. If both arguments X and Y to instr(X,Y) are non-NULL and are not BLOBs then both are interpreted as strings. If either X or Y are NULL in instr(X,Y) then the result is NULL.
hex(*X*)	The hex() function interprets its argument as a BLOB and returns a string which is the upper-case hexadecimal rendering of the content of that blob.
last_insert_rowid()	The last_insert_rowid() function returns the ROWID of the last row insert from the database connection which invoked the function. The last_insert_rowid() SQL function is a wrapper around the

	sqlite3_last_insert_rowid() C/C++ interface function.	
length(*X*)	For a string value X, the length(X) function returns the number of characters (not bytes) in X prior to the first NUL character. Since SQLite strings do not normally contain NUL characters, the length(X) function will usually return the total number of characters in the string X. For a blob value X, length(X) returns the number of bytes in the blob. If X is NULL then length(X) is NULL. If X is numeric then length(X) returns the length of a string representation of X.	
like(*X*,*Y*)	like(*X*,*Y*,*Z*)	The like() function is used to implement tl **"Y LIKE X [ESCAPE** expression. If the optional ESCAPE cl is present, then the l function is invoked v three arguments. Otherwise, it is invol with two arguments Note that the X and parameters are reve in the like() function relative to the infix L operator. The sqlite3_create_funct interface can be use override the like() function and thereby change the operatio the LIKE operator. W overriding the like() function, it may be important to override both the two and thr argument versions c like() function. Otherwise, different code may be called implement the LIKE operator depending whether or not an ESCAPE clause wa specified.

likelihood(*X*,*Y*)	The likelihood(X,Y) function returns argument X unchanged. The value Y in likelihood(X,Y) must be a floating point constant between 0.0 and 1.0, inclusive. The likelihood(X) function is a no-op that the code generator optimizes away so that it consumes no CPU cycles during run-time (that is, during calls to sqlite3_step()). The purpose of the likelihood(X,Y) function is to provide a hint to the query planner that the argument X is a boolean that is true with a probability of approximately Y. The unlikely(X) function is short-hand for likelihood(X,0.0625). The likely(X) function is short-hand for likelihood(X,0.9375).
likely(*X*)	The likely(X) function returns the argument X unchanged. The likely(X) function is a no-op that the code generator optimizes away so that it consumes no CPU cycles at run-time (that is, during calls to sqlite3_step()). The purpose of the likely(X) function is to provide a hint to the query planner that the argument X is a boolean value that is usually true. The likely(X) function is equivalent to likelihood(X,0.9375). See also: unlikely(X).
load*extension(_X*)	The load_extension(X,Y) function loads SQLite extensions out of the shared library file named X using the entry point Y. The result of load_extension() is always a NULL. If Y is omitted then the default entry point name is used. The load_extension() function raises an exception if the extension fails to load or initialize correctly.The load_extension() function will fail if the extension attempts to modify or delete an SQL function or collating sequence. The extension can add new functions

load*extension(_X*,*Y*)	or collating sequences, but cannot modify or delete existing functions or collating sequences because those functions and/or collating sequences might be used elsewhere in the currently running SQL statement. To load an extension that changes or deletes functions or collating sequences, use the sqlite3_load_extension() C-language API.For security reasons, extension loaded is turned off by default and must be enabled by a prior call to sqlite3_enable_load_extension().
lower(*X*)	The lower(X) function returns a copy of string X with all ASCII characters converted to lower case. The default built-in lower() function works for ASCII characters only. To do case conversions on non-ASCII characters, load the ICU extension.
ltrim(*X*) ltrim(*X*,*Y*)	The ltrim(X,Y) function returns a string formed by removing any and all characters that appear in Y from the left side of X. If the Y argument is omitted, ltrim(X) removes spaces from the left side of X.
max(*X*,*Y*,...)	The multi-argument max() function returns the argument with the maximum value, or return NULL if any argument is NULL. The multi-argument max() function searches its arguments from left to right for an argument that defines a collating function and uses that collating function for all string comparisons. If none of the arguments to max() define a collating function, then the BINARY collating function is used. Note that **max()** is a simple function when it has 2 or more arguments but operates as an aggregate function if given only a single argument.

min(*X*,*Y*,...)	The multi-argument min() function returns the argument with the minimum value. The multi-argument min() function searches its arguments from left to right for an argument that defines a collating function and uses that collating function for all string comparisons. If none of the arguments to min() define a collating function, then the BINARY collating function is used. Note that **min()** is a simple function when it has 2 or more arguments but operates as an aggregate function if given only a single argument.
nullif(*X*,*Y*)	The nullif(X,Y) function returns its first argument if the arguments are different and NULL if the arguments are the same. The nullif(X,Y) function searches its arguments from left to right for an argument that defines a collating function and uses that collating function for all string comparisons. If neither argument to nullif() defines a collating function then the BINARY is used.
printf(*FORMAT*,...)	The printf(FORMAT,...) SQL function works like the sqlite3_mprintf() C-language function and the printf() function from the standard C library. The first argument is a format string that specifies how to construct the output string using values taken from subsequent arguments. If the FORMAT argument is missing or NULL then the result is NULL. The %n format is silently ignored and does not consume an argument. The %p format is an alias for %X. The %z format is interchangeable with %s. If there are too few arguments in the argument list, missing arguments are assumed to have a NULL value, which is translated into 0 or 0.0 for numeric formats or an

	empty string for %s.
quote(*X*)	The quote(X) function returns the text of an SQL literal which is the value of its argument suitable for inclusion into an SQL statement. Strings are surrounded by single-quotes with escapes on interior quotes as needed. BLOBs are encoded as hexadecimal literals. Strings with embedded NUL characters cannot be represented as string literals in SQL and hence the returned string literal is truncated prior to the first NUL.
random()	The random() function returns a pseudo-random integer between -9223372036854775808 and +9223372036854775807.

randomblob(*N*)	The randomblob(N) function return an N-byte blob containing pseudo-random bytes. If N is less than 1 then a 1-byte random blob is returned.Hint: applications can generate globally unique identifiers using this function together with hex() and/or lower() like this:
	`hex(randomblob(16))`
	`lower(hex(randomblob(16)))`

replace(*X*,*Y*,*Z*)	The replace(X,Y,Z) function returns a string formed by substituting string Z for every occurrence of string Y in string X. The BINARY collating sequence is used for comparisons. If Y is an empty string then return X unchanged. If Z is not initially a string, it is cast to a UTF-8 string prior to processing.
round(*X*) round(*X*,*Y*)	The round(X,Y) function returns a floating-point value X rounded to Y digits to the right of the decimal point. If the Y argument is omitted, it is assumed to be 0.
	The rtrim(X,Y) function returns a string formed by removing any and all characters that appear in

rtrim(*X*) rtrim(*X*,*Y*)	Y from the right side of X. If the Y argument is omitted, rtrim(X) removes spaces from the right side of X.
soundex(*X*)	The soundex(X) function returns a string that is the soundex encoding of the string X. The string "?000" is returned if the argument is NULL or contains no ASCII alphabetic characters. This function is omitted from SQLite by default. It is only available if the SQLITE_SOUNDEX compile-time option is used when SQLite is built.
sqlite*compileoption_get(_N*)	The sqlite_compileoption_get() SQL function is a wrapper around the sqlite3_compileoption_get() C/C++ function. This routine returns the N-th compile-time option used to build SQLite or NULL if N is out of range. See also the compile_options pragma.
sqlite*compileoption_used(_X*)	The sqlite_compileoption_used() SQL function is a wrapper around the sqlite3_compileoption_used() C/C++ function. When the argument X to sqlite_compileoption_used(X) is a string which is the name of a compile-time option, this routine returns true (1) or false (0) depending on whether or not that option was used during the build.
sqlite_source_id()	The sqlite_source_id() function returns a string that identifies the specific version of the source code that was used to build the SQLite library. The string returned by sqlite_source_id() is the date and time that the source code was checked in followed by the SHA1 hash for that check-in. This function is an SQL wrapper around the sqlite3_sourceid() C interface.

sqlite_version()	The sqlite_version() function returns the version string for the SQLite library that is running. This function is an SQL wrapper around the sqlite3_libversion() C-interface.
substr(*X*,*Y*,*Z*) substr(*X*,*Y*)	The substr(X,Y,Z) function returns a substring of input string X that begins with the Y-th character and which is Z characters long. If Z is omitted then substr(X,Y) returns all characters through the end of the string X beginning with the Y-th. The left-most character of X is number 1. If Y is negative then the first character of the substring is found by counting from the right rather than the left. If Z is negative then the abs(Z) characters preceding the Y-th character are returned. If X is a string then characters indices refer to actual UTF-8 characters. If X is a BLOB then the indices refer to bytes.
total_changes()	The total_changes() function returns the number of row changes caused by INSERT, UPDATE or DELETE statements since the current database connection was opened. This function is a wrapper around the sqlite3_total_changes() C/C++ interface.
trim(*X*) trim(*X*,*Y*)	The trim(X,Y) function returns a string formed by removing any and all characters that appear in Y from both ends of X. If the Y argument is omitted, trim(X) removes spaces from both ends of X.
typeof(*X*)	The typeof(X) function returns a string that indicates the datatype of the expression X: "null", "integer", "real", "text", or "blob".
	The unlikely(X) function returns the argument X unchanged. The unlikely(X) function is a no-op that the code generator

unlikely(*X*)	optimizes away so that it consumes no CPU cycles at run-time (that is, during calls to sqlite3_step()). The purpose of the unlikely(X) function is to provide a hint to the query planner that the argument X is a boolean value that is usually not true. The unlikely(X) function is equivalent to likelihood(X, 0.0625).
unicode(*X*)	The unicode(X) function returns the numeric unicode code point corresponding to the first character of the string X. If the argument to unicode(X) is not a string then the result is undefined.
upper(*X*)	The upper(X) function returns a copy of input string X in which all lower-case ASCII characters are converted to their upper-case equivalent.
zeroblob(*N*)	The zeroblob(N) function returns a BLOB consisting of N bytes of 0x00. SQLite manages these zeroblobs very efficiently. Zeroblobs can be used to reserve space for a BLOB that is later written using incremental BLOB I/O. This SQL function is implemented using the sqlite3_result_zeroblob() routine from the C/C++ interface.

SQL As Understood By SQLite

[Top]

Aggregate Functions

The aggregate functions shown below are available by default. Additional aggregate functions written in C may be added using the sqlite3_create_function() API.

In any aggregate function that takes a single argument, that argument can be preceded by the keyword DISTINCT. In such cases, duplicate elements are filtered before being passed into the aggregate function. For example, the function "count(distinct X)" will return the number of distinct values of column X instead of the total number of non-null values in column X.

avg(*X*)	The avg() function returns the average value of all non-NULL *X* within a group. String and BLOB values that do not look like numbers are interpreted as 0. The result of avg() is always a floating point value as long as at there is at least one non-NULL input even if all inputs are integers. The result of avg() is NULL if and only if there are no non-NULL inputs.
count(*X*) count(*)	The count(X) function returns a count of the number of times that *X* is not NULL in a group. The count(*) function (with no arguments) returns the total number of rows in the group.
group*concat(_X*) group*concat(_X*,*Y*)	The group*concat() function returns a string which is the concatenation of all non-NULL values of _X*. If parameter *Y* is present then it is used as the separator between instances of *X*. A comma (",") is used as the separator if *Y* is omitted. The order of the concatenated elements is arbitrary.
max(*X*)	The max() aggregate function returns the maximum value of all values in the group. The maximum value is the value that would be returned last in an ORDER BY on the same column. Aggregate max() returns NULL if and only if there are no non-NULL values in the group.
min(*X*)	The min() aggregate function returns the minimum non-NULL value of all values in the group. The minimum value is the first non-NULL value that would appear in an ORDER BY of the column. Aggregate min() returns NULL if and only if there are no non-NULL values in the group.
sum(*X*) total(*X*)	The sum() and total() aggregate functions return sum of all non-NULL values in the group. If there are no non-NULL input rows then sum() returns NULL but total() returns 0.0. NULL is not normally a helpful result for the sum of no rows but the SQL standard requires it and most other SQL database engines implement sum() that way so SQLite does it in the same way in order to be compatible. The non-standard total() function is provided as a convenient way to work around this design problem in the SQL language.The result of total() is always a floating point value. The result of sum() is an integer value if all non-NULL inputs are integers. If any input to sum() is neither an integer or a NULL then sum() returns a floating point value which might be an approximation to the true sum.Sum() will throw an "integer overflow" exception if all inputs are integers or NULL and an integer overflow occurs at any point during the computation. Total() never throws an integer overflow.

SQL As Understood By SQLite

[Top]

Date And Time Functions

SQLite supports five date and time functions as follows:

1. **date(***timestring, modifier, modifier, ...***)**
2. **time(***timestring, modifier, modifier, ...***)**
3. **datetime(***timestring, modifier, modifier, ...***)**
4. **julianday(***timestring, modifier, modifier, ...***)**
5. **strftime(***format, timestring, modifier, modifier, ...***)**

All five date and time functions take a time string as an argument. The time string is followed by zero or more modifiers. The strftime() function also takes a format string as its first argument.

The date and time functions use a subset of IS0-8601 date and time formats. The date() function returns the date in this format: YYYY-MM-DD. The time() function returns the time as HH:MM:SS. The datetime() function returns "YYYY-MM-DD HH:MM:SS". The julianday() function returns the Julian day - the number of days since noon in Greenwich on November 24, 4714 B.C. (Proleptic Gregorian calendar). The strftime() routine returns the date formatted according to the format string specified as the first argument. The format string supports the most common substitutions found in the strftime() function from the standard C library plus two new substitutions, %f and %J. The following is a complete list of valid strftime() substitutions:

%d	day of month: 00
%f	fractional seconds: SS.SSS
%H	hour: 00-24
%j	day of year: 001-366
%J	Julian day number
%m	month: 01-12
%M	minute: 00-59
%s	seconds since 1970-01-01
%S	seconds: 00-59
%w	day of week 0-6 with Sunday==0
%W	week of year: 00-53
%Y	year: 0000-9999
%%	%

Notice that all other date and time functions can be expressed in terms of strftime():

Function	Equivalent strftime()
date(...)	strftime('%Y-%m-%d', ...)
time(...)	strftime('%H:%M:%S', ...)
datetime(...)	strftime('%Y-%m-%d %H:%M:%S', ...)
julianday(...)	strftime('%J', ...)

The only reasons for providing functions other than strftime() is for convenience and for efficiency.

Time Strings

A time string can be in any of the following formats:

1. *YYYY-MM-DD*
2. *YYYY-MM-DD HH:MM*
3. *YYYY-MM-DD HH:MM:SS*
4. *YYYY-MM-DD HH:MM:SS.SSS*
5. *YYYY-MM-DD*__T__*HH:MM*
6. *YYYY-MM-DD*__T__*HH:MM:SS*
7. *YYYY-MM-DD*__T__*HH:MM:SS.SSS*
8. *HH:MM*

9. *HH:MM:SS*
10. *HH:MM:SS.SSS*
11. **now**
12. *DDDDDDDDDD*

In formats 5 through 7, the "T" is a literal character separating the date and the time, as required by ISO-8601. Formats 8 through 10 that specify only a time assume a date of 2000-01-01. Format 11, the string 'now', is converted into the current date and time as obtained from the xCurrentTime method of the sqlite3_vfs object in use. The 'now' argument to date and time functions always returns exactly the same value for multiple invocations within the same sqlite3_step() call. Universal Coordinated Time (UTC) is used. Format 12 is the Julian day number expressed as a floating point value.

Formats 2 through 10 may be optionally followed by a timezone indicator of the form "*[+-]HH:MM*" or just "*Z*". The date and time functions use UTC or "zulu" time internally, and so the "Z" suffix is a no-op. Any non-zero "HH:MM" suffix is subtracted from the indicated date and time in order to compute zulu time. For example, all of the following time strings are equivalent:

> 2013-10-07 08:23:19.120 2013-10-07T08:23:19.120Z 2013-10-07 04:23:19.120-04:00 2456572.84952685

In formats 4, 7, and 10, the fractional seconds value SS.SSS can have one or more digits following the decimal point. Exactly three digits are shown in the examples because only the first three digits are significant to the result, but the input string can have fewer or more than three digits and the date/time functions will still operate correctly. Similarly, format 12 is shown with 10 significant digits, but the date/time functions will really accept as many or as few digits as are necessary to represent the Julian day number.

Modifiers

The time string can be followed by zero or more modifiers that alter date and/or time. Each modifier is a transformation that is applied to the time value to its left. Modifiers are applied from left to right; order is important. The available modifiers are as follows.

1. NNN days
2. NNN hours
3. NNN minutes
4. NNN.NNNN seconds
5. NNN months
6. NNN years
7. start of month
8. start of year

9. start of day
10. weekday N
11. unixepoch
12. localtime
13. utc

The first six modifiers (1 through 6) simply add the specified amount of time to the date and time specified by the preceding timestring and modifiers. The 's' character at the end of the modifier names is optional. Note that "±NNN months" works by rendering the original date into the YYYY-MM-DD format, adding the ±NNN to the MM month value, then normalizing the result. Thus, for example, the data 2001-03-31 modified by '+1 month' initially yields 2001-04-31, but April only has 30 days so the date is normalized to 2001-05-01. A similar effect occurs when the original date is February 29 of a leapyear and the modifier is ±N years where N is not a multiple of four.

The "start of" modifiers (7 through 9) shift the date backwards to the beginning of the current month, year or day.

The "weekday" modifier advances the date forward to the next date where the weekday number is N. Sunday is 0, Monday is 1, and so forth.

The "unixepoch" modifier (11) only works if it immediately follows a timestring in the DDDDDDDDDD format. This modifier causes the DDDDDDDDDD to be interpreted not as a Julian day number as it normally would be, but as Unix Time - the number of seconds since 1970. If the "unixepoch" modifier does not follow a timestring of the form DDDDDDDDDD which expresses the number of seconds since 1970 or if other modifiers separate the "unixepoch" modifier from prior DDDDDDDDDD then the behavior is undefined. Due to precision limitations imposed by the implementations use of 64-bit integers, the "unixepoch" modifier only works for dates between 0000-01-01 00:00:00 and 5352-11-01 10:52:47 (unix times of -62167219200 through 10675199167).

The "localtime" modifier (12) assumes the time string to its left is in Universal Coordinated Time (UTC) and adjusts the time string so that it displays localtime. If "localtime" follows a time that is not UTC, then the behavior is undefined. The "utc" modifier is the opposite of "localtime". "utc" assumes that the string to its left is in the local timezone and adjusts that string to be in UTC. If the prior string is not in localtime, then the result of "utc" is undefined.

Examples

Compute the current date.

```
SELECT date('now');
```

Compute the last day of the current month.

```
SELECT date('now','start of month','+1 month','-1 day');
```

Compute the date and time given a unix timestamp 1092941466.

```
SELECT datetime(1092941466, 'unixepoch');
```

Compute the date and time given a unix timestamp 1092941466, and compensate for your local timezone.

```
SELECT datetime(1092941466, 'unixepoch', 'localtime');
```

Compute the current unix timestamp.

```
SELECT strftime('%s','now');
```

Compute the number of days since the signing of the US Declaration of Independence.

```
SELECT julianday('now') - julianday('1776-07-04');
```

Compute the number of seconds since a particular moment in 2004:

```
SELECT strftime('%s','now') - strftime('%s','2004-01-01 02:34:56');
```

Compute the date of the first Tuesday in October for the current year.

```
SELECT date('now','start of year','+9 months','weekday 2');
```

Compute the time since the unix epoch in seconds (like strftime('%s','now') except includes fractional part):

```
SELECT (julianday('now') - 2440587.5)*86400.0;
```

Caveats And Bugs

The computation of local time depends heavily on the whim of politicians and is thus difficult to get correct for all locales. In this implementation, the standard C library function localtime_r() is used to assist in the calculation of local time. The localtime_r() C function normally only works for years between 1970 and 2037. For dates outside this range, SQLite attempts to map the year into an equivalent year within this range, do the calculation, then map the year back.

These functions only work for dates between 0000-01-01 00:00:00 and 9999-12-31 23:59:59 (julidan day numbers 1721059.5 through 5373484.5). For dates outside that range, the results of these functions are undefined.

Non-Vista Windows platforms only support one set of DST rules. Vista only supports two. Therefore, on these platforms, historical DST calculations will be incorrect. For example, in the US, in 2007 the DST rules changed. Non-Vista Windows platforms apply the new 2007

DST rules to all previous years as well. Vista does somewhat better getting results correct back to 1986, when the rules were also changed.

All internal computations assume the Gregorian calendar system. It is also assumed that every day is exactly 86400 seconds in duration.

The JSON1 Extension

The **json1** extension is a loadable extension that implements thirteen application-defined SQL functions and two table-valued functions that are useful for managing JSON content stored in an SQLite database. These are the scalar SQL functions implemented by json1:

1.	json(*json*)	Validate and minify a JSON string
2.	json_array(*value1,value2,...*)	Return a JSON array holding the function arguments.

3.	json_array_length(*json*)
	json_array_length(*json,path*)
		Return the number of elements in the JSON array identified by the arguments.

4.	json_extract(*json,path,...*)	Extract values or subcomponents from a JSON string.
5.	json_insert(*json,path,value,...*)	Insert values into a JSON string without overwriting existing values.
6.	json_object(*label1,value1,...*)	Construct and return a new JSON object based on the arguments.
7.	json_remove(*json,path,...*)	Remove the specified values from a JSON string.
8.	json_replace(*json,path,value,...*)	Update existing values within a JSON string.
9.	json_set(*json,path,value,...*)	Insert or replace values in a JSON string. Overwrite existing elements or create new entries in the JSON string for elements that do not previously exist.

10.	json_type(*json*)
	json_type(*json,path*)
		Return the type of a JSON string or subcomponent.

11.	json_valid(*json*)	Return true (1) if the input text is a valid JSON string
	There are two aggregate SQL functions:	
12.	json_group_array(*value*)	Return a JSON array composed of all *value* elements in the aggregation.
13.	json_group_object(name,*value*)	Return a JSON object composed of all *name* and *value* pairs in the aggregation.
	The table-valued functions implemented by this routine are:	

14.	json_each(*json*)
	json_each(*json,path*)
		Return one row describing each element in an array or object at the top-level or at "path" within the input JSON.

15.	json_tree(*json*)
	json_tree(*json,path*)
		Walk the JSON recursively starting at the top-level or at the specified "path" and return one row for each element.

1.0 Compiling the JSON1 Extension

The loadable extensions documentation contains instructions on how to compile loadable extensions as shared libraries. The techniques described there work fine for the json1 module.

The json1 source code is included with the SQLite amalgamation, though it is turned off by default. Add the -DSQLITE_ENABLE_JSON1 compile-time option to enable the json1 extension that is built into the amalgamation.

2.0 Interface Overview

The json1 extension (currently) stores JSON as ordinary text.

Backwards compatibility constraints mean that SQLite is only able to store values that are NULL, integers, floating-point numbers, text, and BLOBs. It is not possible to add a sixth "JSON" type.

The json1 extension does not (currently) support a binary encoding of JSON. Experiments have so far been unable to find a binary encoding that is significantly smaller or faster than a plain text encoding. (The present implementation parses JSON text at over 300 MB/s.) All json1 functions currently throw an error if any of their arguments are BLOBs because BLOBs are reserved for a future enhancement in which BLOBs will store the binary encoding for JSON.

The "1" at the end of the name for the json1 extension is deliberate. The designers anticipate that there will be future incompatible JSON extensions building upon the lessons learned from json1. Once sufficient experience is gained, some kind of JSON extension might be folded into the SQLite core. For now, JSON support remains an extension.

2.1 JSON arguments

For functions that accept JSON as their first argument, that argument can be a JSON object, array, number, string, or null. SQLite numeric values and NULL values are interpreted as JSON numbers and nulls, respectively. SQLite text values can be understood as JSON objects, arrays, or strings. If an SQLite text value that is not a well-formed JSON object, array, or string is passed into json1 function, that function will usually throw an error. (An exception is the json_valid(X) function which returns 1 if X is well-formed JSON and 0 if it is not.)

For the purposes of determining validity, leading and trailing whitespace on JSON inputs is ignored. Interior whitespace is also ignored, in accordance with the JSON spec. These routines accept exactly the rfc-7159 JSON syntax — no more and no less.

2.2 PATH arguments

For functions that accept PATH arguments, that PATH must be well-formed or else the function will throw an error. A well-formed PATH is a text value that begins with exactly one '$' character followed by zero or more instances of ".*objectlabel*" or "[*arrayindex*]".

2.3 VALUE arguments

For functions that accept "*value*" arguments (also shown as "*value1*" and "*value2*"), those arguments is usually understood to be a literal strings that are quoted and becomes JSON string values in the result. Even if the input *value* strings look like well-formed JSON, they are still interpreted as literal strings in the result.

However, if a *value* argument come directly from the result of another json1 function, then the argument is understood to be actual JSON and the complete JSON is inserted rather than a quoted string.

For example, in the following call to json*object(), the _value* argument looks like a well-formed JSON array. However, because it is just ordinary SQL text, it is interpreted as a literal string and added to the result as a quoted string:

> | json_object('ex','[52,3.14159]') | → | '{"ex":"[52,3.14159]"}' |

But if the *value* argument in the outer json_object() call is the result of another json1 function like json() or json_array(), then the value is understood to be actual JSON and is inserted as such:

> | json_object('ex',json('[52,3.14159]')) | → | '{"ex":[52,3.14159]}' | | json_object('ex',json_array(52,3.14159)) | → | '{"ex":[52,3.14159]}' |

To be clear: "*json*" arguments are always interpreted as JSON regardless of where the value for that argument comes from. But "*value*" arguments are only interpreted as JSON if those arguments come directly from another json1 function.

2.4 Compatibility

The json1 extension uses the sqlite3_value_subtype() and sqlite3_result_subtype() interfaces that were introduced with SQLite version 3.9.0. Therefore the json1 extension will not work in earlier versions of SQLite.

3.0 Function Details

The following sections provide additional detail on the operation of the various functions that are part of the json1 extension.

3.1 The json() function

The json(X) function verifies that its argument X is a valid JSON string and returns a minified version of that JSON string (with all unnecessary whitespace removed). If X is not a well-formed JSON string, then this routine throws an error.

If the argument X to json(X) contains JSON objects with duplicate labels, then it is undefined whether or not the duplicates are preserved. The current implementation preserves duplicates. However, future enhancements to this routine may choose to silently remove duplicates.

Example:

> | json(' { "this" : "is", "a": ["test"] } ') | → | '{"this":"is","a":["test"]}' |

3.2 The json_array() function

The json_array() SQL function accepts zero or more arguments and returns a well-formed JSON array that is composed from those arguments. If any argument to json_array() is a BLOB then an error is thrown.

An argument with SQL type TEXT it is normally converted into a quoted JSON string. However, if the argument is the output from another json1 function, then it is stored as JSON. This allows calls to json_array() and json_object() to be nested. The json() function can also be used to force strings to be recognized as JSON.

Examples:

> | json_array(1,2,'3',4) | → | '[1,2,"3",4]' | | json_array('[1,2]') | → | '["[1,2]"]' | | json_array(json_array(1,2)) | → | '[[1,2]]' | | json_array(1,null,'3','[4,5]','{"six":7.7}') | → | '[1,null,"3","[4,5]","{\"six\":7.7}"]' | | json_array(1,null,'3',json('[4,5]'),json('{"six":7.7}')) | → | '[1,null,"3",[4,5],{"six":7.7}]' |

3.3 The json_array_length() function

The json_array_length(X) function returns the number of elements in the JSON array X, or 0 if X is some kind of JSON value other than an array. The json_array_length(X,P) locates the array at path P within X and returns the length of that array, or 0 if path P locates a element or X other than a JSON array, and NULL if path P does not locate any element of X. Errors are thrown if either X is not well-formed JSON or if P is not a well-formed path.

Examples:

> | json_array_length('[1,2,3,4]') | → | 4 | | json_array_length('[1,2,3,4]', '$') | → | 4 | | json_array_length('[1,2,3,4]', '$[2]') | → | 0 | | json_array_length('{"one":[1,2,3]}') | → | 0 | | json_array_length('{"one":[1,2,3]}', '$.one') | → | 3 | | json_array_length('{"one":[1,2,3]}', '$.two') | → | NULL |

3.4 The json_extract() function

The json_extract(X,P1,P2,...) extracts and returns one or more values from the well-formed JSON at X. If only a single path P1 is provided, then the SQL datatype of the result is NULL for a JSON null, INTEGER or REAL for a JSON numeric value, an INTEGER zero for a JSON false value, an INTEGER one for a JSON true value, the dequoted text for a JSON string value, and a text representation for JSON object and array values. If there are multiple path arguments (P1, P2, and so forth) then this routine returns SQLite text which is a well-formed JSON array holding the various values.

Examples:

> | json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$') | → | '{"a":2,"c":[4,5,{"f":7}]}' | | json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c') | → | '[4,5,{"f":7}]' | | json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c[2]') | → | '{"f":7}' | | json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c[2].f') | → | 7 | | json_extract('{"a":2,"c":[4,5],"f":7}','$.c','$.a') | → | '[[4,5],2]' | | json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.x') | → | NULL | | json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.x', '$.a') | → | '[null,2]' |

3.5 The json_insert(), json_replace, and json_set() functions

The json_insert(), json_replace, and json_set() functions all take a single JSON value as their first argument followed by zero or more pairs of path and value arguments, and return a new JSON string formed by updating the input JSON by the path/value pairs. The functions differ only in how they deal with creating new values and overwriting preexisting values.

Function	Overwrite if already exists?	Create if does not exist?
json_insert()	No	Yes
json_replace()	Yes	No
json_set()	Yes	Yes

The json_insert(), json_replace(), and json_set() functions always take an odd number of arguments. The first argument is always the original JSON to be edited. Subsequent arguments occur in pairs with the first element of each pair being a path and the second element being the value to insert or replace or set on that path.

Edits occurs sequentially from left to right. Changes caused by prior edits can affect the path search for subsequent edits.

If the value of a path/value pair is an SQLite TEXT value, then it is normally inserted as a quoted JSON string, even if the string looks like valid JSON. However, if the value is the result of another json1 function (such as json() or json_array() or json_object()) then it is interpreted as JSON and is inserted as JSON retaining all of its substructure.

These routines throw an error if the first JSON argument is not well-formed or if any PATH argument is not well-formed or if any argument is a BLOB.

Examples:

> | json_insert('{"a":2,"c":4}', '$.a', 99) | → | '{"a":2,"c":4}' | | json_insert('{"a":2,"c":4}', '$.e', 99) | → | '{"a":2,"c":4,"e":99}' | | json_replace('{"a":2,"c":4}', '$.a', 99) | → | '{"a":99,"c":4}' | | json_replace('{"a":2,"c":4}', '$.e', 99) | → | '{"a":2,"c":4}' | | json_set('{"a":2,"c":4}', '$.a', 99) | → | '{"a":99,"c":4}' | | json_set('{"a":2,"c":4}', '$.e', 99) | → | '{"a":2,"c":4,"e":99}' | | json_set('{"a":2,"c":4}', '$.c', '[97,96]') | → | '{"a":2,"c":"[97,96]"}' | | json_set('{"a":2,"c":4}', '$.c', json('[97,96]')) | → | '{"a":2,"c":[97,96]}' | | json_set('{"a":2,"c":4}', '$.c', json_array(97,96)) | → | '{"a":2,"c":[97,96]}' |

3.6 The json_object() function

The json_object() SQL function accepts zero or more pairs of arguments and returns a well-formed JSON object that is composed from those arguments. The first argument of each pair is the label and the second argument of each pair is the value. If any argument to json_object() is a BLOB then an error is thrown.

The json_object() function currently allows duplicate labels without complaint, though this might change in a future enhancement.

An argument with SQL type TEXT it is normally converted into a quoted JSON string even if the input text is well-formed JSON. However, if the argument is the direct result from another json1 function, then it is treated as JSON and all of its JSON type information and substructure is preserved. This allows calls to json_object() and json_array() to be nested. The json() function can also be used to force strings to be recognized as JSON.

Examples:

> | json_object('a',2,'c',4) | → | '{"a":2,"c":4}' | | json_object('a',2,'c','{e:5}') | → | '{"a":2,"c":"{e:5}"}' | | json_object('a',2,'c',json_object('e',5)) | → | '{"a":2,"c":{"e":5}}' |

3.7 The json_remove() function

The json_remove(X,P,...) function takes a single JSON value as its first argument followed by zero or more path arguments. The json_remove(X,P,...) function returns a new JSON value that is the X with all the elements identified by path arguments removed. Paths that select elements not found in X are silently ignored.

Removals occurs sequentially from left to right. Changes caused by prior removals can affect the path search for subsequent arguments.

If the json_remove(X) function is called with no path arguments, then it returns the input X reformatted, with excess whitespace removed.

The json_remove() function throws an error if the first argument is not well-formed JSON or if any later argument is not a well-formed path, or if any argument is a BLOB.

Examples:

> | json_remove('[0,1,2,3,4]','$[2]') | → | '[0,1,3,4]' | | json_remove('[0,1,2,3,4]','$[2]','$[0]') | → | '[1,3,4]' | | json_remove('[0,1,2,3,4]','$[0]','$[2]') | → | '[1,2,4]' | | json_remove('{"x":25,"y":42}') | → | '{"x":25,"y":42}' | | json_remove('{"x":25,"y":42}','$.z') | → | '{"x":25,"y":42}' | | json_remove('{"x":25,"y":42}','$.y') | → | '{"x":25}' | | json_remove('{"x":25,"y":42}','$') | → | NULL |

3.7 The json_type() function

The json_type(X) function returns the "type" of the outermost element of X. The json_type(X,P) function returns the "type" of the element in X that is selected by path P. The "type" returned by json_type() is on of the following an SQL text values: 'null', 'true', 'false', 'integer', 'real', 'text', 'array', or 'object'. If the path P in json_type(X,P) selects a element that does not exist in X, then this function returns NULL.

The json_type() function throws an error if any of its arguments are not well-formed or is a BLOB.

Examples:

> | json_type('{"a":[2,3.5,true,false,null,"x"]}') | → | 'object' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$') | → | 'object' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a') | → | 'array' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[0]') | → | 'integer' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[1]') | → | 'real' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[2]') | → | 'true' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[3]') | → | 'false' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[4]') | → | 'null' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[5]') | → | 'text' | | json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[6]') | → | NULL |

3.9 The json_valid() function

The json_valid(X) function return 1 if the argument X is well-formed JSON and return 0 if the argument X is not well-formed JSON.

Examples:

> | json_valid('{"x":35}') | → | 1 | | json_valid('{"x":35') | → | 0 |

3.10 The json_group_array() and json_group_object() aggregate SQL functions

The json_group_array(X) function is an aggregate SQL function that returns a JSON array comprised of all X values in the aggregation. Similarly, the json_group_object(NAME,VALUE) function returns a JSON object comprised of all NAME/VALUE pairs in the aggregation.

3.11 The json_each() and json_tree() table-valued functions

The json_each(X) and json_tree(X) table-valued functions walk the JSON value provided as their first argument and return one row for each element. The json_each(X) function only walks the immediate children of the top-level array or object or or just the top-level element itself if the top-level element is a primitive value. The json_tree(X) function recursively walks through the JSON substructure starting with the top-level element.

The json_each(X,P) and json_tree(X,P) functions work just like their one-argument counterparts except that they treat the element identified by path P as the top-level element.

The schema for the table returned by json_each() and json_tree() is as follows:

```
CREATE TABLE json_tree(
    key ANY,             -- key for current element relative to its parent
    value ANY,           -- value for the current element
    type TEXT,           -- 'object','array','string','integer', etc.
    atom ANY,            -- value for primitive types, null for array & object
    id INTEGER           -- integer ID for this element
    parent INTEGER,      -- integer ID for the parent of this element
    fullkey TEXT,        -- full path describing the current element
    path TEXT,           -- path to the container of the current row
    json JSON HIDDEN,    -- 1st input parameter: the raw JSON
    root TEXT HIDDEN     -- 2nd input parameter: the PATH at which to start
);
```

The "key" column is the integer array index for elements of a JSON array and the text label for elements of a JSON object. The key column is NULL in all other cases.

The "atom" column is the SQL value corresponding to primitive elements - elements other than JSON arrays and objects. The "atom" column is NULL for a JSON array or object. The "value" column is the same as the "atom" column for primitive JSON elements but takes on the text JSON value for arrays and objects.

The "type" column is an SQL text value taken from ('null', 'true', 'false', 'integer', 'real', 'text', 'array', 'object') according to the type of the current JSON element.

The "id" column is an integer that identifies a specific JSON element within the complete JSON string. The "id" integer is an internal housekeeping number, the computation of which might change in future releases. The only guarantee is that the "id" column will be different for every row.

The "parent" column is always NULL for json_each(). For json_tree(), the "parent" column is the "id" integer for the parent of the current element, or NULL for the top-level JSON element or the element identified by the root path in the second argument.

The "fullkey" column is a text path that uniquely identifies the current row element within the original JSON string. The complete key to the true top-level element is returned even if an alternative starting point is provided by the "root" argument.

The "path" column is the path to the array or object container the holds the current row, or the path to the current row in the case where the iteration starts on a primitive type and thus only provides a single row of output.

3.11.1 Examples using json_each() and json_tree()

Suppose the table "CREATE TABLE user(name,phone)" stores zero or more phone numbers as a JSON array object in the user.phone field. To find all users who have any phone number with a 704 area code:

```
SELECT DISTINCT user.name
  FROM user, json_each(user.phone)
 WHERE json_each.value LIKE '704-%';
```

Now suppose the user.phone field contains plain text if the user has only a single phone number and a JSON array if the user has multiple phone numbers. The same question is posed: "Which users have a phone number in the 704 area code?" But now the json_each() function can only be called for those users that have two or more phone numbers since json_each() requires well-formed JSON as its first argument:

```
SELECT name FROM user WHERE phone LIKE '704-%'
UNION
SELECT user.name
  FROM user, json_each(user.phone)
 WHERE json_valid(user.phone)
   AND json_each.value LIKE '704-%';
```

Consider a different database with "CREATE TABLE big(json JSON)". To see a complete line-by-line decomposition of the data:

```
SELECT big.rowid, fullkey, value
  FROM big, json_tree(big.json)
 WHERE json_tree.type NOT IN ('object','array');
```

In the previous, the "type NOT IN ('object','array')" term of the WHERE clause suppresses containers and only lets through leaf elements. The same effect could be achieved this way:

```
SELECT big.rowid, fullkey, atom
  FROM big, json_tree(big.json)
 WHERE atom IS NOT NULL;
```

Suppose each entry in the BIG table is a JSON object with a '$.id' field that is a unique identifier and a '$.partlist' field that can be a deeply nested object. You want to find the id of every entry that contains one or more references to uuid '6fa5181e-5721-11e5-a04e-57f3d7b32808' anywhere in its '$.partlist'.

```
SELECT DISTINCT json_extract(big.json,'$.id')
  FROM big, json_tree(big.json, '$.partlist')
 WHERE json_tree.key='uuid'
   AND json_tree.value='6fa5181e-5721-11e5-a04e-57f3d7b32808';
```

Datatypes In SQLite Version 3

Most SQL database engines (every SQL database engine other than SQLite, as far as we know) uses static, rigid typing. With static typing, the datatype of a value is determined by its container - the particular column in which the value is stored.

SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container. The dynamic type system of SQLite is backwards compatible with the more common static type systems of other database engines in the sense that SQL statements that work on statically typed databases should work the same way in SQLite. However, the dynamic typing in SQLite allows it to do things which are not possible in traditional rigidly typed databases.

1.0 Storage Classes and Datatypes

Each value stored in an SQLite database (or manipulated by the database engine) has one of the following storage classes:

- **NULL**. The value is a NULL value.
- **INTEGER**. The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- **REAL**. The value is a floating point value, stored as an 8-byte IEEE floating point number.
- **TEXT**. The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB**. The value is a blob of data, stored exactly as it was input.

Note that a storage class is slightly more general than a datatype. The INTEGER storage class, for example, includes 6 different integer datatypes of different lengths. This makes a difference on disk. But as soon as INTEGER values are read off of disk and into memory for processing, they are converted to the most general datatype (8-byte signed integer). And so for the most part, "storage class" is indistinguishable from "datatype" and the two terms can be used interchangeably.

Any column in an SQLite version 3 database, except an INTEGER PRIMARY KEY column, may be used to store a value of any storage class.

All values in SQL statements, whether they are literals embedded in SQL statement text or parameters bound to precompiled SQL statements have an implicit storage class. Under circumstances described below, the database engine may convert values between numeric storage classes (INTEGER and REAL) and TEXT during query execution.

1.1 Boolean Datatype

SQLite does not have a separate Boolean storage class. Instead, Boolean values are stored as integers 0 (false) and 1 (true).

1.2 Date and Time Datatype

SQLite does not have a storage class set aside for storing dates and/or times. Instead, the built-in Date And Time Functions of SQLite are capable of storing dates and times as TEXT, REAL, or INTEGER values:

- **TEXT** as ISO8601 strings ("YYYY-MM-DD HH:MM:SS.SSS").
- **REAL** as Julian day numbers, the number of days since noon in Greenwich on November 24, 4714 B.C. according to the proleptic Gregorian calendar.
- **INTEGER** as Unix Time, the number of seconds since 1970-01-01 00:00:00 UTC.

Applications can chose to store dates and times in any of these formats and freely convert between formats using the built-in date and time functions.

2.0 Type Affinity

In order to maximize compatibility between SQLite and other database engines, SQLite supports the concept of "type affinity" on columns. The type affinity of a column is the recommended type for data stored in that column. The important idea here is that the type is recommended, not required. Any column can still store any type of data. It is just that some columns, given the choice, will prefer to use one storage class over another. The preferred storage class for a column is called its "affinity".

Each column in an SQLite 3 database is assigned one of the following type affinities:

- TEXT
- NUMERIC
- INTEGER
- REAL
- BLOB

(Historical note: The "BLOB" type affinity used to be called "NONE". But that term was easy to confuse with "no affinity" and so it was renamed.)

A column with TEXT affinity stores all data using storage classes NULL, TEXT or BLOB. If numerical data is inserted into a column with TEXT affinity it is converted into text form before being stored.

A column with NUMERIC affinity may contain values using all five storage classes. When text data is inserted into a NUMERIC column, the storage class of the text is converted to INTEGER or REAL (in order of preference) if such conversion is lossless and reversible. For conversions between TEXT and REAL storage classes, SQLite considers the conversion to be lossless and reversible if the first 15 significant decimal digits of the number are preserved. If the lossless conversion of TEXT to INTEGER or REAL is not possible then the value is stored using the TEXT storage class. No attempt is made to convert NULL or BLOB values.

A string might look like a floating-point literal with a decimal point and/or exponent notation but as long as the value can be expressed as an integer, the NUMERIC affinity will convert it into an integer. Hence, the string '3.0e+5' is stored in a column with NUMERIC affinity as the integer 300000, not as the floating point value 300000.0.

A column that uses INTEGER affinity behaves the same as a column with NUMERIC affinity. The difference between INTEGER and NUMERIC affinity is only evident in a CAST expression.

A column with REAL affinity behaves like a column with NUMERIC affinity except that it forces integer values into floating point representation. (As an internal optimization, small floating point values with no fractional component and stored in columns with REAL affinity are written to disk as integers in order to take up less space and are automatically converted back into floating point as the value is read out. This optimization is completely invisible at the SQL level and can only be detected by examining the raw bits of the database file.)

A column with affinity BLOB does not prefer one storage class over another and no attempt is made to coerce data from one storage class into another.

2.1 Determination Of Column Affinity

The affinity of a column is determined by the declared type of the column, according to the following rules in the order shown:

1. If the declared type contains the string "INT" then it is assigned INTEGER affinity.

2. If the declared type of the column contains any of the strings "CHAR", "CLOB", or "TEXT" then that column has TEXT affinity. Notice that the type VARCHAR contains the string "CHAR" and is thus assigned TEXT affinity.
3. If the declared type for a column contains the string "BLOB" or if no type is specified then the column has affinity BLOB.
4. If the declared type for a column contains any of the strings "REAL", "FLOA", or "DOUB" then the column has REAL affinity.
5. Otherwise, the affinity is NUMERIC.

Note that the order of the rules for determining column affinity is important. A column whose declared type is "CHARINT" will match both rules 1 and 2 but the first rule takes precedence and so the column affinity will be INTEGER.

2.2 Affinity Name Examples

The following table shows how many common datatype names from more traditional SQL implementations are converted into affinities by the five rules of the previous section. This table shows only a small subset of the datatype names that SQLite will accept. Note that numeric arguments in parentheses that following the type name (ex: "VARCHAR(255)") are ignored by SQLite - SQLite does not impose any length restrictions (other than the large global SQLITE_MAX_LENGTH limit) on the length of strings, BLOBs or numeric values.

Example Typenames From The CREATE TABLE Statement or CAST Expression	Resulting Affinity	Rule Used To Determine Affinity
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT INT2 INT8	INTEGER	1
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT	2
BLOB *no datatype specified*	BLOB	3
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL	4
NUMERIC DECIMAL(10,5) DATE DATETIME	NUMERIC	5

Note that a declared type of "FLOATING POINT" would give INTEGER affinity, not REAL affinity, due to the "INT" at the end of "POINT". And the declared type of "STRING" has an affinity of NUMERIC, not TEXT.

2.3 Column Affinity Behavior Example

The following SQL demonstrates how SQLite uses column affinity to do type conversions when values are inserted into a table.

```
CREATE TABLE t1(
    t  TEXT,     -- text affinity by rule 2
    nu NUMERIC,  -- numeric affinity by rule 5
    i  INTEGER,  -- integer affinity by rule 1
    r  REAL,     -- real affinity by rule 4
    no BLOB      -- no affinity by rule 3
);

-- Values stored as TEXT, INTEGER, INTEGER, REAL, TEXT.
INSERT INTO t1 VALUES('500.0', '500.0', '500.0', '500.0', '500.0');
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
text|integer|integer|real|text

-- Values stored as TEXT, INTEGER, INTEGER, REAL, REAL.
DELETE FROM t1;
INSERT INTO t1 VALUES(500.0, 500.0, 500.0, 500.0, 500.0);
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
text|integer|integer|real|real

-- Values stored as TEXT, INTEGER, INTEGER, REAL, INTEGER.
DELETE FROM t1;
INSERT INTO t1 VALUES(500, 500, 500, 500, 500);
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
text|integer|integer|real|integer

-- BLOBs are always stored as BLOBs regardless of column affinity.
DELETE FROM t1;
INSERT INTO t1 VALUES(x'0500', x'0500', x'0500', x'0500', x'0500');
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
blob|blob|blob|blob|blob

-- NULLs are also unaffected by affinity
DELETE FROM t1;
INSERT INTO t1 VALUES(NULL,NULL,NULL,NULL,NULL);
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
null|null|null|null|null
```

3.0 Comparison Expressions

SQLite version 3 has the usual set of SQL comparison operators including "=", "==", "<", "<=", ">", ">=", "!=", "<>", "IN", "NOT IN", "BETWEEN", "IS", and "IS NOT", .

3.1 Sort Order

The results of a comparison depend on the storage classes of the operands, according to the following rules:

- A value with storage class NULL is considered less than any other value (including another value with storage class NULL).
- An INTEGER or REAL value is less than any TEXT or BLOB value. When an INTEGER or REAL is compared to another INTEGER or REAL, a numerical comparison is performed.

- A TEXT value is less than a BLOB value. When two TEXT values are compared an appropriate collating sequence is used to determine the result.
- When two BLOB values are compared, the result is determined using memcmp().

3.2 Affinity Of Comparison Operands

SQLite may attempt to convert values between the storage classes INTEGER, REAL, and/or TEXT before performing a comparison. Whether or not any conversions are attempted before the comparison takes place depends on the type affinity of the operands.

Note that every table column as a type affinity (one of BLOB, TEXT, INTEGER, REAL, or NUMERIC) but expressions do no necessarily have an affinity.

Operand affinity is determined by the following rules:

- The right-hand operand of an IN or NOT IN operator has no affinity if the operand is a list and has the same affinity as the affinity of the result set expression if the operand is a SELECT.
- An expression that is a simple reference to a column value has the same affinity as the column. Note that if X and Y.Z are column names, then +X and +Y.Z are considered expressions for the purpose of determining affinity.
- An expression of the form "CAST(*expr* AS *type*)" has an affinity that is the same as a column with a declared type of "*type*".
- Otherwise, an expression has no affinity.

3.3 Type Conversions Prior To Comparison

To "apply affinity" means to convert an operand to a particular storage class if and only if the conversion is lossless and reversible. Affinity is applied to operands of a comparison operator prior to the comparison according to the following rules in the order shown:

- If one operand has INTEGER, REAL or NUMERIC affinity and the other operand has TEXT or BLOB or no affinity then NUMERIC affinity is applied to other operand.
- If one operand has TEXT affinity and the other has no affinity, then TEXT affinity is applied to the other operand.
- Otherwise, no affinity is applied and both operands are compared as is.

The expression "a BETWEEN b AND c" is treated as two separate binary comparisons "a >= b AND a <= c", even if that means different affinities are applied to 'a' in each of the comparisons. Datatype conversions in comparisons of the form "x IN (SELECT y ...)" are handled is if the comparison were really "x=y". The expression "a IN (x, y, z, ...)" is equivalent to "a = +x OR a = +y OR a = +z OR ...". In other words, the values to the right of the IN operator (the "x", "y", and "z" values in this example) are considered to have no affinity, even if they happen to be column values or CAST expressions.

3.4 Comparison Example

```
CREATE TABLE t1(
    a TEXT,      -- text affinity
    b NUMERIC,   -- numeric affinity
    c BLOB,      -- no affinity
    d            -- no affinity
);

-- Values will be stored as TEXT, INTEGER, TEXT, and INTEGER respectively
INSERT INTO t1 VALUES('500', '500', '500', 500);
SELECT typeof(a), typeof(b), typeof(c), typeof(d) FROM t1;
text|integer|text|integer

-- Because column "a" has text affinity, numeric values on the
-- right-hand side of the comparisons are converted to text before
-- the comparison occurs.
SELECT a &lt; 40,   a &lt; 60,   a &lt; 600 FROM t1;
0|1|1

-- Text affinity is applied to the right-hand operands but since
-- they are already TEXT this is a no-op; no conversions occur.
SELECT a &lt; '40', a &lt; '60', a &lt; '600' FROM t1;
0|1|1

-- Column "b" has numeric affinity and so numeric affinity is applied
-- to the operands on the right.  Since the operands are already numeric,
-- the application of affinity is a no-op; no conversions occur.  All
-- values are compared numerically.
SELECT b &lt; 40,   b &lt; 60,   b &lt; 600 FROM t1;
0|0|1

-- Numeric affinity is applied to operands on the right, converting them
-- from text to integers.  Then a numeric comparison occurs.
SELECT b &lt; '40', b &lt; '60', b &lt; '600' FROM t1;
0|0|1

-- No affinity conversions occur.  Right-hand side values all have
-- storage class INTEGER which are always less than the TEXT values
-- on the left.
SELECT c &lt; 40,   c &lt; 60,   c &lt; 600 FROM t1;
0|0|0

-- No affinity conversions occur.  Values are compared as TEXT.
SELECT c &lt; '40', c &lt; '60', c &lt; '600' FROM t1;
0|1|1

-- No affinity conversions occur.  Right-hand side values all have
-- storage class INTEGER which compare numerically with the INTEGER
-- values on the left.
SELECT d &lt; 40,   d &lt; 60,   d &lt; 600 FROM t1;
0|0|1

-- No affinity conversions occur.  INTEGER values on the left are
-- always less than TEXT values on the right.
SELECT d &lt; '40', d &lt; '60', d &lt; '600' FROM t1;
1|1|1
```

All of the result in the example are the same if the comparisons are commuted - if expressions of the form "a<40" are rewritten as "40>a".

4.0 Operators

All mathematical operators (+, -, *, /, %, <<, >>, &, and |) cast both operands to the NUMERIC storage class prior to being carried out. The cast is carried through even if it is lossy and irreversible. A NULL operand on a mathematical operator yields a NULL result. An operand on a mathematical operator that does not look in any way numeric and is not NULL is converted to 0 or 0.0.

5.0 Sorting, Grouping and Compound SELECTs

When query results are sorted by an ORDER BY clause, values with storage class NULL come first, followed by INTEGER and REAL values interspersed in numeric order, followed by TEXT values in collating sequence order, and finally BLOB values in memcmp() order. No storage class conversions occur before the sort.

When grouping values with the GROUP BY clause values with different storage classes are considered distinct, except for INTEGER and REAL values which are considered equal if they are numerically equal. No affinities are applied to any values as the result of a GROUP by clause.

The compound SELECT operators UNION, INTERSECT and EXCEPT perform implicit comparisons between values. No affinity is applied to comparison operands for the implicit comparisons associated with UNION, INTERSECT, or EXCEPT - the values are compared as is.

6.0 Collating Sequences

When SQLite compares two strings, it uses a collating sequence or collating function (two words for the same thing) to determine which string is greater or if the two strings are equal. SQLite has three built-in collating functions: BINARY, NOCASE, and RTRIM.

- **BINARY** - Compares string data using memcmp(), regardless of text encoding.
- **NOCASE** - The same as binary, except the 26 upper case characters of ASCII are folded to their lower case equivalents before the comparison is performed. Note that only ASCII characters are case folded. SQLite does not attempt to do full UTF case folding due to the size of the tables required.
- **RTRIM** - The same as binary, except that trailing space characters are ignored.

An application can register additional collating functions using the sqlite3_create_collation() interface.

6.1 Assigning Collating Sequences from SQL

Every column of every table has an associated collating function. If no collating function is explicitly defined, then the collating function defaults to BINARY. The COLLATE clause of the column definition is used to define alternative collating functions for a column.

The rules for determining which collating function to use for a binary comparison operator (=, <, >, <=, >=, !=, IS, and IS NOT) are as follows and in the order shown:

1. If either operand has an explicit collating function assignment using the postfix COLLATE operator, then the explicit collating function is used for comparison, with precedence to the collating function of the left operand.

2. If either operand is a column, then the collating function of that column is used with precedence to the left operand. For the purposes of the previous sentence, a column name preceded by one or more unary "+" operators is still considered a column name.

3. Otherwise, the BINARY collating function is used for comparison.

An operand of a comparison is considered to have an explicit collating function assignment (rule 1 above) if any subexpression of the operand uses the postfix COLLATE operator. Thus, if a COLLATE operator is used anywhere in a comparision expression, the collating function defined by that operator is used for string comparison regardless of what table columns might be a part of that expression. If two or more COLLATE operator subexpressions appear anywhere in a comparison, the left most explicit collating function is used regardless of how deeply the COLLATE operators are nested in the expression and regardless of how the expression is parenthesized.

The expression "x BETWEEN y and z" is logically equivalent to two comparisons "x >= y AND x <= z" and works with respect to collating functions as if it were two separate comparisons. The expression "x IN (SELECT y ...)" is handled in the same way as the expression "x = y" for the purposes of determining the collating sequence. The collating sequence used for expressions of the form "x IN (y, z, ...)" is the collating sequence of x.

Terms of the ORDER BY clause that is part of a SELECT statement may be assigned a collating sequence using the COLLATE operator, in which case the specified collating function is used for sorting. Otherwise, if the expression sorted by an ORDER BY clause is a column, then the collating sequence of the column is used to determine sort order. If the expression is not a column and has no COLLATE clause, then the BINARY collating sequence is used.

6.2 Collation Sequence Examples

The examples below identify the collating sequences that would be used to determine the results of text comparisons that may be performed by various SQL statements. Note that a text comparison may not be required, and no collating sequence used, in the case of numeric, blob or NULL values.

```
CREATE TABLE t1(
    x INTEGER PRIMARY KEY,
    a,                 /* collating sequence BINARY */
    b COLLATE BINARY,  /* collating sequence BINARY */
    c COLLATE RTRIM,   /* collating sequence RTRIM  */
    d COLLATE NOCASE   /* collating sequence NOCASE */
);
                   /* x   a     b    c        d */
INSERT INTO t1 VALUES(1,'abc','abc', 'abc  ','abc');
INSERT INTO t1 VALUES(2,'abc','abc', 'abc',  'ABC');
INSERT INTO t1 VALUES(3,'abc','abc', 'abc ', 'Abc');
INSERT INTO t1 VALUES(4,'abc','abc ','ABC',  'abc');

/* Text comparison a=b is performed using the BINARY collating sequence. */
SELECT x FROM t1 WHERE a = b ORDER BY x;
--result 1 2 3

/* Text comparison a=b is performed using the RTRIM collating sequence. */
SELECT x FROM t1 WHERE a = b COLLATE RTRIM ORDER BY x;
--result 1 2 3 4

/* Text comparison d=a is performed using the NOCASE collating sequence. */
SELECT x FROM t1 WHERE d = a ORDER BY x;
--result 1 2 3 4

/* Text comparison a=d is performed using the BINARY collating sequence. */
SELECT x FROM t1 WHERE a = d ORDER BY x;
--result 1 4

/* Text comparison 'abc'=c is performed using the RTRIM collating sequence. */
SELECT x FROM t1 WHERE 'abc' = c ORDER BY x;
--result 1 2 3

/* Text comparison c='abc' is performed using the RTRIM collating sequence. */
SELECT x FROM t1 WHERE c = 'abc' ORDER BY x;
--result 1 2 3

/* Grouping is performed using the NOCASE collating sequence (Values
** 'abc', 'ABC', and 'Abc' are placed in the same group). */
SELECT count(*) FROM t1 GROUP BY d ORDER BY 1;
--result 4

/* Grouping is performed using the BINARY collating sequence.  'abc' and
** 'ABC' and 'Abc' form different groups */
SELECT count(*) FROM t1 GROUP BY (d || '') ORDER BY 1;
--result 1 1 2

/* Sorting or column c is performed using the RTRIM collating sequence. */
SELECT x FROM t1 ORDER BY c, x;
--result 4 1 2 3

/* Sorting of (c||'') is performed using the BINARY collating sequence. */
SELECT x FROM t1 ORDER BY (c||''), x;
--result 4 2 3 1

/* Sorting of column c is performed using the NOCASE collating sequence. */
SELECT x FROM t1 ORDER BY c COLLATE NOCASE, x;
--result 2 4 3 1
```

SQLite Features and Extensions

SQLite And 8+3 Filenames

The default configuration of SQLite assumes the underlying filesystem supports long filenames.

SQLite does not impose any naming requirements on database files. SQLite will happily work with a database file that has any filename extension or with no extension at all. When auxiliary files are needed for a rollback journal or a write-ahead log or for one of the other kinds of temporary disk files, then the name for the auxiliary file is normally constructed by appending a suffix onto the end of the database file name. For example, if the original database is call "`app.db`" then the rollback journal will be called "`app.db-journal`" and the write-ahead log will be called "`app.db-wal`". This approach to auxiliary file naming works great on systems that support long filenames. But on systems that impose 8+3 filename constraints, the auxiliary files do not fit the 8+3 format even though the original database file does.

Changing Filesystems

The recommended fix for this problem is to select a different filesystem. These days, there is a huge selection of high-performance, reliable, patent-free filesystems that support long filenames. Where possible, it is recommended that embedded devices use one of these other filesystems. This will avoid compatibility issues and the danger of database corruption caused by inconsistent use of 8+3 filenames.

Adjusting SQLite To Use 8+3 Filenames

Some devices are compelled to use an older filesystem with 8+3 filename restrictions for backwards compatibility, or due to other non-technical factors. In such situations, SQLite can be coerced into using auxiliary files that fit the 8+3 pattern as follows:

1. Compile the SQLite library with the either the compile-time options SQLITE_ENABLE_8_3_NAMES=1 or SQLITE_ENABLE_8_3_NAMES=2. Support for 8+3 filenames is not included in SQLite by default because it does introduce some overhead. The overhead is tiny, but even so, we do not want to burden the billions of SQLite applications that do not need 8+3 filename support.

2. If the SQLITE_ENABLE_8_3_NAMES=1 option is used, then SQLite is capable of using 8+3 filenames but that capabilities is disabled and must be enabled separately for each database connection by using using URI filenames when opening or ATTACH-ing the

database files and include the "`8_3_names=1`" query parameter in the URI. If SQLite is compiled with SQLITE_ENABLE_8_3_NAMES=2 then 8+3 filenames are enabled by default and this step can be skipped.

3. Make sure that database filenames follow the 8+3 filename format and that they do not have an empty name or extension. In other words, the database filename must contain between 1 and 8 characters in the base name and between 1 and 3 characters in the extension. Blank extensions are not allowed.

When the steps above are used, SQLite will shorten filename extensions by only using the last 3 characters of the extension. Thus, for example, a file that would normally be called "`app.db-journal`" is shortened to just "`app.nal`". Similarly, "`app.db-wal`" will become "`app.wal`" and "`app.db-shm`" becomes "`app.shm`".

Note that it is very important that the database filename have some kind of extension. If there is no extension, then SQLite creates auxiliary filenames by appending to the base name of the file. Thus, a database named "`db01`" would have a rollback journal file named "`db01-journal`". And as this filename has no extension to shorten to 3 characters, it will be used as-is, and will violate 8+3 naming rules.

Database Corruption Warning

If a database file is accessed using 8+3 naming rather than the default long filename, then it must be consistently accessed using 8+3 naming by every database connection every time it is opened, or else there is a risk of database corruption. The auxiliary rollback journal and write-ahead log files are essential to SQLite for being about to recover from a crash. If an application is using 8+3 names and crashes, then the information needed to safely recover from the crash is stored in files with the "`.nal`" or "`.wal`" extension. If the next application to open the database does not specify the "`8_3_names=1`" URI parameter, then SQLite will use the long filenames to try to locate the rollback journal or write-ahead log files. It will not find them, since they were saved using 8+3 names by the application that crashed, and hence the database will not be properly recovered and will likely go corrupt.

Using a database file with 8+3 filenames in some cases while in other cases using long filenames is equivalent to deleting a hot journal.

Autoincrement In SQLite

Summary

1. The AUTOINCREMENT keyword imposes extra CPU, memory, disk space, and disk I/O overhead and should be avoided if not strictly needed. It is usually not needed.
2. In SQLite, a column with type INTEGER PRIMARY KEY is an alias for the ROWID (except in WITHOUT ROWID tables) which is always a 64-bit signed integer.
3. On an INSERT, if the ROWID or INTEGER PRIMARY KEY column is not explicitly given a value, then it will be filled automatically with an unused integer, usually one more than the largest ROWID currently in use. This is true regardless of whether or not the AUTOINCREMENT keyword is used.
4. If the AUTOINCREMENT keyword appears after INTEGER PRIMARY KEY, that changes the automatic ROWID assignment algorithm to prevent the reuse of ROWIDs over the lifetime of the database. In other words, the purpose of AUTOINCREMENT is to prevent the reuse of ROWIDs from previously deleted rows.

Background

In SQLite, table rows normally have a 64-bit signed integer ROWID which is unique among all rows in the same table. (WITHOUT ROWID tables are the exception.)

You can access the ROWID of an SQLite table using one of the special column names ROWID, *ROWID*, or OID. Except if you declare an ordinary table column to use one of those special names, then the use of that name will refer to the declared column not to the internal ROWID.

If a table contains a column of type INTEGER PRIMARY KEY, then that column becomes an alias for the ROWID. You can then access the ROWID using any of four different names, the original three names described above or the name given to the INTEGER PRIMARY KEY column. All these names are aliases for one another and work equally well in any context.

When a new row is inserted into an SQLite table, the ROWID can either be specified as part of the INSERT statement or it can be assigned automatically by the database engine. To specify a ROWID manually, just include it in the list of values to be inserted. For example:

```
CREATE TABLE test1(a INT, b TEXT);
INSERT INTO test1(rowid, a, b) VALUES(123, 5, 'hello');
```

If no ROWID is specified on the insert, or if the specified ROWID has a value of NULL, then an appropriate ROWID is created automatically. The usual algorithm is to give the newly created row a ROWID that is one larger than the largest ROWID in the table prior to the insert. If the table is initially empty, then a ROWID of 1 is used. If the largest ROWID is equal to the largest possible integer (9223372036854775807) then the database engine starts picking positive candidate ROWIDs at random until it finds one that is not previously used. If no unused ROWID can be found after a reasonable number of attempts, the insert operation fails with an SQLITE_FULL error. If no negative ROWID values are inserted explicitly, then automatically generated ROWID values will always be greater than zero.

The normal ROWID selection algorithm described above will generate monotonically increasing unique ROWIDs as long as you never use the maximum ROWID value and you never delete the entry in the table with the largest ROWID. If you ever delete rows or if you ever create a row with the maximum possible ROWID, then ROWIDs from previously deleted rows might be reused when creating new rows and newly created ROWIDs might not be in strictly ascending order.

The AUTOINCREMENT Keyword

If a column has the type INTEGER PRIMARY KEY AUTOINCREMENT then a slightly different ROWID selection algorithm is used. The ROWID chosen for the new row is at least one larger than the largest ROWID that has ever before existed in that same table. If the table has never before contained any data, then a ROWID of 1 is used. If the table has previously held a row with the largest possible ROWID, then new INSERTs are not allowed and any attempt to insert a new row will fail with an SQLITE_FULL error. Only ROWID values from previously transactions that were committed are considered. ROWID values that were rolled back are ignored and can be reused.

SQLite keeps track of the largest ROWID that a table has ever held using an internal table named "sqlite_sequence". The sqlite_sequence table is created and initialized automatically whenever a normal table that contains an AUTOINCREMENT column is created. The content of the sqlite_sequence table can be modified using ordinary UPDATE, INSERT, and DELETE statements. But making modifications to this table will likely perturb the AUTOINCREMENT key generation algorithm. Make sure you know what you are doing before you undertake such changes.

The behavior implemented by the AUTOINCREMENT keyword is subtly different from the default behavior. With AUTOINCREMENT, rows with automatically selected ROWIDs are guaranteed to have ROWIDs that have never been used before by the same table in the same database. And the automatically generated ROWIDs are guaranteed to be monotonically increasing. These are important properties in certain applications. But if your application does not need these properties, you should probably stay with the default behavior since the use of AUTOINCREMENT requires additional work to be done as each row is inserted and thus causes INSERTs to run a little slower.

Note that "monotonically increasing" does not imply that the ROWID always increases by exactly one. One is the usual increment. However, if an insert fails due to (for example) a uniqueness constraint, the ROWID of the failed insertion attempt might not be reused on subsequent inserts, resulting in gaps in the ROWID sequence. AUTOINCREMENT guarantees that automatically chosen ROWIDs will be increasing but not that they will be sequential.

Because AUTOINCREMENT keyword changes the behavior of the ROWID selection algorithm, AUTOINCREMENT is not allowed on WITHOUT ROWID tables or on any table column other than INTEGER PRIMARY KEY. Any attempt to use AUTOINCREMENT on a WITHOUT ROWID table or on a column other than the INTEGER PRIMARY KEY column results in an error.

Using the SQLite Online Backup API

Historically, backups (copies) of SQLite databases have been created using the following method:

1. Establish a shared lock on the database file using the SQLite API (i.e. the shell tool).
2. Copy the database file using an external tool (for example the unix 'cp' utility or the DOS 'copy' command).
3. Relinquish the shared lock on the database file obtained in step 1.

This procedure works well in many scenarios and is usually very fast. However, this technique has the following shortcomings:

- Any database clients wishing to write to the database file while a backup is being created must wait until the shared lock is relinquished.
- It cannot be used to copy data to or from in-memory databases.
- If a power failure or operating system failure occurs while copying the database file the backup database may be corrupted following system recovery.

The Online Backup API was created to address these concerns. The online backup API allows the contents of one database to be copied into another database, overwriting the original contents of the target database. The copy operation may be done incrementally, in which case the source database does not need to be locked for the duration of the copy, only for the brief periods of time when it is actually being read from. This allows other database users to continue uninterrupted while a backup of an online database is made.

The online backup API is documented here. The remainder of this page contains two C language examples illustrating common uses of the API and discussions thereof. Reading these examples is no substitute for reading the API documentation!

Example 1: Loading and Saving In-Memory Databases

```
/*
** This function is used to load the contents of a database file on disk
** into the "main" database of open database connection pInMemory, or
** to save the current contents of the database opened by pInMemory into
** a database file on disk. pInMemory is probably an in-memory database,
** but this function will also work fine if it is not.
**
** Parameter zFilename points to a nul-terminated string containing the
** name of the database file on disk to load from or save to. If parameter
** isSave is non-zero, then the contents of the file zFilename are
** overwritten with the contents of the database opened by pInMemory. If
** parameter isSave is zero, then the contents of the database opened by
** pInMemory are replaced by data loaded from the file zFilename.
**
** If the operation is successful, SQLITE_OK is returned. Otherwise, if
** an error occurs, an SQLite error code is returned.
*/
int loadOrSaveDb(sqlite3 *pInMemory, const char *zFilename, int isSave){
  int rc;                   /* Function return code */
  sqlite3 *pFile;           /* Database connection opened on zFilename */
  sqlite3_backup *pBackup;  /* Backup object used to copy data */
  sqlite3 *pTo;             /* Database to copy to (pFile or pInMemory) */
  sqlite3 *pFrom;           /* Database to copy from (pFile or pInMemory) */

  /* Open the database file identified by zFilename. Exit early if this fails
  ** for any reason. */
  rc = sqlite3_open(zFilename, &pFile);
  if( rc==SQLITE_OK ){

    /* If this is a 'load' operation (isSave==0), then data is copied
    ** from the database file just opened to database pInMemory.
    ** Otherwise, if this is a 'save' operation (isSave==1), then data
    ** is copied from pInMemory to pFile.  Set the variables pFrom and
    ** pTo accordingly. */
    pFrom = (isSave ? pInMemory : pFile);
    pTo   = (isSave ? pFile     : pInMemory);

    /* Set up the backup procedure to copy from the "main" database of
    ** connection pFile to the main database of connection pInMemory.
    ** If something goes wrong, pBackup will be set to NULL and an error
    ** code and  message left in connection pTo.
    **
    ** If the backup object is successfully created, call backup_step()
    ** to copy data from pFile to pInMemory. Then call backup_finish()
    ** to release resources associated with the pBackup object.  If an
    ** error occurred, then  an error code and message will be left in
    ** connection pTo. If no error occurred, then the error code belonging
    ** to pTo is set to SQLITE_OK.
    */
    pBackup = sqlite3_backup_init(pTo, "main", pFrom, "main");
    if( pBackup ){
      (void)sqlite3_backup_step(pBackup, -1);
      (void)sqlite3_backup_finish(pBackup);
    }
    rc = sqlite3_errcode(pTo);
  }

  /* Close the database connection opened on database file zFilename
  ** and return the result of this function. */
  (void)sqlite3_close(pFile);
  return rc;
}
```

The C function to the right demonstrates of one of the simplest, and most common, uses of the backup API: loading and saving the contents of an in-memory database to a file on disk. The backup API is used as follows in this example:

1. Function sqlite3_backup_init() is called to create an sqlite3_backup object to copy data between the two databases (either from a file and into the in-memory database, or vice-versa).
2. Function sqlite3_backup_step() is called with a parameter of `-1` to copy the entire source database to the destination.
3. Function sqlite3_backup_finish() is called to clean up resources allocated by sqlite3_backup_init().

Error handling

If an error occurs in any of the three main backup API routines then the error code and message are attached to the destination database connection. Additionally, if sqlite3_backup_step() encounters an error, then the error code is returned by both the sqlite3_backup_step() call itself, and by the subsequent call to sqlite3_backup_finish(). So a call to sqlite3_backup_finish() does not overwrite an error code stored in the destination database connection by sqlite3_backup_step(). This feature is used in the example code to reduce amount of error handling required. The return values of the sqlite3_backup_step() and sqlite3_backup_finish() calls are ignored and the error code indicating the success or failure of the copy operation collected from the destination database connection afterward.

Possible Enhancements

The implementation of this function could be enhanced in at least two ways:

1. Failing to obtain the lock on database file zFilename (an SQLITE_BUSY error) could be handled, and
2. Cases where the page-sizes of database pInMemory and zFilename are different could be handled better.

Since database zFilename is a file on disk, then it may be accessed externally by another process. This means that when the call to sqlite3_backup_step() attempts to read from or write data to it, it may fail to obtain the required file lock. If this happens, this implementation will fail, returning SQLITE_BUSY immediately. The solution would be to register a busy-handler callback or timeout with database connection pFile using sqlite3_busy_handler() or sqlite3_busy_timeout() as soon as it is opened. If it fails to obtain a required lock immediately, sqlite3_backup_step() uses any registered busy-handler callback or timeout in the same way as sqlite3_step() or sqlite3_exec() does.

Usually, it does not matter if the page-sizes of the source database and the destination database are different before the contents of the destination are overwritten. The page-size of the destination database is simply changed as part of the backup operation. The exception is if the destination database happens to be an in-memory database. In this case, if the page sizes are not the same at the start of the backup operation, then the operation fails with an SQLITE_READONLY error. Unfortunately, this could occur when loading a database image from a file into an in-memory database using function loadOrSaveDb().

However, if in-memory database pInMemory has just been opened (and is therefore completely empty) before being passed to function loadOrSaveDb(), then it is still possible to change its page size using an SQLite "PRAGMA page_size" command. Function loadOrSaveDb() could detect this case, and attempt to set the page-size of the in-memory database to the page-size of database zFilename before invoking the online backup API functions.

Example 2: Online Backup of a Running Database

```
/*
** Perform an online backup of database pDb to the database file named
** by zFilename. This function copies 5 database pages from pDb to
** zFilename, then unlocks pDb and sleeps for 250 ms, then repeats the
** process until the entire database is backed up.
**
** The third argument passed to this function must be a pointer to a progress
** function. After each set of 5 pages is backed up, the progress function
** is invoked with two integer parameters: the number of pages left to
** copy, and the total number of pages in the source file. This information
** may be used, for example, to update a GUI progress bar.
**
** While this function is running, another thread may use the database pDb, or
** another process may access the underlying database file via a separate
** connection.
**
** If the backup process is successfully completed, SQLITE_OK is returned.
** Otherwise, if an error occurs, an SQLite error code is returned.
*/
int backupDb(
  sqlite3 *pDb,               /* Database to back up */
  const char *zFilename,      /* Name of file to back up to */
  void(*xProgress)(int, int)  /* Progress function to invoke */
){
  int rc;                     /* Function return code */
  sqlite3 *pFile;             /* Database connection opened on zFilename */
  sqlite3_backup *pBackup;    /* Backup handle used to copy data */

  /* Open the database file identified by zFilename. */
  rc = sqlite3_open(zFilename, &pFile);
  if( rc==SQLITE_OK ){

    /* Open the [sqlite3_backup](c3ref/backup.html) object used to accomplish the transfe
    pBackup = sqlite3_backup_init(pFile, "main", pDb, "main");
    if( pBackup ){

      /* Each iteration of this loop copies 5 database pages from database
      ** pDb to the backup database. If the return value of backup_step()
      ** indicates that there are still further pages to copy, sleep for
      ** 250 ms before repeating. */
      do {
        rc = sqlite3_backup_step(pBackup, 5);
        xProgress(
            sqlite3_backup_remaining(pBackup),
            sqlite3_backup_pagecount(pBackup)
        );
        if( rc==SQLITE_OK || rc==SQLITE_BUSY || rc==SQLITE_LOCKED ){
          sqlite3_sleep(250);
        }
      } while( rc==SQLITE_OK || rc==SQLITE_BUSY || rc==SQLITE_LOCKED );

      /* Release resources allocated by backup_init(). */
      (void)sqlite3_backup_finish(pBackup);
    }
    rc = sqlite3_errcode(pFile);
  }

  /* Close the database connection opened on database file zFilename
  ** and return the result of this function. */
  (void)sqlite3_close(pFile);
  return rc;
}
```

The function presented in the previous example copies the entire source database in one call to sqlite3_backup_step(). This requires holding a read-lock on the source database file for the duration of the operation, preventing any other database user from writing to the database. It also holds the mutex associated with database pInMemory throughout the copy, preventing any other thread from using it. The C function in this section, designed to be called by a background thread or process for creating a backup of an online database, avoids these problems using the following approach:

1. Function sqlite3_backup_init() is called to create an sqlite3_backup object to copy data from database pDb to the backup database file identified by zFilename.
2. Function sqlite3_backup_step() is called with a parameter of 5 to copy 5 pages of database pDb to the backup database (file zFilename).
3. If there are still more pages to copy from database pDb, then the function sleeps for 250 milliseconds (using the sqlite3_sleep() utility) and then returns to step 2.
4. Function sqlite3_backup_finish() is called to clean up resources allocated by sqlite3_backup_init().

File and Database Connection Locking

During the 250 ms sleep in step 3 above, no read-lock is held on the database file and the mutex associated with pDb is not held. This allows other threads to use database connection pDb and other connections to write to the underlying database file.

If another thread or process writes to the source database while this function is sleeping, then SQLite detects this and usually restarts the backup process when sqlite3_backup_step() is next called. There is one exception to this rule: If the source database is not an in-memory database, and the write is performed from within the same process as the backup operation and uses the same database handle (pDb), then the destination database (the one opened using connection pFile) is automatically updated along with the source. The backup process may then be continued after the xSleep() call returns as if nothing had happened.

Whether or not the backup process is restarted as a result of writes to the source database mid-backup, the user can be sure that when the backup operation is completed the backup database contains a consistent and up-to-date snapshot of the original. However:

- Writes to an in-memory source database, or writes to a file-based source database by an external process or thread using a database connection other than pDb are significantly more expensive than writes made to a file-based source database using pDb (as the entire backup operation must be restarted in the former two cases).
- If the backup process is restarted frequently enough it may never run to completion and the backupDb() function may never return.

backup_remaining() and backup_pagecount()

The backupDb() function uses the sqlite3_backup_remaining() and sqlite3_backup_pagecount() functions to report its progress via the user-supplied xProgress() callback. Function sqlite3_backup_remaining() returns the number of pages left to copy and sqlite3_backup_pagecount() returns the total number of pages in the source database (in this case the database opened by pDb). So the percentage completion of the process may be calculated as:

Completion = 100% * (pagecount() - remaining()) / pagecount()

The sqlite3_backup_remaining() and sqlite3_backup_pagecount() APIs report values stored by the previous call to sqlite3_backup_step(), they do not actually inspect the source database file. This means that if the source database is written to by another thread or process after the call to sqlite3_backup_step() returns but before the values returned by sqlite3_backup_remaining() and sqlite3_backup_pagecount() are used, the values may be technically incorrect. This is not usually a problem.

Command Line Shell For SQLite

The SQLite project provides a simple command-line utility named **sqlite3** (or **sqlite3.exe** on windows) that allows the user to manually enter and execute SQL statements against an SQLite database. This document provides a brief introduction on how to use the **sqlite3** program.

Getting Started

To start the **sqlite3** program, just type "sqlite3" optionally followed by the name the file that holds the SQLite database. If the file does not exist, a new database file with the given name will be created automatically. If no database file is specified, a temporary database is created, then deleted when the "sqlite3" program exits.

When started, the **sqlite3** program will show a brief banner message then prompt you to enter SQL. Type in SQL statements (terminated by a semicolon), press "Enter" and the SQL will be executed.

For example, to create a new SQLite database named "ex1" with a single table named "tbl1", you might do this:

```
$ sqlite3 ex1
SQLite version 3.8.5 2014-05-29 12:36:14
Enter ".help" for usage hints.
sqlite> create table tbl1(one varchar(10), two smallint);
sqlite> insert into tbl1 values('hello!',10);
sqlite> insert into tbl1 values('goodbye', 20);
sqlite> select * from tbl1;
hello!|10
goodbye|20
sqlite>
```

You can terminate the sqlite3 program by typing your systems End-Of-File character (usually a Control-D). Use the interrupt character (usually a Control-C) to stop a long-running SQL statement.

Make sure you type a semicolon at the end of each SQL command! The sqlite3 program looks for a semicolon to know when your SQL command is complete. If you omit the semicolon, sqlite3 will give you a continuation prompt and wait for you to enter more text to be added to the current SQL command. This feature allows you to enter SQL commands that span multiple lines. For example:

```
sqlite> CREATE TABLE tbl2 (
   ...&gt;   f1 varchar(30) primary key,
   ...&gt;   f2 text,
   ...&gt;   f3 real
   ...&gt; );
sqlite>
```

Double-click Startup On Windows

Windows users can double-click on the **sqlite3.exe** icon to cause the command-line shell to pop-up a terminal window running SQLite. Note, however, that by default this SQLite session is using an in-memory database, not a file on disk, and so all changes will be lost when the session exits. To use a persistent disk file as the database, enter the ".open" command immediately after the terminal window starts up:

```
SQLite version 3.8.5 2014-05-29 12:36:14
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open ex1.db
sqlite>
```

The example above causes the database file named "ex1.db" to be opened and used, and created if it does not previously exist. You might want to use a full pathname to ensure that the file is in the directory that you think it is in. Use forward-slashes as the directory separator character. In other words use "c:/work/ex1.db", not "c:\work\ex1.db".

Alternatively, you can create a new database using the default in-memory storage, then save that database into a disk file using the ".save" command:

```
SQLite version 3.8.5 2014-05-29 12:36:14
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> ... many SQL commands omitted ...
sqlite> .save ex1.db
sqlite>
```

Be careful when using the ".save" command as it will overwrite any preexisting database files having the same name without prompting for confirmation. As with the ".open" command, you might want to use a full pathname with forward-slash directory separators to avoid ambiguity.

Special commands to sqlite3 (dot-commands)

Most of the time, sqlite3 just reads lines of input and passes them on to the SQLite library for execution. But if an input line begins with a dot ("."), then that line is intercepted and interpreted by the sqlite3 program itself. These "dot commands" are typically used to change the output format of queries, or to execute certain prepackaged query statements.

For a listing of the available dot commands, you can enter ".help" at any time. For example:

```
sqlite> .help
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail on|off           Stop after hitting an error.  Default OFF
.binary on|off         Turn binary output on or off.  Default OFF
.changes on|off        Show number of rows changed by SQL
.clone NEWDB           Clone data into NEWDB from the existing database
.databases             List names and files of attached databases
.dbinfo ?DB?           Show status information about the database
.dump ?TABLE? ...      Dump the database in an SQL text format
                         If TABLE specified, only dump tables matching
                         LIKE pattern TABLE.
.echo on|off           Turn command echo on or off
.eqp on|off            Enable or disable automatic EXPLAIN QUERY PLAN
.exit                  Exit this program
.explain ?on|off|auto? Turn EXPLAIN output mode on or off or to automatic
.fullschema            Show schema and the content of sqlite_stat tables
.headers on|off        Turn display of headers on or off
.help                  Show this message
.import FILE TABLE     Import data from FILE into TABLE
.indexes ?TABLE?       Show names of all indexes
                         If TABLE specified, only show indexes for tables
                         matching LIKE pattern TABLE.
.limit ?LIMIT? ?VAL?   Display or change the value of an SQLITE_LIMIT
.load FILE ?ENTRY?     Load an extension library
.log FILE|off          Turn logging on or off.  FILE can be stderr/stdout
.mode MODE ?TABLE?     Set output mode where MODE is one of:
                         ascii    Columns/rows delimited by 0x1F and 0x1E
                         csv      Comma-separated values
                         column   Left-aligned columns.  (See .width)
                         html     HTML &lt;table&gt; code
                         insert   SQL insert statements for TABLE
                         line     One value per line
                         list     Values delimited by .separator strings
                         tabs     Tab-separated values
                         tcl      TCL list elements
.nullvalue STRING      Use STRING in place of NULL values
.once FILENAME         Output for the next SQL command only to FILENAME
.open ?FILENAME?       Close existing database and reopen FILENAME
.output ?FILENAME?     Send output to FILENAME or stdout
.print STRING...       Print literal STRING
.prompt MAIN CONTINUE  Replace the standard prompts
.quit                  Exit this program
.read FILENAME         Execute SQL in FILENAME
.restore ?DB? FILE     Restore content of DB (default "main") from FILE
.save FILE             Write in-memory database into FILE
.scanstats on|off      Turn sqlite3_stmt_scanstatus() metrics on or off
.schema ?TABLE?        Show the CREATE statements
                         If TABLE specified, only show tables matching
                         LIKE pattern TABLE.
.separator COL ?ROW?   Change the column separator and optionally the row
                         separator for both the output mode and .import
.shell CMD ARGS...     Run CMD ARGS... in a system shell
.show                  Show the current values for various settings
.stats on|off          Turn stats on or off
.system CMD ARGS...    Run CMD ARGS... in a system shell
.tables ?TABLE?        List names of tables
                         If TABLE specified, only list tables matching
                         LIKE pattern TABLE.
.timeout MS            Try opening locked tables for MS milliseconds
.timer on|off          Turn SQL timer on or off
.trace FILE|off        Output each SQL statement as it is run
.vfsinfo ?AUX?         Information about the top-level VFS
.vfslist               List all available VFSes
.vfsname ?AUX?         Print the name of the VFS stack
.width NUM1 NUM2 ...   Set column widths for "column" mode
                         Negative values right-justify
sqlite>
```

Rules for "dot-commands"

Ordinary SQL statements are free-form, and can be spread across multiple lines, and can have whitespace and comments anywhere. But dot-commands are more restrictive:

- A dot-command must begin with the "." at the left margin with no preceding whitespace.
- The dot-command must be entirely contained on a single input line.
- A dot-command cannot occur in the middle of an ordinary SQL statement. In other words, a dot-command cannot occur at a continuation prompt.
- Dot-commands do not recognize comments.

And, of course, it is important to remember that the dot-commands are interpreted by the sqlite3.exe command-line program, not by SQLite itself. So none of the dot-commands will work as an argument to SQLite interfaces like sqlite3_prepare() or sqlite3_exec().

Changing Output Formats

The sqlite3 program is able to show the results of a query in eight different formats: "csv", "column", "html", "insert", "line", "list", "tabs", and "tcl". You can use the ".mode" dot command to switch between these output formats.

The default output mode is "list". In list mode, each record of a query result is written on one line of output and each column within that record is separated by a specific separator string. The default separator is a pipe symbol ("|"). List mode is especially useful when you are going to send the output of a query to another program (such as AWK) for additional processing.

```
sqlite> .mode list
sqlite> select * from tbl1;
hello|10
goodbye|20
sqlite>
```

You can use the ".separator" dot command to change the separator for list mode. For example, to change the separator to a comma and a space, you could do this:

```
sqlite> .separator ", "
sqlite> select * from tbl1;
hello, 10
goodbye, 20
sqlite>
```

In "line" mode, each column in a row of the database is shown on a line by itself. Each line consists of the column name, an equal sign and the column data. Successive records are separated by a blank line. Here is an example of line mode output:

```
sqlite> .mode line
sqlite> select * from tbl1;
one = hello
two = 10

one = goodbye
two = 20
sqlite>
```

In column mode, each record is shown on a separate line with the data aligned in columns. For example:

```
sqlite> .mode column
sqlite> select * from tbl1;
one         two
----------  ----------
hello       10
goodbye     20
sqlite>
```

By default, each column is between 1 and 10 characters wide, depending on the column header name and the width of the first column of data. Data that is too wide to fit in a column is truncated. You can adjust the column widths using the ".width" command. Like this:

```
sqlite> .width 12 6
sqlite> select * from tbl1;
one           two
------------  ------
hello         10
goodbye       20
sqlite>
```

The ".width" command in the example above sets the width of the first column to 12 and the width of the second column to 6. All other column widths were unaltered. You can gives as many arguments to ".width" as necessary to specify the widths of as many columns as are in your query results.

If you specify a column a width of 0, then the column width is automatically adjusted to be the maximum of three numbers: 10, the width of the header, and the width of the first row of data. This makes the column width self-adjusting. The default width setting for every column is this auto-adjusting 0 value.

You can specify a negative column width to get right-justified columns.

The column labels that appear on the first two lines of output can be turned on and off using the ".header" dot command. In the examples above, the column labels are on. To turn them off you could do this:

```
sqlite> .header off
sqlite> select * from tbl1;
hello         10
goodbye       20
sqlite>
```

Another useful output mode is "insert". In insert mode, the output is formatted to look like SQL INSERT statements. You can use insert mode to generate text that can later be used to input data into a different database.

When specifying insert mode, you have to give an extra argument which is the name of the table to be inserted into. For example:

```
sqlite> .mode insert new_table
sqlite> select * from tbl1;
INSERT INTO "new_table" VALUES('hello',10);
INSERT INTO "new_table" VALUES('goodbye',20);
sqlite>
```

The last output mode is "html". In this mode, sqlite3 writes the results of the query as an XHTML table. The beginning <TABLE> and the ending </TABLE> are not written, but all of the intervening <TR>s, <TH>s, and <TD>s are. The html output mode is envisioned as being useful for CGI.

The ".explain" dot command can be used to set the output mode to "column" and to set the column widths to values that are reasonable for looking at the output of an EXPLAIN command.

Beginning with Version 3.11.0, the command-line shell defaults to "auto-explain" mode, in which the EXPLAIN commands are automatically detected and the output is automatically formatted. So the ".explain" command has become superfluous.

Writing results to a file

By default, sqlite3 sends query results to standard output. You can change this using the ".output" and ".once" commands. Just put the name of an output file as an argument to .output and all subsequent query results will be written to that file. Or use the .once command instead of .output and output will only be redirected for the single next command before returning the console. Use .output with no arguments to begin writing to standard output again. For example:

```
sqlite> .mode list
sqlite> .separator |
sqlite> .output test_file_1.txt
sqlite> select * from tbl1;
sqlite> .exit
$ cat test_file_1.txt
hello|10
goodbye|20
$
```

If the first character of the ".output" or ".once" filename is a pipe symbol ("|") then the remaining characters are treated as a command and the output is sent to that command. This makes it easy to pipe the results of a query into some other process. For example, the "open -f" command on a Mac opens a text editor to display the content that it reads from standard input. So to see the results of a query in a text editor, one could type:

```
sqlite3&gt; .once '|open -f'
sqlite3&gt; SELECT * FROM bigTable;
```

File I/O Functions

The command-line shell adds two application-defined SQL functions that facilitate read content from a file into an table column, and writing the content of a column into a file, respectively.

The readfile(X) SQL function reads the entire content of the file named X and returns that content as a BLOB. This can be used to load content into a table. For example:

```
sqlite> CREATE TABLE images(name TEXT, type TEXT, img BLOB);
sqlite> INSERT INTO images(name,type,img)
   ...&gt;   VALUES('icon','jpeg',readfile('icon.jpg'));
```

The writefile(X,Y) SQL function write the blob Y into the file named X and returns the number of bytes written. Use this function to extract the content of a single table column into a file. For example:

```
sqlite> SELECT writefile('icon.jpg',img) FROM images WHERE name='icon';
```

Note that the readfile(X) and writefile(X,Y) functions are extension functions and are not built into the core SQLite library. These routines are available as a loadable extension in the ext/misc/fileio.c source file in the SQLite source code repositories.

Querying the database schema

The sqlite3 program provides several convenience commands that are useful for looking at the schema of the database. There is nothing that these commands do that cannot be done by some other means. These commands are provided purely as a shortcut.

For example, to see a list of the tables in the database, you can enter ".tables".

```
sqlite> .tables
tbl1
tbl2
sqlite>
```

The ".tables" command is similar to setting list mode then executing the following query:

```
SELECT name FROM sqlite_master
WHERE type IN ('table','view') AND name NOT LIKE 'sqlite_%'
UNION ALL
SELECT name FROM sqlite_temp_master
WHERE type IN ('table','view')
ORDER BY 1
```

In fact, if you look at the source code to the sqlite3 program (found in the source tree in the file src/shell.c) you'll find a query very much like the above.

The ".indices" command works in a similar way to list all of the indices for a particular table. The ".indices" command takes a single argument which is the name of the table for which the indices are desired. Last, but not least, is the ".schema" command. With no arguments, the ".schema" command shows the original CREATE TABLE and CREATE INDEX statements that were used to build the current database. If you give the name of a table to ".schema", it shows the original CREATE statement used to make that table and all if its indices. We have:

```
sqlite> .schema
create table tbl1(one varchar(10), two smallint)
CREATE TABLE tbl2 (
  f1 varchar(30) primary key,
  f2 text,
  f3 real
)
sqlite> .schema tbl2
CREATE TABLE tbl2 (
  f1 varchar(30) primary key,
  f2 text,
  f3 real
)
sqlite>
```

The ".schema" command accomplishes the same thing as setting list mode, then entering the following query:

```
SELECT sql FROM
   (SELECT * FROM sqlite_master UNION ALL
    SELECT * FROM sqlite_temp_master)
WHERE type!='meta'
ORDER BY tbl_name, type DESC, name
```

Or, if you give an argument to ".schema" because you only want the schema for a single table, the query looks like this:

```
SELECT sql FROM
   (SELECT * FROM sqlite_master UNION ALL
    SELECT * FROM sqlite_temp_master)
WHERE type!='meta' AND sql NOT NULL AND name NOT LIKE 'sqlite_%'
ORDER BY substr(type,2,1), name
```

You can supply an argument to the .schema command. If you do, the query looks like this:

```
SELECT sql FROM
   (SELECT * FROM sqlite_master UNION ALL
    SELECT * FROM sqlite_temp_master)
WHERE tbl_name LIKE '%s'
  AND type!='meta' AND sql NOT NULL AND name NOT LIKE 'sqlite_%'
ORDER BY substr(type,2,1), name
```

The "%s" in the query is replace by your argument. This allows you to view the schema for some subset of the database.

```
sqlite> .schema %abc%
```

Along these same lines, the ".table" command also accepts a pattern as its first argument. If you give an argument to the .table command, a "%" is both appended and prepended and a LIKE clause is added to the query. This allows you to list only those tables that match a particular pattern.

The ".databases" command shows a list of all databases open in the current connection. There will always be at least 2. The first one is "main", the original database opened. The second is "temp", the database used for temporary tables. There may be additional databases listed for databases attached using the ATTACH statement. The first output column is the name the database is attached with, and the second column is the filename of the external file.

```
sqlite> .databases
```

The ".fullschema" dot-command works like the ".schema" command in that it displays the entire database schema. But ".fullschema" also includes dumps of the statistics tables "sqlite_stat1", "sqlite_stat3", and "sqlite_stat4", if they exist. The ".fullschema" command normally provides all of the information needed to exactly recreate a query plan for a specific query. When reporting suspected problems with the SQLite query planner to the SQLite development team, developers are requested to provide the complete ".fullschema" output as part of the trouble report. Note that the sqlite_stat3 and sqlite_stat4 tables contain samples of index entries and so might contain sensitive data, so do not send the ".fullschema" output of a proprietary database over a public channel.

CSV Import

Use the ".import" command to import CSV (comma separated value) data into an SQLite table. The ".import" command takes two arguments which are the name of the disk file from which CSV data is to be read and the name of the SQLite table into which the CSV data is to be inserted.

Note that it is important to set the "mode" to "csv" before running the ".import" command. This is necessary to prevent the command-line shell from trying to interpret the input file text as some other format.

```
sqlite> .mode csv
sqlite> .import C:/work/somedata.csv tab1
```

There are two cases to consider: (1) Table "tab1" does not previously exist and (2) table "tab1" does already exist.

In the first case, when the table does not previously exist, the table is automatically created and the content of the first row of the input CSV file is used to determine the name of all the columns in the table. In other words, if the table does not previously exist, the first row of the CSV file is interpreted to be column names and the actual data starts on the second row of the CSV file.

For the second case, when the table already exists, every row of the CSV file, including the first row, is assumed to be actual content. If the CSV file contains an initial row of column labels, that row will be read as data and inserted into the table. To avoid this, make sure that table does not previously exist.

CSV Export

To export an SQLite table (or part of a table) as CSV, simply set the "mode" to "csv" and then run a query to extract the desired rows of the table.

```
sqlite> .header on
sqlite> .mode csv
sqlite> .once c:/work/dataout.csv
sqlite> SELECT * FROM tab1;
sqlite> .system c:/work/dataout.csv
```

In the example above, the ".header on" line causes column labels to be printed as the first row of output. This means that the first row of the resulting CSV file will contain column labels. If column labels are not desired, set ".header off" instead. (The ".header off" setting is the default and can be omitted if the headers have not been previously turned on.)

The line ".once *FILENAME*" causes all query output to go into the named file instead of being printed on the console. In the example above, that line causes the CSV content to be written into a file named "C:/work/dataout.csv".

The final line of the example (the ".system c:/work/dataout.csv") has the same effect as double-clicking on the c:/work/dataout.csv file in windows. This will typically bring up a spreadsheet program to display the CSV file. That command only works as shown on Windows. The equivalent line on a Mac would be ".system open /work/dataout.csv". On Linux and other unix systems you will need to enter something like ".system libreoffice /work/dataout.csv", substituting your preferred CSV viewing program for "libreoffice".

Converting An Entire Database To An ASCII Text File

Use the ".dump" command to convert the entire contents of a database into a single ASCII text file. This file can be converted back into a database by piping it back into **sqlite3**.

A good way to make an archival copy of a database is this:

```
$ echo '.dump' | sqlite3 ex1 | gzip -c &gt;ex1.dump.gz
```

This generates a file named **ex1.dump.gz** that contains everything you need to reconstruct the database at a later time, or on another machine. To reconstruct the database, just type:

```
$ zcat ex1.dump.gz | sqlite3 ex2
```

The text format is pure SQL so you can also use the .dump command to export an SQLite database into other popular SQL database engines. Like this:

```
$ createdb ex2
$ sqlite3 ex1 .dump | psql ex2
```

Loading Extensions

You can add new custom application-defined SQL functions, collating sequences, virtual tables, and VFSes to the command-line shell at run-time using the ".load" command. First, convert the extension in to a DLL or shared library (as described in the Run-Time Loadable Extensions document) then type:

```
sqlite> .load /path/to/my_extension
```

Note that SQLite automatically adds the appropriate extension suffix (".dll" on windows, ".dylib" on Mac, ".so" on most other unixes) to the extension filename. It is generally a good idea to specify the full pathname of the extension.

SQLite computes the entry point for the extension based on the extension filename. To override this choice, simply add the name of the extension as a second argument to the ".load" command.

Source code for several useful extensions can be found in the ext/misc subdirectory of the SQLite source tree. You can use these extensions as-is, or as a basis for creating your own custom extensions to address your own particular needs.

Other Dot Commands

There are many other dot-commands available in the command-line shell. See the ".help" command for a complete list for any particular version and build of SQLite.

Using sqlite3 in a shell script

One way to use sqlite3 in a shell script is to use "echo" or "cat" to generate a sequence of commands in a file, then invoke sqlite3 while redirecting input from the generated command file. This works fine and is appropriate in many circumstances. But as an added convenience, sqlite3 allows a single SQL command to be entered on the command line as a second argument after the database name. When the sqlite3 program is launched with two arguments, the second argument is passed to the SQLite library for processing, the query results are printed on standard output in list mode, and the program exits. This mechanism is designed to make sqlite3 easy to use in conjunction with programs like "awk". For example:

```
$ sqlite3 ex1 'select * from tbl1' |
>  awk '{printf "&lt;tr&gt;&lt;td&gt;%s&lt;td&gt;%s\n",$1,$2 }'
&lt;tr&gt;&lt;td&gt;hello&lt;td&gt;10
&lt;tr&gt;&lt;td&gt;goodbye&lt;td&gt;20
$
```

Ending shell commands

SQLite commands are normally terminated by a semicolon. In the shell you can also use the word "GO" (case-insensitive) or a slash character "/" on a line by itself to end a command. These are used by SQL Server and Oracle, respectively. These won't work in **sqlite3_exec()**, because the shell translates these into a semicolon before passing them to that function.

Compiling the sqlite3 program from sources

The source code to the sqlite3 command line interface is in a single file named "shell.c" which you can download from the SQLite website. Compile this file (together with the sqlite3 library source code) to generate the executable. For example:

```
gcc -o sqlite3 shell.c sqlite3.c -ldl -lpthread
```

The Error And Warning Log

SQLite can be configured to invoke a callback function containing an error code and a terse error message whenever anomalies occur. This mechanism is very helpful in tracking obscure problems that occur rarely and in the field. Application developers are encouraged to take advantage of the error logging facility of SQLite in their products, as it is very low CPU and memory cost but can be a huge aid for debugging.

Setting Up The Error Logging Callback

There can only be a single error logging callback per process. The error logging callback is registered at start-time using C-code similar to the following:

```
sqlite3_config(SQLITE_CONFIG_LOG, errLogCallback, pData);
```

The error logger callback function might look something like this:

```
void errorLogCallback(void *pArg, int iErrCode, const char *zMsg){
  fprintf(stderr, "(%d) %s\n", iErrCode, zMsg);
}
```

The example above illustrates the signature of the error logger callback. However, in an embedded application, one usually does not print messages on stderr. Instead, one might store the messages in a preallocated circular buffer where they can be accessed when diagnostic information is needed during debugging. Or perhaps the messages can be sent to Syslog. Somehow, the messages need to be stored where they are accessible to developers, not displayed to end users.

Do not misunderstand: There is nothing technically wrong with displaying the error logger messages to end users. The messages do not contain sensitive or private information that must be protected from unauthorized viewing. Rather the messages are technical in nature and are not useful or meaningful to the typical end user. The messages coming from the error logger are intended for database geeks. Display them accordingly.

Interface Details

The third argument to the sqlite3_config(SQLITE_CONFIG_LOG,...) interface (the "pData" argument in the example above) is a pointer to arbitrary data. SQLite passes this pointer through to the first argument of the error logger callback. The pointer can be used to pass application-specific setup or state information, if desired. Or it can simply be a NULL pointer which is ignored by the callback.

The second argument to the error logger callback is an integer extended error code. The third argument to the error logger is the text of the error message. The error message text is stored in a fixed-length stack buffer in the calling function and so will only be valid for the duration of the error logger callback function. The error logger should make a copy of this message into persistent storage if retention of the message is needed.

The error logger callback should be treated like a signal handler. The application should save off or otherwise process the error, then return as soon as possible. No other SQLite APIs should be invoked, directly or indirectly, from the error logger. SQLite is not reentrant through the error logger callback. In particular, the error logger callback is invoked when a memory allocation fails, so it is generally a bad idea to try to allocate memory inside the error logger. Do not even think about trying to store the error message in another SQLite database.

Applications can use the sqlite3_log(E,F,..) API to send new messages to the log, if desired, but this is discouraged. The sqlite3_log() interface is intended for use by extensions only, not by applications.

Variety of Error Messages

The error messages that might be sent to the error logger and their exact format is subject to changes from one release to the next. So applications should not depend on any particular error message text formats or error codes. Things do not change capriciously, but they do sometimes changes.

The following is a partial list of the kinds of messages that might appear in the error logger callback.

- Any time there is an error either compiling an SQL statement (using sqlite3_prepare_v2() or its siblings) or running an SQL statement (using sqlite3_step()) that error is logged.
- When a schema change occurs that requires a prepared statement to be reparsed and reprepared, that event is logged with the error code SQLITE_SCHEMA. The reparse and reprepare is normally automatic (assuming that sqlite3_prepare_v2() has been used to prepared the statements originally, which is recommended) and so these logging events are normally the only way to know that reprepares are taking place.

- SQLITE_NOTICE messages are logged whenever a database has to be recovered because the previous writer crashed without completing its transaction. The error code is SQLITE_NOTICE_RECOVER_ROLLBACK when recovering a rollback journal and SQLITE_NOTICE_RECOVER_WAL when recovering a write-ahead log.
- SQLITE_WARNING messages are logged when database files are renamed or aliased in ways that can lead to database corruption. (See 1 and 2 for additional information.)
- Out of memory (OOM) error conditions generate error logging events with the SQLITE_NOMEM error code and a message that says how many bytes of memory were requested by the failed allocation.
- I/O errors in the OS-interface generate error logging events. The message to these events gives the line number in the source code where the error originated and the filename associated with the event when there is a corresponding file.
- When database corruption is detected, an SQLITE_CORRUPT error logger callback is invoked. As with I/O errors, the error message text contains the line number in the original source code where the error was first detected.
- An error logger callback is invoked on SQLITE_MISUSE errors. This is useful in detecting application design issues when return codes are not consistently checked in the application code.

SQLite strives to keep error logger traffic low and only send messages to the error logger when there really is something wrong. Applications might further cull the error message traffic by deliberately ignore certain classes of error messages that they do not care about. For example, an application that makes frequent database schema changes might want to ignore all SQLITE_SCHEMA errors.

Summary

The use of the error logger callback is highly recommended. The debugging information that the error logger provides has proven very useful in tracking down obscure problems that occurs with applications after they get into the field. The error logger callback has also proven useful in catching errors occasional errors that the application misses because of inconsistent checking of API return codes. Developers are encouraged to implement an error logger callback early in the development cycle in order to spot unexpected behavior quickly, and to leave the error logger callback turned on through deployment. If the error logger never finds a problem, then no harm is done. But failure to set up an appropriate error logger might compromise diagnostic capabilities later on.

SQLite Foreign Key Support

Overview

This document describes the support for SQL foreign key constraints introduced in SQLite version 3.6.19.

The first section introduces the concept of an SQL foreign key by example and defines the terminology used for the remainder of the document. Section 2 describes the steps an application must take in order to enable foreign key constraints in SQLite (it is disabled by default). The next section, section 3, describes the indexes that the user must create in order to use foreign key constraints, and those that should be created in order for foreign key constraints to function efficiently. Section 4 describes the advanced foreign key related features supported by SQLite and section 5 describes the way the ALTER and DROP TABLE commands are enhanced to support foreign key constraints. Finally, section 6 enumerates the missing features and limits of the current implementation.

This document does not contain a full description of the syntax used to create foreign key constraints in SQLite. This may be found as part of the documentation for the CREATE TABLE statement.

1. Introduction to Foreign Key Constraints

SQL foreign key constraints are used to enforce "exists" relationships between tables. For example, consider a database schema created using the following SQL commands:

```
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER     -- Must map to an artist.artistid!
);
```

The applications using this database are entitled to assume that for each row in the *track* table there exists a corresponding row in the *artist* table. After all, the comment in the declaration says so. Unfortunately, if a user edits the database using an external tool or if there is a bug in an application, rows might be inserted into the *track* table that do not correspond to any row in the *artist* table. Or rows might be deleted from the *artist* table, leaving orphaned rows in the *track* table that do not correspond to any of the remaining rows in *artist*. This might cause the application or applications to malfunction later on, or at least make coding the application more difficult.

One solution is to add an SQL foreign key constraint to the database schema to enforce the relationship between the *artist* and *track* table. To do so, a foreign key definition may be added by modifying the declaration of the *track* table to the following:

```
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER,
  FOREIGN KEY(trackartist) REFERENCES artist(artistid)
);
```

This way, the constraint is enforced by SQLite. Attempting to insert a row into the *track* table that does not correspond to any row in the *artist* table will fail, as will attempting to delete a row from the *artist* table when there exist dependent rows in the *track* table There is one exception: if the foreign key column in the *track* table is NULL, then no corresponding entry in the *artist* table is required. Expressed in SQL, this means that for every row in the *track* table, the following expression evaluates to true:

```
trackartist IS NULL OR EXISTS(SELECT 1 FROM artist WHERE artistid=trackartist)
```

Tip: If the application requires a stricter relationship between *artist* and *track*, where NULL values are not permitted in the *trackartist* column, simply add the appropriate "NOT NULL" constraint to the schema.

There are several other ways to add an equivalent foreign key declaration to a CREATE TABLE statement. Refer to the CREATE TABLE documentation for details.

The following SQLite command-line session illustrates the effect of the foreign key constraint added to the *track* table:

```
sqlite> SELECT * FROM artist;
artistid  artistname       
--------  -----------------
1         Dean Martin      
2         Frank Sinatra    

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
11       That's Amore       1          
12       Christmas Blues    1          
13       My Way             2          

sqlite> -- This fails because the value inserted into the trackartist column (3)
sqlite> -- does not correspond to row in the artist table.
sqlite> INSERT INTO track VALUES(14, 'Mr. Bojangles', 3);
SQL error: foreign key constraint failed

sqlite> -- This succeeds because a NULL is inserted into trackartist. A
sqlite> -- corresponding row in the artist table is not required in this case.
sqlite> INSERT INTO track VALUES(14, 'Mr. Bojangles', NULL);

sqlite> -- Trying to modify the trackartist field of the record after it has
sqlite> -- been inserted does not work either, since the new value of trackartist (3)
sqlite> -- Still does not correspond to any row in the artist table.
sqlite> UPDATE track SET trackartist = 3 WHERE trackname = 'Mr. Bojangles';
SQL error: foreign key constraint failed

sqlite> -- Insert the required row into the artist table. It is then possible to
sqlite> -- update the inserted row to set trackartist to 3 (since a corresponding
sqlite> -- row in the artist table now exists).
sqlite> INSERT INTO artist VALUES(3, 'Sammy Davis Jr.');
sqlite> UPDATE track SET trackartist = 3 WHERE trackname = 'Mr. Bojangles';

sqlite> -- Now that "Sammy Davis Jr." (artistid = 3) has been added to the database,
sqlite> -- it is possible to INSERT new tracks using this artist without violating
sqlite> -- the foreign key constraint:
sqlite> INSERT INTO track VALUES(15, 'Boogie Woogie', 3);
```

As you would expect, it is not possible to manipulate the database to a state that violates the foreign key constraint by deleting or updating rows in the *artist* table either:

```
sqlite> -- Attempting to delete the artist record for "Frank Sinatra" fails, since
sqlite> -- the track table contains a row that refer to it.
sqlite> DELETE FROM artist WHERE artistname = 'Frank Sinatra';
SQL error: foreign key constraint failed

sqlite> -- Delete all the records from the track table that refer to the artist
sqlite> -- "Frank Sinatra". Only then is it possible to delete the artist.
sqlite> DELETE FROM track WHERE trackname = 'My Way';
sqlite> DELETE FROM artist WHERE artistname = 'Frank Sinatra';

sqlite> -- Try to update the artistid of a row in the artist table while there
sqlite> -- exists records in the track table that refer to it.
sqlite> UPDATE artist SET artistid=4 WHERE artistname = 'Dean Martin';
SQL error: foreign key constraint failed

sqlite> -- Once all the records that refer to a row in the artist table have
sqlite> -- been deleted, it is possible to modify the artistid of the row.
sqlite> DELETE FROM track WHERE trackname IN('That''s Amore', 'Christmas Blues');
sqlite> UPDATE artist SET artistid=4 WHERE artistname = 'Dean Martin';
```

SQLite uses the following terminology:

- The **parent table** is the table that a foreign key constraint refers to. The parent table in the example in this section is the *artist* table. Some books and articles refer to this as the *referenced table*, which is arguably more correct, but tends to lead to confusion.
- The **child table** is the table that a foreign key constraint is applied to and the table that contains the REFERENCES clause. The example in this section uses the *track* table as the child table. Other books and articles refer to this as the *referencing table*.
- The **parent key** is the column or set of columns in the parent table that the foreign key constraint refers to. This is normally, but not always, the primary key of the parent table. The parent key must be a named column or columns in the parent table, not the rowid.
- The **child key** is the column or set of columns in the child table that are constrained by the foreign key constraint and which hold the REFERENCES clause.

The foreign key constraint is satisfied if for each row in the child table either one or more of the child key columns are NULL, or there exists a row in the parent table for which each parent key column contains a value equal to the value in its associated child key column.

In the above paragraph, the term "equal" means equal when values are compared using the rules specified here. The following clarifications apply:

- When comparing text values, the collating sequence associated with the parent key column is always used.
- When comparing values, if the parent key column has an affinity, then that affinity is applied to the child key value before the comparison is performed.

2. Enabling Foreign Key Support

In order to use foreign key constraints in SQLite, the library must be compiled with neither SQLITE_OMIT_FOREIGN_KEY or SQLITE_OMIT_TRIGGER defined. If SQLITE_OMIT_TRIGGER is defined but SQLITE_OMIT_FOREIGN_KEY is not, then SQLite behaves as it did prior to version 3.6.19 - foreign key definitions are parsed and may be queried using PRAGMA foreign_key_list, but foreign key constraints are not enforced. The PRAGMA foreign_keys command is a no-op in this configuration. If OMIT_FOREIGN_KEY is defined, then foreign key definitions cannot even be parsed (attempting to specify a foreign key definition is a syntax error).

Assuming the library is compiled with foreign key constraints enabled, it must still be enabled by the application at runtime, using the PRAGMA foreign_keys command. For example:

```
sqlite> PRAGMA foreign_keys = ON;
```

Foreign key constraints are disabled by default (for backwards compatibility), so must be enabled separately for each database connection. (Note, however, that future releases of SQLite might change so that foreign key constraints enabled by default. Careful developers will not make any assumptions about whether or not foreign keys are enabled by default but will instead enable or disable them as necessary.) The application can also use a PRAGMA foreign_keys statement to determine if foreign keys are currently enabled. The following command-line session demonstrates this:

```
sqlite> PRAGMA foreign_keys;
0
sqlite> PRAGMA foreign_keys = ON;
sqlite> PRAGMA foreign_keys;
1
sqlite> PRAGMA foreign_keys = OFF;
sqlite> PRAGMA foreign_keys;
0
```

Tip: If the command "PRAGMA foreign_keys" returns no data instead of a single row containing "0" or "1", then the version of SQLite you are using does not support foreign keys (either because it is older than 3.6.19 or because it was compiled with SQLITE_OMIT_FOREIGN_KEY or SQLITE_OMIT_TRIGGER defined).

It is not possible to enable or disable foreign key constraints in the middle of a multi-statement transaction (when SQLite is not in autocommit mode). Attempting to do so does not return an error; it simply has no effect.

3. Required and Suggested Database Indexes

Usually, the parent key of a foreign key constraint is the primary key of the parent table. If they are not the primary key, then the parent key columns must be collectively subject to a UNIQUE constraint or have a UNIQUE index. If the parent key columns have a UNIQUE index, then that index must use the collation sequences that are specified in the CREATE TABLE statement for the parent table. For example,

```
CREATE TABLE parent(a PRIMARY KEY, b UNIQUE, c, d, e, f);
CREATE UNIQUE INDEX i1 ON parent(c, d);
CREATE INDEX i2 ON parent(e);
CREATE UNIQUE INDEX i3 ON parent(f COLLATE nocase);

CREATE TABLE child1(f, g REFERENCES parent(a));                        -- Ok
CREATE TABLE child2(h, i REFERENCES parent(b));                        -- Ok
CREATE TABLE child3(j, k, FOREIGN KEY(j, k) REFERENCES parent(c, d));  -- Ok
CREATE TABLE child4(l, m REFERENCES parent(e));                        -- Error!
CREATE TABLE child5(n, o REFERENCES parent(f));                        -- Error!
CREATE TABLE child6(p, q, FOREIGN KEY(p, q) REFERENCES parent(b, c));  -- Error!
CREATE TABLE child7(r REFERENCES parent(c));                           -- Error!
```

The foreign key constraints created as part of tables *child1*, *child2* and *child3* are all fine. The foreign key declared as part of table *child4* is an error because even though the parent key column is indexed, the index is not UNIQUE. The foreign key for table *child5* is an error because even though the parent key column has a unique index, the index uses a different collating sequence. Tables *child6* and *child7* are incorrect because while both have UNIQUE indices on their parent keys, the keys are not an exact match to the columns of a single UNIQUE index.

If the database schema contains foreign key errors that require looking at more than one table definition to identify, then those errors are not detected when the tables are created. Instead, such errors prevent the application from preparing SQL statements that modify the content of the child or parent tables in ways that use the foreign keys. Errors reported when content is changed are "DML errors" and errors reported when the schema is changed are "DDL errors". So, in other words, misconfigured foreign key constraints that require looking at both the child and parent are DML errors. The English language error message for foreign key DML errors is usually "foreign key mismatch" but can also be "no such table" if the parent table does not exist. Foreign key DML errors are may be reported if:

- The parent table does not exist, or
- The parent key columns named in the foreign key constraint do not exist, or
- The parent key columns named in the foreign key constraint are not the primary key of the parent table and are not subject to a unique constraint using collating sequence specified in the CREATE TABLE, or
- The child table references the primary key of the parent without specifying the primary key columns and the number of primary key columns in the parent do not match the number of child key columns.

The last bullet above is illustrated by the following:

```
CREATE TABLE parent2(a, b, PRIMARY KEY(a,b));

CREATE TABLE child8(x, y, FOREIGN KEY(x,y) REFERENCES parent2);        -- Ok
CREATE TABLE child9(x REFERENCES parent2);                             -- Error!
CREATE TABLE child10(x,y,z, FOREIGN KEY(x,y,z) REFERENCES parent2);    -- Error!
```

By contrast, if foreign key errors can be recognized simply by looking at the definition of the child table and without having to consult the parent table definition, then the CREATE TABLE statement for the child table fails. Because the error occurs during a schema change, this is a DDL error. Foreign key DDL errors are reported regardless of whether or not foreign key constraints are enabled when the table is created.

Indices are not required for child key columns but they are almost always beneficial. Returning to the example in section 1, each time an application deletes a row from the *artist* table (the parent table), it performs the equivalent of the following SELECT statement to search for referencing rows in the *track* table (the child table).

```
SELECT rowid FROM track WHERE trackartist = ?
```

where ? in the above is replaced with the value of the *artistid* column of the record being deleted from the *artist* table (recall that the *trackartist* column is the child key and the *artistid* column is the parent key). Or, more generally:

```
SELECT rowid FROM &lt;child-table&gt; WHERE &lt;child-key&gt; = :parent_key_value
```

If this SELECT returns any rows at all, then SQLite concludes that deleting the row from the parent table would violate the foreign key constraint and returns an error. Similar queries may be run if the content of the parent key is modified or a new row is inserted into the parent table. If these queries cannot use an index, they are forced to do a linear scan of the entire child table. In a non-trivial database, this may be prohibitively expensive.

So, in most real systems, an index should be created on the child key columns of each foreign key constraint. The child key index does not have to be (and usually will not be) a UNIQUE index. Returning again to the example in section 1, the complete database schema for efficient implementation of the foreign key constraint might be:

```
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER REFERENCES artist
);
CREATE INDEX trackindex ON track(trackartist);
```

The block above uses a shorthand form to create the foreign key constraint. Attaching a "REFERENCES *<parent-table>*" clause to a column definition creates a foreign key constraint that maps the column to the primary key of *<parent-table>*. Refer to the CREATE TABLE documentation for further details.

4. Advanced Foreign Key Constraint Features

4.1. Composite Foreign Key Constraints

A composite foreign key constraint is one where the child and parent keys are both composite keys. For example, consider the following database schema:

```
CREATE TABLE album(
  albumartist TEXT,
  albumname TEXT,
  albumcover BINARY,
  PRIMARY KEY(albumartist, albumname)
);

CREATE TABLE song(
  songid     INTEGER,
  songartist TEXT,
  songalbum TEXT,
  songname   TEXT,
  FOREIGN KEY(songartist, songalbum) REFERENCES album(albumartist, albumname)
);
```

In this system, each entry in the song table is required to map to an entry in the album table with the same combination of artist and album.

Parent and child keys must have the same cardinality. In SQLite, if any of the child key columns (in this case songartist and songalbum) are NULL, then there is no requirement for a corresponding row in the parent table.

4.2. Deferred Foreign Key Constraints

Each foreign key constraint in SQLite is classified as either immediate or deferred. Foreign key constraints are immediate by default. All the foreign key examples presented so far have been of immediate foreign key constraints.

If a statement modifies the contents of the database so that an immediate foreign key constraint is in violation at the conclusion the statement, an exception is thrown and the effects of the statement are reverted. By contrast, if a statement modifies the contents of the database such that a deferred foreign key constraint is violated, the violation is not reported immediately. Deferred foreign key constraints are not checked until the transaction tries to COMMIT. For as long as the user has an open transaction, the database is allowed to exist in a state that violates any number of deferred foreign key constraints. However, COMMIT will fail as long as foreign key constraints remain in violation.

If the current statement is not inside an explicit transaction (a BEGIN/COMMIT/ROLLBACK block), then an implicit transaction is committed as soon as the statement has finished executing. In this case deferred constraints behave the same as immediate constraints.

To mark a foreign key constraint as deferred, its declaration must include the following clause:

```
DEFERRABLE INITIALLY DEFERRED                -- A deferred foreign key constraint
```

The full syntax for specifying foreign key constraints is available as part of the CREATE TABLE documentation. Replacing the phrase above with any of the following creates an immediate foreign key constraint.

```
NOT DEFERRABLE INITIALLY DEFERRED            -- An immediate foreign key constraint
NOT DEFERRABLE INITIALLY IMMEDIATE           -- An immediate foreign key constraint
NOT DEFERRABLE                               -- An immediate foreign key constraint
DEFERRABLE INITIALLY IMMEDIATE               -- An immediate foreign key constraint
DEFERRABLE                                   -- An immediate foreign key constraint
```

The defer_foreign_keys pragma can be used to temporarily change all foreign key constraints to deferred regardless of how they are declared.

The following example illustrates the effect of using a deferred foreign key constraint.

```
-- Database schema. Both tables are initially empty.
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER REFERENCES artist(artistid) DEFERRABLE INITIALLY DEFERRED
);

sqlite3&gt; -- If the foreign key constraint were immediate, this INSERT would
sqlite3&gt; -- cause an error (since as there is no row in table artist with
sqlite3&gt; -- artistid=5). But as the constraint is deferred and there is an
sqlite3&gt; -- open transaction, no error occurs.
sqlite3&gt; BEGIN;
sqlite3&gt;   INSERT INTO track VALUES(1, 'White Christmas', 5);

sqlite3&gt; -- The following COMMIT fails, as the database is in a state that
sqlite3&gt; -- does not satisfy the deferred foreign key constraint. The
sqlite3&gt; -- transaction remains open.
sqlite3&gt; COMMIT;
SQL error: foreign key constraint failed

sqlite3&gt; -- After inserting a row into the artist table with artistid=5, the
sqlite3&gt; -- deferred foreign key constraint is satisfied. It is then possible
sqlite3&gt; -- to commit the transaction without error.
sqlite3&gt;   INSERT INTO artist VALUES(5, 'Bing Crosby');
sqlite3&gt; COMMIT;
```

A nested savepoint transaction may be RELEASEd while the database is in a state that does not satisfy a deferred foreign key constraint. A transaction savepoint (a non-nested savepoint that was opened while there was not currently an open transaction), on the other hand, is subject to the same restrictions as a COMMIT - attempting to RELEASE it while the database is in such a state will fail.

If a COMMIT statement (or the RELEASE of a transaction SAVEPOINT) fails because the database is currently in a state that violates a deferred foreign key constraint and there are currently nested savepoints, the nested savepoints remain open.

4.3. ON DELETE and ON UPDATE Actions

Foreign key ON DELETE and ON UPDATE clauses are used to configure actions that take place when deleting rows from the parent table (ON DELETE), or modifying the parent key values of existing rows (ON UPDATE). A single foreign key constraint may have different actions configured for ON DELETE and ON UPDATE. Foreign key actions are similar to triggers in many ways.

The ON DELETE and ON UPDATE action associated with each foreign key in an SQLite database is one of "NO ACTION", "RESTRICT", "SET NULL", "SET DEFAULT" or "CASCADE". If an action is not explicitly specified, it defaults to "NO ACTION".

- **NO ACTION**: Configuring "NO ACTION" means just that: when a parent key is modified or deleted from the database, no special action is taken.
- **RESTRICT**: The "RESTRICT" action means that the application is prohibited from deleting (for ON DELETE RESTRICT) or modifying (for ON UPDATE RESTRICT) a parent key when there exists one or more child keys mapped to it. The difference between the effect of a RESTRICT action and normal foreign key constraint enforcement is that the RESTRICT action processing happens as soon as the field is updated - not at the end of the current statement as it would with an immediate constraint, or at the end of the current transaction as it would with a deferred constraint. Even if the foreign key constraint it is attached to is deferred, configuring a RESTRICT action causes SQLite to return an error immediately if a parent key with dependent child keys is deleted or modified.
- **SET NULL**: If the configured action is "SET NULL", then when a parent key is deleted (for ON DELETE SET NULL) or modified (for ON UPDATE SET NULL), the child key columns of all rows in the child table that mapped to the parent key are set to contain SQL NULL values.
- **SET DEFAULT**: The "SET DEFAULT" actions are similar to "SET NULL", except that each of the child key columns is set to contain the columns default value instead of NULL. Refer to the CREATE TABLE documentation for details on how default values are assigned to table columns.
- **CASCADE**: A "CASCADE" action propagates the delete or update operation on the parent key to each dependent child key. For an "ON DELETE CASCADE" action, this means that each row in the child table that was associated with the deleted parent row is also deleted. For an "ON UPDATE CASCADE" action, it means that the values stored in each dependent child key are modified to match the new parent key values.

For example, adding an "ON UPDATE CASCADE" clause to the foreign key as shown below enhances the example schema from section 1 to allow the user to update the artistid (the parent key of the foreign key constraint) column without breaking referential integrity:

```
-- Database schema
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER REFERENCES artist(artistid) ON UPDATE CASCADE
);

sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
1         Dean Martin
2         Frank Sinatra

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
11       That's Amore       1
12       Christmas Blues    1
13       My Way             2

sqlite> -- Update the artistid column of the artist record for "Dean Martin".
sqlite> -- Normally, this would raise a constraint, as it would orphan the two
sqlite> -- dependent records in the track table. However, the ON UPDATE CASCADE clause
sqlite> -- attached to the foreign key definition causes the update to "cascade"
sqlite> -- to the child table, preventing the foreign key constraint violation.
sqlite> UPDATE artist SET artistid = 100 WHERE artistname = 'Dean Martin';

sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
2         Frank Sinatra
100       Dean Martin

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
11       That's Amore       100
12       Christmas Blues    100
13       My Way             2
```

Configuring an ON UPDATE or ON DELETE action does not mean that the foreign key constraint does not need to be satisfied. For example, if an "ON DELETE SET DEFAULT" action is configured, but there is no row in the parent table that corresponds to the default values of the child key columns, deleting a parent key while dependent child keys exist still causes a foreign key violation. For example:

```
-- Database schema
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER DEFAULT 0 REFERENCES artist(artistid) ON DELETE SET DEFAULT
);

sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
3         Sammy Davis Jr.

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
14       Mr. Bojangles      3

sqlite> -- Deleting the row from the parent table causes the child key
sqlite> -- value of the dependent row to be set to integer value 0\. However, this
sqlite> -- value does not correspond to any row in the parent table. Therefore
sqlite> -- the foreign key constraint is violated and an is exception thrown.
sqlite> DELETE FROM artist WHERE artistname = 'Sammy Davis Jr.';
SQL error: foreign key constraint failed

sqlite> -- This time, the value 0 does correspond to a parent table row. And
sqlite> -- so the DELETE statement does not violate the foreign key constraint
sqlite> -- and no exception is thrown.
sqlite> INSERT INTO artist VALUES(0, 'Unknown Artist');
sqlite> DELETE FROM artist WHERE artistname = 'Sammy Davis Jr.';

sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
0         Unknown Artist

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
14       Mr. Bojangles      0
```

Those familiar with SQLite triggers will have noticed that the "ON DELETE SET DEFAULT" action demonstrated in the example above is similar in effect to the following AFTER DELETE trigger:

```
CREATE TRIGGER on_delete_set_default AFTER DELETE ON artist BEGIN
  UPDATE child SET trackartist = 0 WHERE trackartist = old.artistid;
END;
```

Whenever a row in the parent table of a foreign key constraint is deleted, or when the values stored in the parent key column or columns are modified, the logical sequence of events is:

1. Execute applicable BEFORE trigger programs,
2. Check local (non foreign key) constraints,
3. Update or delete the row in the parent table,
4. Perform any required foreign key actions,
5. Execute applicable AFTER trigger programs.

There is one important difference between ON UPDATE foreign key actions and SQL triggers. An ON UPDATE action is only taken if the values of the parent key are modified so that the new parent key values are not equal to the old. For example:

```
-- Database schema
CREATE TABLE parent(x PRIMARY KEY);
CREATE TABLE child(y REFERENCES parent ON UPDATE SET NULL);

sqlite> SELECT * FROM parent;
x
----
key

sqlite> SELECT * FROM child;
y
----
key

sqlite> -- Since the following UPDATE statement does not actually modify
sqlite> -- the parent key value, the ON UPDATE action is not performed and
sqlite> -- the child key value is not set to NULL.
sqlite> UPDATE parent SET x = 'key';
sqlite> SELECT IFNULL(y, 'null') FROM child;
y
----
key

sqlite> -- This time, since the UPDATE statement does modify the parent key
sqlite> -- value, the ON UPDATE action is performed and the child key is set
sqlite> -- to NULL.
sqlite> UPDATE parent SET x = 'key2';
sqlite> SELECT IFNULL(y, 'null') FROM child;
y
----
null
```

5. CREATE, ALTER and DROP TABLE commands

This section describes the way the CREATE TABLE, ALTER TABLE, and DROP TABLE commands interact with SQLite's foreign keys.

A CREATE TABLE command operates the same whether or not foreign key constraints are enabled. The parent key definitions of foreign key constraints are not checked when a table is created. There is nothing stopping the user from creating a foreign key definition that refers to a parent table that does not exist, or to parent key columns that do not exist or are not collectively bound by a PRIMARY KEY or UNIQUE constraint.

The ALTER TABLE command works differently in two respects when foreign key constraints are enabled:

- It is not possible to use the "ALTER TABLE ... ADD COLUMN" syntax to add a column that includes a REFERENCES clause, unless the default value of the new column is NULL. Attempting to do so returns an error.

- If an "ALTER TABLE ... RENAME TO" command is used to rename a table that is the parent table of one or more foreign key constraints, the definitions of the foreign key constraints are modified to refer to the parent table by its new name. The text of the child CREATE TABLE statement or statements stored in the sqlite_master table are modified to reflect the new parent table name.

If foreign key constraints are enabled when it is prepared, the DROP TABLE command performs an implicit DELETE to remove all rows from the table before dropping it. The implicit DELETE does not cause any SQL triggers to fire, but may invoke foreign key actions or constraint violations. If an immediate foreign key constraint is violated, the DROP TABLE statement fails and the table is not dropped. If a deferred foreign key constraint is violated, then an error is reported when the user attempts to commit the transaction if the foreign key constraint violations still exist at that point. Any "foreign key mismatch" errors encountered as part of an implicit DELETE are ignored.

The intent of these enhancements to the ALTER TABLE and DROP TABLE commands is to ensure that they cannot be used to create a database that contains foreign key violations, at least while foreign key constraints are enabled. There is one exception to this rule though. If a parent key is not subject to a PRIMARY KEY or UNIQUE constraint created as part of the parent table definition, but is subject to a UNIQUE constraint by virtue of an index created using the CREATE INDEX command, then the child table may be populated without causing a "foreign key mismatch" error. If the UNIQUE index is dropped from the database schema, then the parent table itself is dropped, no error will be reported. However the database may be left in a state where the child table of the foreign key constraint contains rows that do not refer to any parent table row. This case can be avoided if all parent keys in the database schema are constrained by PRIMARY KEY or UNIQUE constraints added as part of the parent table definition, not by external UNIQUE indexes.

The properties of the DROP TABLE and ALTER TABLE commands described above only apply if foreign keys are enabled. If the user considers them undesirable, then the workaround is to use PRAGMA foreign_keys to disable foreign key constraints before executing the DROP or ALTER TABLE command. Of course, while foreign key constraints are disabled, there is nothing to stop the user from violating foreign key constraints and thus creating an internally inconsistent database.

6. Limits and Unsupported Features

This section lists a few limitations and omitted features that are not mentioned elsewhere.

1. **No support for the MATCH clause.** According to SQL92, a MATCH clause may be attached to a composite foreign key definition to modify the way NULL values that occur in child keys are handled. If "MATCH SIMPLE" is specified, then a child key is not

required to correspond to any row of the parent table if one or more of the child key values are NULL. If "MATCH FULL" is specified, then if any of the child key values is NULL, no corresponding row in the parent table is required, but all child key values must be NULL. Finally, if the foreign key constraint is declared as "MATCH PARTIAL" and one of the child key values is NULL, there must exist at least one row in the parent table for which the non-NULL child key values match the parent key values.

SQLite parses MATCH clauses (i.e. does not report a syntax error if you specify one), but does not enforce them. All foreign key constraints in SQLite are handled as if MATCH SIMPLE were specified.

2. **No support for switching constraints between deferred and immediate mode.** Many systems allow the user to toggle individual foreign key constraints between deferred and immediate mode at runtime (for example using the Oracle "SET CONSTRAINT" command). SQLite does not support this. In SQLite, a foreign key constraint is permanently marked as deferred or immediate when it is created.

3. **Recursion limit on foreign key actions.** The SQLITE_MAX_TRIGGER_DEPTH and SQLITE_LIMIT_TRIGGER_DEPTH settings determine the maximum allowable depth of trigger program recursion. For the purposes of these limits, foreign key actions are considered trigger programs. The PRAGMA recursive_triggers setting does not affect the operation of foreign key actions. It is not possible to disable recursive foreign key actions.

SQLite FTS3 and FTS4 Extensions

Overview

FTS3 and FTS4 are SQLite virtual table modules that allows users to perform full-text searches on a set of documents. The most common (and effective) way to describe full-text searches is "what Google, Yahoo, and Bing do with documents placed on the World Wide Web". Users input a term, or series of terms, perhaps connected by a binary operator or grouped together into a phrase, and the full-text query system finds the set of documents that best matches those terms considering the operators and groupings the user has specified. This article describes the deployment and usage of FTS3 and FTS4.

FTS1 and FTS2 are obsolete full-text search modules for SQLite. There are known issues with these older modules and their use should be avoided. Portions of the original FTS3 code were contributed to the SQLite project by Scott Hess of Google. It is now developed and maintained as part of SQLite.

1. Introduction to FTS3 and FTS4

The FTS3 and FTS4 extension modules allows users to create special tables with a built-in full-text index (hereafter "FTS tables"). The full-text index allows the user to efficiently query the database for all rows that contain one or more words (hereafter "tokens"), even if the table contains many large documents.

For example, if each of the 517430 documents in the "Enron E-Mail Dataset" is inserted into both an FTS table and an ordinary SQLite table created using the following SQL script:

```
CREATE VIRTUAL TABLE enrondata1 USING fts3(content TEXT);     /* FTS3 table */
CREATE TABLE enrondata2(content TEXT);                        /* Ordinary table */
```

Then either of the two queries below may be executed to find the number of documents in the database that contain the word "linux" (351). Using one desktop PC hardware configuration, the query on the FTS3 table returns in approximately 0.03 seconds, versus 22.5 for querying the ordinary table.

```
SELECT count(*) FROM enrondata1 WHERE content MATCH 'linux';  /* 0.03 seconds */
SELECT count(*) FROM enrondata2 WHERE content LIKE '%linux%'; /* 22.5 seconds */
```

Of course, the two queries above are not entirely equivalent. For example the LIKE query matches rows that contain terms such as "linuxophobe" or "EnterpriseLinux" (as it happens, the Enron E-Mail Dataset does not actually contain any such terms), whereas the MATCH query on the FTS3 table selects only those rows that contain "linux" as a discrete token. Both searches are case-insensitive. The FTS3 table consumes around 2006 MB on disk compared to just 1453 MB for the ordinary table. Using the same hardware configuration used to perform the SELECT queries above, the FTS3 table took just under 31 minutes to populate, versus 25 for the ordinary table.

1.1. Differences between FTS3 and FTS4

FTS3 and FTS4 are nearly identical. They share most of their code in common, and their interfaces are the same. The differences are:

- FTS4 contains query performance optimizations that may significantly improve the performance of full-text queries that contain terms that are very common (present in a large percentage of table rows).
- FTS4 supports some additional options that may used with the matchinfo() function.
- Because it stores extra information on disk in two new shadow tables in order to support the performance optimizations and extra matchinfo() options, FTS4 tables may consume more disk space than the equivalent table created using FTS3. Usually the overhead is 1-2% or less, but may be as high as 10% if the documents stored in the FTS table are very small. The overhead may be reduced by specifying the directive "matchinfo=fts3" as part of the FTS4 table declaration, but this comes at the expense of sacrificing some of the extra supported matchinfo() options.
- FTS4 provides hooks (the compress and uncompress options) allowing data to be stored in a compressed form, reducing disk usage and IO.

FTS4 is an enhancement to FTS3. FTS3 has been available since SQLite version 3.5.0 in 2007-09-04. The enhancements for FTS4 were added with SQLite version 3.7.4 on 2010-12-08.

Which module, FTS3 or FTS4, should you use in your application? FTS4 is sometimes significantly faster than FTS3, even orders of magnitude faster depending on the query, though in the common case the performance of the two modules is similar. FTS4 also offers the enhanced matchinfo() outputs which can be useful in ranking the results of a MATCH operation. On the other hand, in the absence of a matchinfo=fts3 directive FTS4 requires a little more disk space than FTS3, though only a percent of two in most cases.

For newer applications, FTS4 is recommended; though if compatibility with older versions of SQLite is important, then FTS3 will usually serve just as well.

1.2. Creating and Destroying FTS Tables

Like other virtual table types, new FTS tables are created using a CREATE VIRTUAL TABLE statement. The module name, which follows the USING keyword, is either "fts3" or "fts4". The virtual table module arguments may be left empty, in which case an FTS table with a single user-defined column named "content" is created. Alternatively, the module arguments may be passed a list of comma separated column names.

If column names are explicitly provided for the FTS table as part of the CREATE VIRTUAL TABLE statement, then a datatype name may be optionally specified for each column. This is pure syntactic sugar, the supplied typenames are not used by FTS or the SQLite core for any purpose. The same applies to any constraints specified along with an FTS column name - they are parsed but not used or recorded by the system in any way.

```
-- Create an FTS table named "data" with one column - "content":
CREATE VIRTUAL TABLE data USING fts3();

-- Create an FTS table named "pages" with three columns:
CREATE VIRTUAL TABLE pages USING fts4(title, keywords, body);

-- Create an FTS table named "mail" with two columns. Datatypes
-- and column constraints are specified along with each column. These
-- are completely ignored by FTS and SQLite.
CREATE VIRTUAL TABLE mail USING fts3(
  subject VARCHAR(256) NOT NULL,
  body TEXT CHECK(length(body)&lt;10240)
);
```

As well as a list of columns, the module arguments passed to a CREATE VIRTUAL TABLE statement used to create an FTS table may be used to specify a tokenizer. This is done by specifying a string of the form "tokenize=<tokenizer name> <tokenizer args>" in place of a column name, where <tokenizer name> is the name of the tokenizer to use and <tokenizer args> is an optional list of whitespace separated qualifiers to pass to the tokenizer implementation. A tokenizer specification may be placed anywhere in the column list, but at most one tokenizer declaration is allowed for each CREATE VIRTUAL TABLE statement. See below for a detailed description of using (and, if necessary, implementing) a tokenizer.

```
-- Create an FTS table named "papers" with two columns that uses
-- the tokenizer "porter".
CREATE VIRTUAL TABLE papers USING fts3(author, document, tokenize=porter);

-- Create an FTS table with a single column - "content" - that uses
-- the "simple" tokenizer.
CREATE VIRTUAL TABLE data USING fts4(tokenize=simple);

-- Create an FTS table with two columns that uses the "icu" tokenizer.
-- The qualifier "en_AU" is passed to the tokenizer implementation
CREATE VIRTUAL TABLE names USING fts3(a, b, tokenize=icu en_AU);
```

FTS tables may be dropped from the database using an ordinary DROP TABLE statement. For example:

```
-- Create, then immediately drop, an FTS4 table.
CREATE VIRTUAL TABLE data USING fts4();
DROP TABLE data;
```

1.3. Populating FTS Tables

FTS tables are populated using INSERT, UPDATE and DELETE statements in the same way as ordinary SQLite tables are.

As well as the columns named by the user (or the "content" column if no module arguments were specified as part of the CREATE VIRTUAL TABLE statement), each FTS table has a "rowid" column. The rowid of an FTS table behaves in the same way as the rowid column of an ordinary SQLite table, except that the values stored in the rowid column of an FTS table remain unchanged if the database is rebuilt using the VACUUM command. For FTS tables, "docid" is allowed as an alias along with the usual "rowid", "oid" and "*oid*" identifiers. Attempting to insert or update a row with a docid value that already exists in the table is an error, just as it would be with an ordinary SQLite table.

There is one other subtle difference between "docid" and the normal SQLite aliases for the rowid column. Normally, if an INSERT or UPDATE statement assigns discrete values to two or more aliases of the rowid column, SQLite writes the rightmost of such values specified in the INSERT or UPDATE statement to the database. However, assigning a non-NULL value to both the "docid" and one or more of the SQLite rowid aliases when inserting or updating an FTS table is considered an error. See below for an example.

```
-- Create an FTS table
CREATE VIRTUAL TABLE pages USING fts4(title, body);

-- Insert a row with a specific docid value.
INSERT INTO pages(docid, title, body) VALUES(53, 'Home Page', 'SQLite is a software...');

-- Insert a row and allow FTS to assign a docid value using the same algorithm as
-- SQLite uses for ordinary tables. In this case the new docid will be 54,
-- one greater than the largest docid currently present in the table.
INSERT INTO pages(title, body) VALUES('Download', 'All SQLite source code...');

-- Change the title of the row just inserted.
UPDATE pages SET title = 'Download SQLite' WHERE rowid = 54;

-- Delete the entire table contents.
DELETE FROM pages;

-- The following is an error. It is not possible to assign non-NULL values to both
-- the rowid and docid columns of an FTS table.
INSERT INTO pages(rowid, docid, title, body) VALUES(1, 2, 'A title', 'A document body');
```

To support full-text queries, FTS maintains an inverted index that maps from each unique term or word that appears in the dataset to the locations in which it appears within the table contents. For the curious, a complete description of the data structure used to store this index within the database file appears below. A feature of this data structure is that at any time the database may contain not one index b-tree, but several different b-trees that are incrementally merged as rows are inserted, updated and deleted. This technique improves performance when writing to an FTS table, but causes some overhead for full-text queries that use the index. Evaluating the special "optimize" command, an SQL statement of the form "INSERT INTO <fts-table>(<fts-table>) VALUES('optimize')", causes FTS to merge all existing index b-trees into a single large b-tree containing the entire index. This can be an expensive operation, but may speed up future queries.

For example, to optimize the full-text index for an FTS table named "docs":

```
-- Optimize the internal structure of FTS table "docs".
INSERT INTO docs(docs) VALUES('optimize');
```

The statement above may appear syntactically incorrect to some. Refer to the section describing the simple fts queries for an explanation.

There is another, deprecated, method for invoking the optimize operation using a SELECT statement. New code should use statements similar to the INSERT above to optimize FTS structures.

1.4. Simple FTS Queries

As for all other SQLite tables, virtual or otherwise, data is retrieved from FTS tables using a SELECT statement.

FTS tables can be queried efficiently using SELECT statements of two different forms:

- **Query by rowid**. If the WHERE clause of the SELECT statement contains a sub-clause of the form "rowid = ?", where ? is an SQL expression, FTS is able to retrieve the requested row directly using the equivalent of an SQLite INTEGER PRIMARY KEY index.
- **Full-text query**. If the WHERE clause of the SELECT statement contains a sub-clause of the form "<column> MATCH ?", FTS is able to use the built-in full-text index to restrict the search to those documents that match the full-text query string specified as the right-hand operand of the MATCH clause.

If neither of these two query strategies can be used, all queries on FTS tables are implemented using a linear scan of the entire table. If the table contains large amounts of data, this may be an impractical approach (the first example on this page shows that a linear scan of 1.5 GB of data takes around 30 seconds using a modern PC).

```
-- The examples in this block assume the following FTS table:
CREATE VIRTUAL TABLE mail USING fts3(subject, body);

SELECT * FROM mail WHERE rowid = 15;                -- Fast. Rowid lookup.
SELECT * FROM mail WHERE body MATCH 'sqlite';       -- Fast. Full-text query.
SELECT * FROM mail WHERE mail MATCH 'search';       -- Fast. Full-text query.
SELECT * FROM mail WHERE rowid BETWEEN 15 AND 20;   -- Slow. Linear scan.
SELECT * FROM mail WHERE subject = 'database';      -- Slow. Linear scan.
SELECT * FROM mail WHERE subject MATCH 'database';  -- Fast. Full-text query.
```

In all of the full-text queries above, the right-hand operand of the MATCH operator is a string consisting of a single term. In this case, the MATCH expression evaluates to true for all documents that contain one or more instances of the specified word ("sqlite", "search" or "database", depending on which example you look at). Specifying a single term as the right-hand operand of the MATCH operator results in the simplest and most common type of full-text query possible. However more complicated queries are possible, including phrase searches, term-prefix searches and searches for documents containing combinations of terms occurring within a defined proximity of each other. The various ways in which the full-text index may be queried are described below.

Normally, full-text queries are case-insensitive. However, this is dependent on the specific tokenizer used by the FTS table being queried. Refer to the section on tokenizers for details.

The paragraph above notes that a MATCH operator with a simple term as the right-hand operand evaluates to true for all documents that contain the specified term. In this context, the "document" may refer to either the data stored in a single column of a row of an FTS table, or to the contents of all columns in a single row, depending on the identifier used as the left-hand operand to the MATCH operator. If the identifier specified as the left-hand operand of the MATCH operator is an FTS table column name, then the document that the search term must be contained in is the value stored in the specified column. However, if the identifier is the name of the FTS *table* itself, then the MATCH operator evaluates to true for each row of the FTS table for which any column contains the search term. The following example demonstrates this:

```
-- Example schema
CREATE VIRTUAL TABLE mail USING fts3(subject, body);

-- Example table population
INSERT INTO mail(docid, subject, body) VALUES(1, 'software feedback', 'found it too slow'
INSERT INTO mail(docid, subject, body) VALUES(2, 'software feedback', 'no feedback');
INSERT INTO mail(docid, subject, body) VALUES(3, 'slow lunch order',  'was a software pro

-- Example queries
SELECT * FROM mail WHERE subject MATCH 'software';    -- Selects rows 1 and 2
SELECT * FROM mail WHERE body    MATCH 'feedback';    -- Selects row 2
SELECT * FROM mail WHERE mail    MATCH 'software';    -- Selects rows 1, 2 and 3
SELECT * FROM mail WHERE mail    MATCH 'slow';        -- Selects rows 1 and 3
```

At first glance, the final two full-text queries in the example above seem to be syntactically incorrect, as there is a table name ("mail") used as an SQL expression. The reason this is acceptable is that each FTS table actually has a HIDDEN column with the same name as the table itself (in this case, "mail"). The value stored in this column is not meaningful to the application, but can be used as the left-hand operand to a MATCH operator. This special column may also be passed as an argument to the FTS auxiliary functions.

The following example illustrates the above. The expressions "docs", "docs.docs" and "main.docs.docs" all refer to column "docs". However, the expression "main.docs" does not refer to any column. It could be used to refer to a table, but a table name is not allowed in the context in which it is used below.

```
-- Example schema
CREATE VIRTUAL TABLE docs USING fts4(content);

-- Example queries
SELECT * FROM docs WHERE docs MATCH 'sqlite';              -- OK.
SELECT * FROM docs WHERE docs.docs MATCH 'sqlite';         -- OK.
SELECT * FROM docs WHERE main.docs.docs MATCH 'sqlite';    -- OK.
SELECT * FROM docs WHERE main.docs MATCH 'sqlite';         -- Error.
```

1.5. Summary

From the users point of view, FTS tables are similar to ordinary SQLite tables in many ways. Data may be added to, modified within and removed from FTS tables using the INSERT, UPDATE and DELETE commands just as it may be with ordinary tables. Similarly, the

SELECT command may be used to query data. The following list summarizes the differences between FTS and ordinary tables:

1. As with all virtual table types, it is not possible to create indices or triggers attached to FTS tables. Nor is it possible to use the ALTER TABLE command to add extra columns to FTS tables (although it is possible to use ALTER TABLE to rename an FTS table).
2. Data-types specified as part of the "CREATE VIRTUAL TABLE" statement used to create an FTS table are ignored completely. Instead of the normal rules for applying type affinity to inserted values, all values inserted into FTS table columns (except the special rowid column) are converted to type TEXT before being stored.
3. FTS tables permit the special alias "docid" to be used to refer to the rowid column supported by all virtual tables.
4. The FTS MATCH operator is supported for queries based on the built-in full-text index.
5. The FTS auxiliary functions, snippet(), offsets(), and matchinfo() are available to support full-text queries.
6. Every FTS table has a hidden column with the same name as the table itself. The value contained in each row for the hidden column is a blob that is only useful as the left operand of a MATCH operator, or as the left-most argument to one of the FTS auxiliary functions.

2. Compiling and Enabling FTS3 and FTS4

Although FTS3 and FTS4 are included with the SQLite core source code, they are not enabled by default. To build SQLite with FTS functionality enabled, define the preprocessor macro SQLITE_ENABLE_FTS3 when compiling. New applications should also define the SQLITE_ENABLE_FTS3_PARENTHESIS macro to enable the enhanced query syntax (see below). Usually, this is done by adding the following two switches to the compiler command line:

```
-DSQLITE_ENABLE_FTS3
-DSQLITE_ENABLE_FTS3_PARENTHESIS
```

Note that enabling FTS3 also makes FTS4 available. There is not a separate SQLITE_ENABLE_FTS4 compile-time option. A build of SQLite either supports both FTS3 and FTS4 or it supports neither.

If using the amalgamation autoconf based build system, setting the CPPFLAGS environment variable while running the 'configure' script is an easy way to set these macros. For example, the following command:

```
CPPFLAGS="-DSQLITE_ENABLE_FTS3 -DSQLITE_ENABLE_FTS3_PARENTHESIS" ./configure &lt;configur
```

where <*configure options*> are those options normally passed to the configure script, if any.

Because FTS3 and FTS4 are virtual tables, The SQLITE_ENABLE_FTS3 compile-time option is incompatible with the SQLITE_OMIT_VIRTUALTABLE option.

If a build of SQLite does not include the FTS modules, then any attempt to prepare an SQL statement to create an FTS3 or FTS4 table or to drop or access an existing FTS table in any way will fail. The error message returned will be similar to "no such module: ftsN" (where N is either 3 or 4).

If the C version of the ICU library is available, then FTS may also be compiled with the SQLITE_ENABLE_ICU pre-processor macro defined. Compiling with this macro enables an FTS tokenizer that uses the ICU library to split a document into terms (words) using the conventions for a specified language and locale.

```
-DSQLITE_ENABLE_ICU
```

3. Full-text Index Queries

The most useful thing about FTS tables is the queries that may be performed using the built-in full-text index. Full-text queries are performed by specifying a clause of the form "<column> MATCH <full-text query expression>" as part of the WHERE clause of a SELECT statement that reads data from an FTS table. Simple FTS queries that return all documents that contain a given term are described above. In that discussion the right-hand operand of the MATCH operator was assumed to be a string consisting of a single term. This section describes the more complex query types supported by FTS tables, and how they may be utilized by specifying a more complex query expression as the right-hand operand of a MATCH operator.

FTS tables support three basic query types:

- **Token or token prefix queries**. An FTS table may be queried for all documents that contain a specified term (the simple case described above), or for all documents that contain a term with a specified prefix. As we have seen, the query expression for a specific term is simply the term itself. The query expression used to search for a term prefix is the prefix itself with a '*' character appended to it. For example:

```
-- Virtual table declaration
CREATE VIRTUAL TABLE docs USING fts3(title, body);

-- Query for all documents containing the term "linux":
SELECT * FROM docs WHERE docs MATCH 'linux';

-- Query for all documents containing a term with the prefix "lin". This will match
-- all documents that contain "linux", but also those that contain terms "linear",
--"linker", "linguistic" and so on.
SELECT * FROM docs WHERE docs MATCH 'lin*';
```

- Normally, a token or token prefix query is matched against the FTS table column specified as the right-hand side of the MATCH operator. Or, if the special column with the same name as the FTS table itself is specified, against all columns. This may be overridden by specifying a column-name followed by a ":" character before a basic term query. There may be space between the ":" and the term to query for, but not between the column-name and the ":" character. For example:

```
-- Query the database for documents for which the term "linux" appears in
-- the document title, and the term "problems" appears in either the title
-- or body of the document.
SELECT * FROM docs WHERE docs MATCH 'title:linux problems';

-- Query the database for documents for which the term "linux" appears in
-- the document title, and the term "driver" appears in the body of the document
-- ("driver" may also appear in the title, but this alone will not satisfy the.
-- query criteria).
SELECT * FROM docs WHERE body MATCH 'title:linux driver';
```

- If the FTS table is an FTS4 table (not FTS3), a token may also be prefixed with a "^" character. In this case, in order to match the token must appear as the very first token in any column of the matching row. Examples:

```
-- All documents for which "linux" is the first token of at least one
-- column.
SELECT * FROM docs WHERE docs MATCH '^linux';

-- All documents for which the first token in column "title" begins with "lin".
SELECT * FROM docs WHERE body MATCH 'title: ^lin*';
```

- **Phrase queries**. A phrase query is a query that retrieves all documents that contain a nominated set of terms or term prefixes in a specified order with no intervening tokens. Phrase queries are specified by enclosing a space separated sequence of terms or term prefixes in double quotes ("). For example:

```
-- Query for all documents that contain the phrase "linux applications".
SELECT * FROM docs WHERE docs MATCH '"linux applications"';

-- Query for all documents that contain a phrase that matches "lin* app*". As well as
-- "linux applications", this will match common phrases such as "linoleum appliances"
-- or "link apprentice".
SELECT * FROM docs WHERE docs MATCH '"lin* app*"';
```

- **NEAR queries**. A NEAR query is a query that returns documents that contain a two or more nominated terms or phrases within a specified proximity of each other (by default with 10 or less intervening terms). A NEAR query is specified by putting the keyword "NEAR" between two phrase, term or prefix queries. To specify a proximity other than the default, an operator of the form "NEAR/<*N*>" may be used, where <*N*> is the maximum number of intervening terms allowed. For example:

```
-- Virtual table declaration.
CREATE VIRTUAL TABLE docs USING fts4();

-- Virtual table data.
INSERT INTO docs VALUES('SQLite is an ACID compliant embedded relational database managem

-- Search for a document that contains the terms "sqlite" and "database" with
-- not more than 10 intervening terms. This matches the only document in
-- table docs (since there are only six terms between "SQLite" and "database"
-- in the document).
SELECT * FROM docs WHERE docs MATCH 'sqlite NEAR database';

-- Search for a document that contains the terms "sqlite" and "database" with
-- not more than 6 intervening terms. This also matches the only document in
-- table docs. Note that the order in which the terms appear in the document
-- does not have to be the same as the order in which they appear in the query.
SELECT * FROM docs WHERE docs MATCH 'database NEAR/6 sqlite';

-- Search for a document that contains the terms "sqlite" and "database" with
-- not more than 5 intervening terms. This query matches no documents.
SELECT * FROM docs WHERE docs MATCH 'database NEAR/5 sqlite';

-- Search for a document that contains the phrase "ACID compliant" and the term
-- "database" with not more than 2 terms separating the two. This matches the
-- document stored in table docs.
SELECT * FROM docs WHERE docs MATCH 'database NEAR/2 "ACID compliant"';

-- Search for a document that contains the phrase "ACID compliant" and the term
-- "sqlite" with not more than 2 terms separating the two. This also matches
-- the only document stored in table docs.
SELECT * FROM docs WHERE docs MATCH '"ACID compliant" NEAR/2 sqlite';
```

- More than one NEAR operator may appear in a single query. In this case each pair of terms or phrases separated by a NEAR operator must appear within the specified proximity of each other in the document. Using the same table and data as in the block of examples above:

```
-- The following query selects documents that contains an instance of the term
-- "sqlite" separated by two or fewer terms from an instance of the term "acid",
-- which is in turn separated by two or fewer terms from an instance of the term
-- "relational".
SELECT * FROM docs WHERE docs MATCH 'sqlite NEAR/2 acid NEAR/2 relational';

-- This query matches no documents. There is an instance of the term "sqlite" with
-- sufficient proximity to an instance of "acid" but it is not sufficiently close
-- to an instance of the term "relational".
SELECT * FROM docs WHERE docs MATCH 'acid NEAR/2 sqlite NEAR/2 relational';
```

Phrase and NEAR queries may not span multiple columns within a row.

The three basic query types described above may be used to query the full-text index for the set of documents that match the specified criteria. Using the FTS query expression language it is possible to perform various set operations on the results of basic queries. There are currently three supported operations:

- The AND operator determines the **intersection** of two sets of documents.
- The OR operator calculates the **union** of two sets of documents.
- The NOT operator (or, if using the standard syntax, a unary "-" operator) may be used to compute the **relative complement** of one set of documents with respect to another.

The FTS modules may be compiled to use one of two slightly different versions of the full-text query syntax, the "standard" query syntax and the "enhanced" query syntax. The basic term, term-prefix, phrase and NEAR queries described above are the same in both versions of the syntax. The way in which set operations are specified is slightly different. The following two sub-sections describe the part of the two query syntaxes that pertains to set operations. Refer to the description of how to compile fts for compilation notes.

3.1. Set Operations Using The Enhanced Query Syntax

The enhanced query syntax supports the AND, OR and NOT binary set operators. Each of the two operands to an operator may be a basic FTS query, or the result of another AND, OR or NOT set operation. Operators must be entered using capital letters. Otherwise, they are interpreted as basic term queries instead of set operators.

The AND operator may be implicitly specified. If two basic queries appear with no operator separating them in an FTS query string, the results are the same as if the two basic queries were separated by an AND operator. For example, the query expression "implicit operator" is a more succinct version of "implicit AND operator".

```
-- Virtual table declaration
CREATE VIRTUAL TABLE docs USING fts3();

-- Virtual table data
INSERT INTO docs(docid, content) VALUES(1, 'a database is a software system');
INSERT INTO docs(docid, content) VALUES(2, 'sqlite is a software system');
INSERT INTO docs(docid, content) VALUES(3, 'sqlite is a database');

-- Return the set of documents that contain the term "sqlite", and the
-- term "database". This query will return the document with docid 3 only.
SELECT * FROM docs WHERE docs MATCH 'sqlite AND database';

-- Again, return the set of documents that contain both "sqlite" and
-- "database". This time, use an implicit AND operator. Again, document
-- 3 is the only document matched by this query.
SELECT * FROM docs WHERE docs MATCH 'database sqlite';

-- Query for the set of documents that contains either "sqlite" or "database".
-- All three documents in the database are matched by this query.
SELECT * FROM docs WHERE docs MATCH 'sqlite OR database';

-- Query for all documents that contain the term "database", but do not contain
-- the term "sqlite". Document 1 is the only document that matches this criteria.
SELECT * FROM docs WHERE docs MATCH 'database NOT sqlite';

-- The following query matches no documents. Because "and" is in lowercase letters,
-- it is interpreted as a basic term query instead of an operator. Operators must
-- be specified using capital letters. In practice, this query will match any documents
-- that contain each of the three terms "database", "and" and "sqlite" at least once.
-- No documents in the example data above match this criteria.
SELECT * FROM docs WHERE docs MATCH 'database and sqlite';
```

The examples above all use basic full-text term queries as both operands of the set operations demonstrated. Phrase and NEAR queries may also be used, as may the results of other set operations. When more than one set operation is present in an FTS query, the precedence of operators is as follows:

Operator	Enhanced Query Syntax Precedence
NOT	Highest precedence (tightest grouping).
AND	
OR	Lowest precedence (loosest grouping).

When using the enhanced query syntax, parenthesis may be used to override the default precedence of the various operators. For example:

```
-- Return the docid values associated with all documents that contain the
-- two terms "sqlite" and "database", and/or contain the term "library".
SELECT docid FROM docs WHERE docs MATCH 'sqlite AND database OR library';

-- This query is equivalent to the above.
SELECT docid FROM docs WHERE docs MATCH 'sqlite AND database'
  UNION
SELECT docid FROM docs WHERE docs MATCH 'library';

-- Query for the set of documents that contains the term "linux", and at least
-- one of the phrases "sqlite database" and "sqlite library".
SELECT docid FROM docs WHERE docs MATCH '("sqlite database" OR "sqlite library") AND linu

-- This query is equivalent to the above.
SELECT docid FROM docs WHERE docs MATCH 'linux'
  INTERSECT
SELECT docid FROM (
  SELECT docid FROM docs WHERE docs MATCH '"sqlite library"'
    UNION
  SELECT docid FROM docs WHERE docs MATCH '"sqlite database"'
);
```

3.2. Set Operations Using The Standard Query Syntax

FTS query set operations using the standard query syntax are similar, but not identical, to set operations with the enhanced query syntax. There are four differences, as follows:

1. Only the implicit version of the AND operator is supported. Specifying the string "AND" as part of a standard query syntax query is interpreted as a term query for the set of documents containing the term "and".

2. Parenthesis are not supported.

3. The NOT operator is not supported. Instead of the NOT operator, the standard query syntax supports a unary "-" operator that may be applied to basic term and term-prefix queries (but not to phrase or NEAR queries). A term or term-prefix that has a unary "-" operator attached to it may not appear as an operand to an OR operator. An FTS query may not consist entirely of terms or term-prefix queries with unary "-" operators attached to them.

```
-- Search for the set of documents that contain the term "sqlite" but do
-- not contain the term "database".
SELECT * FROM docs WHERE docs MATCH 'sqlite -database';
```

1. The relative precedence of the set operations is different. In particular, using the standard query syntax the "OR" operator has a higher precedence than "AND". The precedence of operators when using the standard query syntax is:

Operator	Standard Query Syntax Precedence
Unary "-"	Highest precedence (tightest grouping).
OR	
AND	Lowest precedence (loosest grouping).

1. The following example illustrates precedence of operators using the standard query syntax:

```
-- Search for documents that contain at least one of the terms "database"
-- and "sqlite", and also contain the term "library". Because of the differences
-- in operator precedences, this query would have a different interpretation using
-- the enhanced query syntax.
SELECT * FROM docs WHERE docs MATCH 'sqlite OR database library';
```

4. Auxiliary Functions - Snippet, Offsets and Matchinfo

The FTS3 and FTS4 modules provide three special SQL scalar functions that may be useful to the developers of full-text query systems: "snippet", "offsets" and "matchinfo". The purpose of the "snippet" and "offsets" functions is to allow the user to identify the location of queried terms in the returned documents. The "matchinfo" function provides the user with metrics that may be useful for filtering or sorting query results according to relevance.

The first argument to all three special SQL scalar functions must be the FTS hidden column of the FTS table that the function is applied to. The FTS hidden column is an automatically-generated column found on all FTS tables that has the same name as the FTS table itself. For example, given an FTS table named "mail":

```
SELECT offsets(mail) FROM mail WHERE mail MATCH &lt;full-text query expression&gt;;
SELECT snippet(mail) FROM mail WHERE mail MATCH &lt;full-text query expression&gt;;
SELECT matchinfo(mail) FROM mail WHERE mail MATCH &lt;full-text query expression&gt;;
```

The three auxiliary functions are only useful within a SELECT statement that uses the FTS table's full-text index. If used within a SELECT that uses the "query by rowid" or "linear scan" strategies, then the snippet and offsets both return an empty string, and the matchinfo function returns a blob value zero bytes in size.

All three auxiliary functions extract a set of "matchable phrases" from the FTS query expression to work with. The set of matchable phrases for a given query consists of all phrases (including unquoted tokens and token prefixes) in the expression except those that are prefixed with a unary "-" operator (standard syntax) or are part of a sub-expression that is used as the right-hand operand of a NOT operator.

With the following provisos, each series of tokens in the FTS table that matches one of the matchable phrases in the query expression is known as a "phrase match":

1. If a matchable phrase is part of a series of phrases connected by NEAR operators in the FTS query expression, then each phrase match must be sufficiently close to other phrase matches of the relevant types to satisfy the NEAR condition.
2. If the matchable phrase in the FTS query is restricted to matching data in a specified FTS table column, then only phrase matches that occur within that column are considered.

4.1. The Offsets Function

For a SELECT query that uses the full-text index, the offsets() function returns a text value containing a series of space-separated integers. For each term in each phrase match of the current row, there are four integers in the returned list. Each set of four integers is interpreted as follows:

Integer	Interpretation
0	The column number that the term instance occurs in (0 for the leftmost column of the FTS table, 1 for the next leftmost, etc.).
1	The term number of the matching term within the full-text query expression. Terms within a query expression are numbered starting from 0 in the order that they occur.
2	The byte offset of the matching term within the column.
3	The size of the matching term in bytes.

The following block contains examples that use the offsets function.

```
CREATE VIRTUAL TABLE mail USING fts3(subject, body);
INSERT INTO mail VALUES('hello world', 'This message is a hello world message.');
INSERT INTO mail VALUES('urgent: serious', 'This mail is seen as a more serious mail');

-- The following query returns a single row (as it matches only the first
-- entry in table "mail". The text returned by the offsets function is
-- "0 0 6 5 1 0 24 5".
--
-- The first set of four integers in the result indicate that column 0
-- contains an instance of term 0 ("world") at byte offset 6\. The term instance
-- is 5 bytes in size. The second set of four integers shows that column 1
-- of the matched row contains an instance of term 0 ("world") at byte offset
-- 24\. Again, the term instance is 5 bytes in size.
SELECT offsets(mail) FROM mail WHERE mail MATCH 'world';

-- The following query returns also matches only the first row in table "mail".
-- In this case the returned text is "1 0 5 7 1 0 30 7".
SELECT offsets(mail) FROM mail WHERE mail MATCH 'message';

-- The following query matches the second row in table "mail". It returns the
-- text "1 0 28 7 1 1 36 4". Only those occurrences of terms "serious" and "mail"
-- that are part of an instance of the phrase "serious mail" are identified; the
-- other occurrences of "serious" and "mail" are ignored.
SELECT offsets(mail) FROM mail WHERE mail MATCH '"serious mail"';
```

4.2. The Snippet Function

The snippet function is used to create formatted fragments of document text for display as part of a full-text query results report. The snippet function may be passed between one and six arguments, as follows:

Argument	Default Value	Description
0	N/A	The first argument to the snippet function must always be the FTS hidden column of the FTS table being queried and from which the snippet is to be taken. The FTS hidden column is an automatically generated column with the same name as the FTS table itself.
1	"<b>"	The "start match" text.
2	"</b>"	The "end match" text.
3	"<b>...</b>"	The "ellipses" text.
4	-1	The FTS table column number to extract the returned fragments of text from. Columns are numbered from left to right starting with zero. A negative value indicates that the text may be extracted from any column.
5	-15	The absolute value of this integer argument is used as the (approximate) number of tokens to include in the returned text value. The maximum allowable absolute value is 64. The value of this argument is referred to as *N* in the discussion below.

The snippet function first attempts to find a fragment of text consisting of |*N*| tokens within the current row that contains at least one phrase match for each matchable phrase matched somewhere in the current row, where |*N*| is the absolute value of the sixth argument passed to the snippet function. If the text stored in a single column contains less than |*N*| tokens, then the entire column value is considered. Text fragments may not span multiple columns.

If such a text fragment can be found, it is returned with the following modifications:

- If the text fragment does not begin at the start of a column value, the "ellipses" text is prepended to it.
- If the text fragment does not finish at the end of a column value, the "ellipses" text is appended to it.
- For each token in the text fragment that is part of a phrase match, the "start match" text is inserted into the fragment before the token, and the "end match" text is inserted immediately after it.

If more than one such fragment can be found, then fragments that contain a larger number of "extra" phrase matches are favored. The start of the selected text fragment may be moved a few tokens forward or backward to attempt to concentrate the phrase matches toward the center of the fragment.

Assuming *N* is a positive value, if no fragments can be found that contain a phrase match corresponding to each matchable phrase, the snippet function attempts to find two fragments of approximately *N*/2 tokens that between them contain at least one phrase match for each matchable phrase matched by the current row. If this fails, attempts are made to find three fragments of *N*/3 tokens each and finally four *N*/4 token fragments. If a set of four fragments cannot be found that encompasses the required phrase matches, the four fragments of *N*/4 tokens that provide the best coverage are selected.

If *N* is a negative value, and no single fragment can be found containing the required phrase matches, the snippet function searches for two fragments of |*N*| tokens each, then three, then four. In other words, if the specified value of *N* is negative, the sizes of the fragments is not decreased if more than one fragment is required to provide the desired phrase match coverage.

After the *M* fragments have been located, where *M* is between two and four as described in the paragraphs above, they are joined together in sorted order with the "ellipses" text separating them. The three modifications enumerated earlier are performed on the text before it is returned.

```
Note: In this block of examples, newlines and whitespace characters have
been inserted into the document inserted into the FTS table, and the expected
results described in SQL comments. This is done to enhance readability only,
they would not be present in actual SQLite commands or output.

-- Create and populate an FTS table.
CREATE VIRTUAL TABLE text USING fts4();
INSERT INTO text VALUES('
  During 30 Nov-1 Dec, 2-3oC drops. Cool in the upper portion, minimum temperature 14-16o
  and cool elsewhere, minimum temperature 17-20oC. Cold to very cold on mountaintops,
  minimum temperature 6-12oC. Northeasterly winds 15-30 km/hr. After that, temperature
  increases. Northeasterly winds 15-30 km/hr.
');

-- The following query returns the text value:
--
--   "&lt;b&gt;...&lt;/b&gt;cool elsewhere, minimum temperature 17-20oC. &lt;b&gt;Cold&lt
--    &lt;b&gt;cold&lt;/b&gt; on mountaintops, minimum temperature 6&lt;b&gt;...&lt;/b&gt
--
SELECT snippet(text) FROM text WHERE text MATCH 'cold';

-- The following query returns the text value:
--
--   "...the upper portion, [minimum] [temperature] 14-16oC and cool elsewhere,
--    [minimum] [temperature] 17-20oC. Cold..."
--
SELECT snippet(text, '[ ']', '...') FROM text WHERE text MATCH '"min* tem*"'
```

4.3. The Matchinfo Function

The matchinfo function returns a blob value. If it is used within a query that does not use the full-text index (a "query by rowid" or "linear scan"), then the blob is zero bytes in size. Otherwise, the blob consists of zero or more 32-bit unsigned integers in machine byte-order. The exact number of integers in the returned array depends on both the query and the value of the second argument (if any) passed to the matchinfo function.

The matchinfo function is called with either one or two arguments. As for all auxiliary functions, the first argument must be the special FTS hidden column. The second argument, if it is specified, must be a text value comprised only of the characters 'p', 'c', 'n', 'a', 'l', 's', 'x', 'y' and 'b'. If no second argument is explicitly supplied, it defaults to "pcx". The second argument is referred to as the "format string" below.

Characters in the matchinfo format string are processed from left to right. Each character in the format string causes one or more 32-bit unsigned integer values to be added to the returned array. The "values" column in the following table contains the number of integer values appended to the output buffer for each supported format string character. In the formula given, *cols* is the number of columns in the FTS table, and *phrases* is the number of matchable phrases in the query.

Character	Values	Description
p	1	The number of matchable phrases in the query.
c	1	The number of user defined columns in the FTS table (i.e. including the docid or the FTS hidden column).

x	3 *cols* *phrases*	For each distinct combination of a phrase and table colum following three values: In the current row, the number of ti the phrase appears in the column. The total number of tim the phrase appears in the column in all rows in the FTS ta The total number of rows in the FTS table for which the co contains at least one instance of the phrase. The first set o three values corresponds to the left-most column of the ta (column 0) and the left-most matchable phrase in the quer (phrase 0). If the table has more than one column, the sec set of three values in the output array correspond to phras and column 1. Followed by phrase 0, column 2 and so on columns of the table. And so on for phrase 1, column 0, th phrase 1, column 1 etc. In other words, the data for occurrences of phrase *p* in column *c* may be found using t following formula: `hits_this_row = array[3 * (c + p*cols)`
		`hits_all_rows = array[3 * (c + p*cols) + 1]`
		`docs_with_hits = array[3 * (c + p*cols) + 2]`

		For each distinct combination of a phrase and table colum number of usable phrase matches that appear in the colu This is usually identical to the first value in each set of thre returned by the matchinfo 'x' flag. However, the number of reported by the 'y' flag is zero for any phrase that is part of sub-expression that does not match the current row. This

y	*cols * phrases*	makes a difference for expressions that contain AND opera that are descendants of OR operators. For example, consi the expression: `a OR (b AND c)` and the document: `"a c` The [matchinfo 'x' flag]() would report a single hit for the phras "a" and "c". However, the 'y' directive reports the number o for "c" as zero, as it is part of a sub-expression that does n match the document - (b AND c). For queries that do not contain AND operators descended from OR operators, the result values returned by 'y' are always the same as those returned by 'x'.The first value in the array of integer values corresponds to the leftmost column of the table (column 0) the first phrase in the query (phrase 0). The values corresponding to other column/phrase combinations may b located using the following formula: `hits_for_phrase_p_column_c = array[c + p*cols]` For queri that use OR expressions, or those that use LIMIT or return many rows, the 'y' matchinfo option may be faster than 'x'.
b	*((cols+31)/32) * phrases*	The matchinfo 'b' flag provides similar information to the [matchinfo 'y' flag](), but in a more compact form. Instead of tl precise number of hits, 'b' provides a single boolean flag fo each phrase/column combination. If the phrase is present i column at least once (i.e. if the corresponding integer outp 'y' would be non-zero), the corresponding flag is set. Other cleared.If the table has 32 or fewer columns, a single unsig integer is output for each phrase in the query. The least significant bit of the integer is set if the phrase appears at l once in column 0. The second least significant bit is set if t phrase appears once or more in column 1. And so on.If the table has more than 32 columns, an extra integer is added the output of each phrase for each extra 32 columns or par thereof. Integers corresponding to the same phrase are clumped together. For example, if a table with 45 columns queried for two phrases, 4 integers are output. The first corresponds to phrase 0 and columns 0-31 of the table. Th second integer contains data for phrase 0 and columns 32 and so on.For example, if nCol is the number of columns in table, to determine if phrase p is present in column c: `p_is_in_c = array[p * ((nCol+31)/32)] & (1 &lt;&lt; (c %`
n	1	The number of rows in the FTS4 table. This value is only available when querying FTS4 tables, not FTS3.
a	*cols*	For each column, the average number of tokens in the text values stored in the column (considering all rows in the FT table). This value is only available when querying FTS4 tab not FTS3.
l	*cols*	For each column, the length of the value stored in the curre row of the FTS4 table, in tokens. This value is only availab when querying FTS4 tables, not FTS3. And only if the "matchinfo=fts3" directive was not specified as part of the "CREATE VIRTUAL TABLE" statement used to create the table.

s	*cols*	For each column, the length of the longest subsequence o phrase matches that the column value has in common with query text. For example, if a table column contains the text c d e' and the query is 'a c "d e"', then the length of the long common subsequence is 2 (phrase "c" followed by phrase e").

For example:

```
-- Create and populate an FTS4 table with two columns:
CREATE VIRTUAL TABLE t1 USING fts4(a, b);
INSERT INTO t1 VALUES('transaction default models default', 'Non transaction reads');
INSERT INTO t1 VALUES('the default transaction', 'these semantics present');
INSERT INTO t1 VALUES('single request', 'default data');

-- In the following query, no format string is specified and so it defaults
-- to "pcx". It therefore returns a single row consisting of a single blob
-- value 80 bytes in size (20 32-bit integers - 1 for "p", 1 for "c" and
-- 3*2*3 for "x"). If each block of 4 bytes in the blob is interpreted
-- as an unsigned integer in machine byte-order, the values will be:
--
--     3 2  1 3 2  0 1 1  1 2 2  0 1 1  0 0 0  1 1 1
--
-- The row returned corresponds to the second entry inserted into table t1.
-- The first two integers in the blob show that the query contained three
-- phrases and the table being queried has two columns. The next block of
-- three integers describes column 0 (in this case column "a") and phrase
-- 0 (in this case "default"). The current row contains 1 hit for "default"
-- in column 0, of a total of 3 hits for "default" that occur in column
-- 0 of any table row. The 3 hits are spread across 2 different rows.
--
-- The next set of three integers (0 1 1) pertain to the hits for "default"
-- in column 1 of the table (0 in this row, 1 in all rows, spread across
-- 1 rows).
--
SELECT matchinfo(t1) FROM t1 WHERE t1 MATCH 'default transaction "these semantics"';

-- The format string for this query is "ns". The output array will therefore
-- contain 3 integer values - 1 for "n" and 2 for "s". The query returns
-- two rows (the first two rows in the table match). The values returned are:
--
--     3  1 1
--     3  2 0
--
-- The first value in the matchinfo array returned for both rows is 3 (the
-- number of rows in the table). The following two values are the lengths
-- of the longest common subsequence of phrase matches in each column.
SELECT matchinfo(t1, 'ns') FROM t1 WHERE t1 MATCH 'default transaction';
```

The matchinfo function is much faster than either the snippet or offsets functions. This is because the implementation of both snippet and offsets is required to retrieve the documents being analyzed from disk, whereas all data required by matchinfo is available as part of the same portions of the full-text index that are required to implement the full-text query itself. This means that of the following two queries, the first may be an order of magnitude faster than the second:

```
SELECT docid, matchinfo(tbl) FROM tbl WHERE tbl MATCH &lt;query expression&gt;;
SELECT docid, offsets(tbl) FROM tbl WHERE tbl MATCH &lt;query expression&gt;;
```

The matchinfo function provides all the information required to calculate probabilistic "bag-of-words" relevancy scores such as Okapi BM25/BM25F that may be used to order results in a full-text search application. Appendix A of this document, "search application tips", contains an example of using the matchinfo() function efficiently.

5. Fts4aux - Direct Access to the Full-Text Index

As of version 3.7.6, SQLite includes a new virtual table module called "fts4aux", which can be used to inspect the full-text index of an existing FTS table directly. Despite its name, fts4aux works just as well with FTS3 tables as it does with FTS4 tables. Fts4aux tables are read-only. The only way to modify the contents of an fts4aux table is by modifying the contents of the associated FTS table. The fts4aux module is automatically included in all builds that include FTS.

An fts4aux virtual table is constructed with one or two arguments. When used with a single argument, that argument is the unqualified name of the FTS table that it will be used to access. To access a table in a different database (for example, to create a TEMP fts4aux table that will access an FTS3 table in the MAIN database) use the two-argument form and give the name of the target database (ex: "main") in the first argument and the name of the FTS3/4 table as the second argument. (The two-argument form of fts4aux was added for SQLite version 3.7.17 and will throw an error in prior releases.) For example:

```
-- Create an FTS4 table
CREATE VIRTUAL TABLE ft USING fts4(x, y);

-- Create an fts4aux table to access the full-text index for table "ft"
CREATE VIRTUAL TABLE ft_terms USING fts4aux(ft);

-- Create a TEMP fts4aux table accessing the "ft" table in "main"
CREATE VIRTUAL TABLE temp.ft_terms_2 USING fts4aux(main,ft);
```

For each term present in the FTS table, there are between 2 and N+1 rows in the fts4aux table, where N is the number of user-defined columns in the associated FTS table. An fts4aux table always has the same four columns, as follows, from left to right:

Column Name	Column Contents
term	Contains the text of the term for this row.
col	This column may contain either the text value '*' (i.e. a single character, U+002a) or an integer between 0 and N-1, where N is again the number of user-defined columns in the corresponding FTS table.
documents	This column always contains an integer value greater than zero. If the "col" column contains the value '*', then this column contains the number of rows of the FTS table that contain at least one instance of the term (in any column). If col contains an integer value, then this column contains the number of rows of the FTS table that contain at least one instance of the term in the column identified by the col value. As usual, the columns of the FTS table are numbered from left to right, starting with zero.
occurrences	This column also always contains an integer value greater than zero. If the "col" column contains the value '*', then this column contains the total number of instances of the term in all rows of the FTS table (in any column). Otherwise, if col contains an integer value, then this column contains the total number of instances of the term that appear in the FTS table column identified by the col value.
languageid *(hidden)*	This column determines which languageid is used to extract vocabulary from the FTS3/4 table. The default value for languageid is 0. If an alternative language is specified in WHERE clause constraints, then that alternative is used instead of 0. There can only be a single languageid per query. In other words, the WHERE clause cannot contain a range constraint or IN operator on the languageid.

For example, using the tables created above:

```
INSERT INTO ft(x, y) VALUES('Apple banana', 'Cherry');
INSERT INTO ft(x, y) VALUES('Banana Date Date', 'cherry');
INSERT INTO ft(x, y) VALUES('Cherry Elderberry', 'Elderberry');

-- The following query returns this data:
--
--     apple       &#124;  *  &#124;  1  &#124;  1
--     apple       &#124;  0  &#124;  1  &#124;  1
--     banana      &#124;  *  &#124;  2  &#124;  2
--     banana      &#124;  0  &#124;  2  &#124;  2
--     cherry      &#124;  *  &#124;  3  &#124;  3
--     cherry      &#124;  0  &#124;  1  &#124;  1
--     cherry      &#124;  1  &#124;  2  &#124;  2
--     date        &#124;  *  &#124;  1  &#124;  2
--     date        &#124;  0  &#124;  1  &#124;  2
--     elderberry  &#124;  *  &#124;  1  &#124;  2
--     elderberry  &#124;  0  &#124;  1  &#124;  1
--     elderberry  &#124;  1  &#124;  1  &#124;  1
--
SELECT term, col, documents, occurrences FROM ft_terms;
```

In the example, the values in the "term" column are all lower case, even though they were inserted into table "ft" in mixed case. This is because an fts4aux table contains the terms as extracted from the document text by the tokenizer. In this case, since table "ft" uses the simple tokenizer, this means all terms have been folded to lower case. Also, there is (for example) no row with column "term" set to "apple" and column "col" set to 1. Since there are no instances of the term "apple" in column 1, no row is present in the fts4aux table.

During a transaction, some of the data written to an FTS table may be cached in memory and written to the database only when the transaction is committed. However the implementation of the fts4aux module is only able to read data from the database. In practice this means that if an fts4aux table is queried from within a transaction in which the associated FTS table has been modified, the results of the query are likely to reflect only a (possibly empty) subset of the changes made.

6. FTS4 Options

If the "CREATE VIRTUAL TABLE" statement specifies module FTS4 (not FTS3), then special directives - FTS4 options - similar to the "tokenize=*" option may also appear in place of column names. An FTS4 option consists of the option name, followed by an "=" character, followed by the option value. The option value may optionally be enclosed in single or double quotes, with embedded quote characters escaped in the same way as for SQL literals. There may not be whitespace on either side of the "=" character. For example, to create an FTS4 table with the value of option "matchinfo" set to "fts3":

```
-- Create a reduced-footprint FTS4 table.
CREATE VIRTUAL TABLE papers USING fts4(author, document, matchinfo=fts3);
```

FTS4 currently supports the following options:

Option	Interpretation
compress	The compress option is used to specify the compress function. It is an error to specify a compress function without also specifying an uncompress function. See below for details.
content	The content allows the text being indexed to be stored in a separate table distinct from the FTS4 table, or even outside of SQLite.
languageid	The languageid option causes the FTS4 table to have an additional hidden integer column that identifies the language of the text contained in each row. The use of the languageid option allows the same FTS4 table to hold text in multiple languages or scripts, each with different tokenizer rules, and to query each language independently of the others.
matchinfo	When set to the value "fts3", the matchinfo option reduces the amount of information stored by FTS4 with the consequence that the "l" option of matchinfo() is no longer available.
notindexed	This option is used to specify the name of a column for which data is not indexed. Values stored in columns that are not indexed are not matched by MATCH queries. Nor are they recognized by auxiliary functions. A single CREATE VIRTUAL TABLE statement may have any number of notindexed options.
order	The "order" option may be set to either "DESC" or "ASC" (in upper or lower case). If it is set to "DESC", then FTS4 stores its data in such a way as to optimize returning results in descending order by docid. If it is set to "ASC" (the default), then the data structures are optimized for returning results in ascending order by docid. In other words, if many of the queries run against the FTS4 table use "ORDER BY docid DESC", then it may improve performance to add the "order=desc" option to the CREATE VIRTUAL TABLE statement.
prefix	This option may be set to a comma-separated list of positive non-zero integers. For each integer N in the list, a separate index is created in the database file to optimize prefix queries where the query term is N bytes in length, not including the '*' character, when encoded using UTF-8. See below for details.
uncompress	This option is used to specify the uncompress function. It is an error to specify an uncompress function without also specifying a compress function. See below for details.

When using FTS4, specifying a column name that contains an "=" character and is not either a "tokenize=*" specification or a recognized FTS4 option is an error. With FTS3, the first token in the unrecognized directive is interpreted as a column name. Similarly, specifying multiple "tokenize*=" directives in a single table declaration is an error when using FTS4, whereas the second and subsequent "tokenize=*" directives are interpreted as column names by FTS3. For example:

```
-- An error. FTS4 does not recognize the directive "xyz=abc".
CREATE VIRTUAL TABLE papers USING fts4(author, document, xyz=abc);

-- Create an FTS3 table with three columns - "author", "document"
-- and "xyz".
CREATE VIRTUAL TABLE papers USING fts3(author, document, xyz=abc);

-- An error. FTS4 does not allow multiple tokenize=* directives
CREATE VIRTUAL TABLE papers USING fts4(tokenize=porter, tokenize=simple);

-- Create an FTS3 table with a single column named "tokenize". The
-- table uses the "porter" tokenizer.
CREATE VIRTUAL TABLE papers USING fts3(tokenize=porter, tokenize=simple);

-- An error. Cannot create a table with two columns named "tokenize".
CREATE VIRTUAL TABLE papers USING fts3(tokenize=porter, tokenize=simple, tokenize=icu);
```

6.1. The compress= and uncompress= options

The compress and uncompress options allow FTS4 content to be stored in the database in a compressed form. Both options should be set to the name of an SQL scalar function registered using sqlite3_create_function() that accepts a single argument.

The compress function should return a compressed version of the value passed to it as an argument. Each time data is written to the FTS4 table, each column value is passed to the compress function and the result value stored in the database. The compress function may return any type of SQLite value (blob, text, real, integer or null).

The uncompress function should uncompress data previously compressed by the compress function. In other words, for all SQLite values X, it should be true that uncompress(compress(X)) equals X. When data that has been compressed by the compress function is read from the database by FTS4, it is passed to the uncompress function before it is used.

If the specified compress or uncompress functions do not exist, the table may still be created. An error is not returned until the FTS4 table is read (if the uncompress function does not exist) or written (if it is the compress function that does not exist).

```
-- Create an FTS4 table that stores data in compressed form. This
-- assumes that the scalar functions zip() and unzip() have been (or
-- will be) added to the database handle.
CREATE VIRTUAL TABLE papers USING fts4(author, document, compress=zip, uncompress=unzip);
```

When implementing the compress and uncompress functions it is important to pay attention to data types. Specifically, when a user reads a value from a compressed FTS table, the value returned by FTS is exactly the same as the value returned by the uncompress function, including the data type. If that data type is not the same as the data type of the original value as passed to the compress function (for example if the uncompress function is returning BLOB when compress was originally passed TEXT), then the users query may not function as expected.

6.2. The content= option

The content option allows FTS4 to forego storing the text being indexed. The content option can be used in two ways:

- The indexed documents are not stored within the SQLite database at all (a "contentless" FTS4 table), or
- The indexed documents are stored in a database table created and managed by the user (an "external content" FTS4 table).

Because the indexed documents themselves are usually much larger than the full-text index, the content option can be used to achieve significant space savings.

6.2.1. Contentless FTS4 Tables

In order to create an FTS4 table that does not store a copy of the indexed documents at all, the content option should be set to an empty string. For example, the following SQL creates such an FTS4 table with three columns - "a", "b", and "c":

```
CREATE VIRTUAL TABLE t1 USING fts4(content="", a, b, c);
```

Data can be inserted into such an FTS4 table using an INSERT statements. However, unlike ordinary FTS4 tables, the user must supply an explicit integer docid value. For example:

```
-- This statement is Ok:
INSERT INTO t1(docid, a, b, c) VALUES(1, 'a b c', 'd e f', 'g h i');

-- This statement causes an error, as no docid value has been provided:
INSERT INTO t1(a, b, c) VALUES('j k l', 'm n o', 'p q r');
```

It is not possible to UPDATE or DELETE a row stored in a contentless FTS4 table. Attempting to do so is an error.

Contentless FTS4 tables also support SELECT statements. However, it is an error to attempt to retrieve the value of any table column other than the docid column. The auxiliary function matchinfo() may be used, but snippet() and offsets() may not. For example:

```
-- The following statements are Ok:
SELECT docid FROM t1 WHERE t1 MATCH 'xxx';
SELECT docid FROM t1 WHERE a MATCH 'xxx';
SELECT matchinfo(t1) FROM t1 WHERE t1 MATCH 'xxx';

-- The following statements all cause errors, as the value of columns
-- other than docid are required to evaluate them.
SELECT * FROM t1;
SELECT a, b FROM t1 WHERE t1 MATCH 'xxx';
SELECT docid FROM t1 WHERE a LIKE 'xxx%';
SELECT snippet(t1) FROM t1 WHERE t1 MATCH 'xxx';
```

Errors related to attempting to retrieve column values other than docid are runtime errors that occur within sqlite3_step(). In some cases, for example if the MATCH expression in a SELECT query matches zero rows, there may be no error at all even if a statement does refer to column values other than docid.

6.2.2. External Content FTS4 Tables

An "external content" FTS4 table is similar to a contentless table, except that if evaluation of a query requires the value of a column other than docid, FTS4 attempts to retrieve that value from a table (or view, or virtual table) nominated by the user (hereafter referred to as the "content table"). The FTS4 module never writes to the content table, and writing to the content table does not affect the full-text index. It is the responsibility of the user to ensure that the content table and the full-text index are consistent.

An external content FTS4 table is created by setting the content option to the name of a table (or view, or virtual table) that may be queried by FTS4 to retrieve column values when required. If the nominated table does not exist, then an external content table behaves in the same way as a contentless table. For example:

```
CREATE TABLE t2(id INTEGER PRIMARY KEY, a, b, c);
CREATE VIRTUAL TABLE t3 USING fts4(content="t2", a, c);
```

Assuming the nominated table does exist, then its columns must be the same as or a superset of those defined for the FTS table. The external table must also be in the same database file as the FTS table. In other words, The external table cannot be in a different database file connected using ATTACH nor may one of the FTS table and the external content be in the TEMP database when the other is in a persistent database file such as MAIN.

When a users query on the FTS table requires a column value other than docid, FTS attempts to read the requested value from the corresponding column of the row in the content table with a rowid value equal to the current FTS docid. Or, if such a row cannot be found in the content table, a NULL value is used instead. For example:

```
CREATE TABLE t2(id INTEGER PRIMARY KEY, a, b, c);
CREATE VIRTUAL TABLE t3 USING fts4(content="t2", b, c);

INSERT INTO t2 VALUES(2, 'a b', 'c d', 'e f');
INSERT INTO t2 VALUES(3, 'g h', 'i j', 'k l');
INSERT INTO t3(docid, b, c) SELECT id, b, c FROM t2;
-- The following query returns a single row with two columns containing
-- the text values "i j" and "k l".
--
-- The query uses the full-text index to discover that the MATCH
-- term matches the row with docid=3\. It then retrieves the values
-- of columns b and c from the row with rowid=3 in the content table
-- to return.
--
SELECT * FROM t3 WHERE t3 MATCH 'k';

-- Following the UPDATE, the query still returns a single row, this
-- time containing the text values "xxx" and "yyy". This is because the
-- full-text index still indicates that the row with docid=3 matches
-- the FTS4 query 'k', even though the documents stored in the content
-- table have been modified.
--
UPDATE t2 SET b = 'xxx', c = 'yyy' WHERE rowid = 3;
SELECT * FROM t3 WHERE t3 MATCH 'k';

-- Following the DELETE below, the query returns one row containing two
-- NULL values. NULL values are returned because FTS is unable to find
-- a row with rowid=3 within the content table.
--
DELETE FROM t2;
SELECT * FROM t3 WHERE t3 MATCH 'k';
```

When a row is deleted from an external content FTS4 table, FTS4 needs to retrieve the column values of the row being deleted from the content table. This is so that FTS4 can update the full-text index entries for each token that occurs within the deleted row to indicate that row has been deleted. If the content table row cannot be found, or if it contains values inconsistent with the contents of the FTS index, the results can be difficult to predict. The FTS index may be left containing entries corresponding to the deleted row, which can lead to seemingly nonsensical results being returned by subsequent SELECT queries. The same applies when a row is updated, as internally an UPDATE is the same as a DELETE followed by an INSERT.

This means that in order to keep an FTS in sync with an external content table, any UPDATE or DELETE operations must be applied first to the FTS table, and then to the external content table. For example:

```
CREATE TABLE t1_real(id INTEGER PRIMARY KEY, a, b, c, d);
CREATE VIRTUAL TABLE t1_fts USING fts4(content="t1_real", b, c);

-- This works. When the row is removed from the FTS table, FTS retrieves
-- the row with rowid=123 and tokenizes it in order to determine the entries
-- that must be removed from the full-text index.
--
DELETE FROM t1_fts WHERE rowid = 123;
DELETE FROM t1_real WHERE rowid = 123;

-- This **does not work**. By the time the FTS table is updated, the row
-- has already been deleted from the underlying content table. As a result
-- FTS is unable to determine the entries to remove from the FTS index and
-- so the index and content table are left out of sync.
--
DELETE FROM t1_real WHERE rowid = 123;
DELETE FROM t1_fts WHERE rowid = 123;
```

Instead of writing separately to the full-text index and the content table, some users may wish to use database triggers to keep the full-text index up to date with respect to the set of documents stored in the content table. For example, using the tables from earlier examples:

```
CREATE TRIGGER t2_bu BEFORE UPDATE ON t2 BEGIN
  DELETE FROM t3 WHERE docid=old.rowid;
END;
CREATE TRIGGER t2_bd BEFORE DELETE ON t2 BEGIN
  DELETE FROM t3 WHERE docid=old.rowid;
END;

CREATE TRIGGER t2_au AFTER UPDATE ON t2 BEGIN
  INSERT INTO t3(docid, b, c) VALUES(new.rowid, new.b, new.c);
END;
CREATE TRIGGER t2_ai AFTER INSERT ON t2 BEGIN
  INSERT INTO t3(docid, b, c) VALUES(new.rowid, new.b, new.c);
END;
```

The DELETE trigger must be fired before the actual delete takes place on the content table. This is so that FTS4 can still retrieve the original values in order to update the full-text index. And the INSERT trigger must be fired after the new row is inserted, so as to handle the case where the rowid is assigned automatically within the system. The UPDATE trigger must be split into two parts, one fired before and one after the update of the content table, for the same reasons.

The FTS4 "rebuild" command deletes the entire full-text index and rebuilds it based on the current set of documents in the content table. Assuming again that "t3" is the name of the external content FTS4 table, the rebuild command looks like this:

```
INSERT INTO t3(t3) VALUES('rebuild');
```

This command may also be used with ordinary FTS4 tables, for example if the implementation of the tokenizer changes. It is an error to attempt to rebuild the full-text index maintained by a contentless FTS4 table, since no content will be available to do the rebuilding.

6.3. The languageid= option

When the languageid option is present, it specifies the name of another hidden column that is added to the FTS4 table and which is used to specify the language stored in each row of the FTS4 table. The name of the languageid hidden column must be distinct from all other column names in the FTS4 table. Example:

```
CREATE VIRTUAL TABLE t1 USING fts4(x, y, languageid="lid")
```

The default value of a languageid column is 0. Any value inserted into a languageid column is converted to a 32-bit (not 64) signed integer.

By default, FTS queries (those that use the MATCH operator) consider only those rows with the languageid column set to 0. To query for rows with other languageid values, a constraint of the form " <language-id>= <integer>" must be added to the queries WHERE clause. For example:</integer></language-id>

```
SELECT * FROM t1 WHERE t1 MATCH 'abc' AND lid=5;
```

It is not possible for a single FTS query to return rows with different languageid values. The results of adding WHERE clauses that use other operators (e.g. lid!=5, or lid<=5) are undefined.

If the content option is used along with the languageid option, then the named languageid column must exist in the content= table (subject to the usual rules - if a query never needs to read the content table then this restriction does not apply).

When the languageid option is used, SQLite invokes the xLanguageid() on the sqlite3_tokenizer_module object immediately after the object is created in order to pass in the language id that the tokenizer should use. The xLanguageid() method will never be called more than once for any single tokenizer object. The fact that different languages might be tokenized differently is one reason why no single FTS query can return rows with different languageid values.

6.4. The matchinfo= option

The matchinfo option may only be set to the value "fts3". Attempting to set matchinfo to anything other than "fts3" is an error. If this option is specified, then some of the extra information stored by FTS4 is omitted. This reduces the amount of disk space consumed by an FTS4 table until it is almost the same as the amount that would be used by the equivalent FTS3 table, but also means that the data accessed by passing the 'l' flag to the matchinfo() function is not available.

6.5. The notindexed= option

Normally, the FTS module maintains an inverted index of all terms in all columns of the table. This option is used to specify the name of a column for which entries should not be added to the index. Multiple "notindexed" options may be used to specify that multiple columns should be omitted from the index. For example:

```
-- Create an FTS4 table for which only the contents of columns c2 and c4
-- are tokenized and added to the inverted index.
CREATE VIRTUAL TABLE t1 USING fts4(c1, c2, c3, c4, notindexed=c1, notindexed=c3);
```

Values stored in unindexed columns are not eligible to match MATCH operators. The do not influence the results of the offsets() or matchinfo() auxiliary functions. Nor will the snippet() function ever return a snippet based on a value stored in an unindexed column.

6.6. The prefix= option

The FTS4 prefix option causes FTS to index term prefixes of specified lengths in the same way that it always indexes complete terms. The prefix option must be set to a comma separated list of positive non-zero integers. For each value N in the list, prefixes of length N bytes (when encoded using UTF-8) are indexed. FTS4 uses term prefix indexes to speed up prefix queries. The cost, of course, is that indexing term prefixes as well as complete terms increases the database size and slows down write operations on the FTS4 table.

Prefix indexes may be used to optimize prefix queries in two cases. If the query is for a prefix of N bytes, then a prefix index created with "prefix=N" provides the best optimization. Or, if no "prefix=N" index is available, a "prefix=N+1" index may be used instead. Using a "prefix=N+1" index is less efficient than a "prefix=N" index, but is better than no prefix index at all.

```
-- Create an FTS4 table with indexes to optimize 2 and 4 byte prefix queries.
CREATE VIRTUAL TABLE t1 USING fts4(c1, c2, prefix="2,4");

-- The following two queries are both optimized using the prefix indexes.
SELECT * FROM t1 WHERE t1 MATCH 'ab*';
SELECT * FROM t1 WHERE t1 MATCH 'abcd*';

-- The following two queries are both partially optimized using the prefix
-- indexes. The optimization is not as pronounced as it is for the queries
-- above, but still an improvement over no prefix indexes at all.
SELECT * FROM t1 WHERE t1 MATCH 'a*';
SELECT * FROM t1 WHERE t1 MATCH 'abc*';
```

7. Special Commands For FTS3 and FTS4

Special INSERT operates can be used to issue commands to FTS3 and FTS4 tables. Every FTS3 and FTS4 has a hidden, read-only column which is the same name as the table itself. INSERTs into this hidden column are interpreted as commands to the FTS3/4 table. For a table with the name "xyz" the following commands are supported:

- INSERT INTO xyz(xyz) VALUES('optimize');
- INSERT INTO xyz(xyz) VALUES('rebuild');
- INSERT INTO xyz(xyz) VALUES('integrity-check');
- INSERT INTO xyz(xyz) VALUES('merge=X,Y');
- INSERT INTO xyz(xyz) VALUES('automerge=N');

7.1. The "optimize" command

The "optimize" command causes FTS3/4 to merge together all of its inverted index b-trees into one large and complete b-tree. Doing an optimize will make subsequent queries run faster since there are fewer b-trees to search, and it may reduce disk usage by coalescing redundant entries. However, for a large FTS table, running optimize can be as expensive as running VACUUM. The optimize command essentially has to read and write the entire FTS table, resulting in a large transaction.

In batch-mode operation, where an FTS table is initially built up using a large number of INSERT operations, then queried repeatedly without further changes, it is often a good idea to run "optimize" after the last INSERT and before the first query.

7.2. The "rebuild" command

The "rebuild" command causes SQLite to discard the entire FTS3/4 table and then rebuild it again from original text. The concept is similar to REINDEX, only that it applies to an FTS3/4 table instead of an ordinary index.

The "rebuild" command should be run whenever the implementation of a custom tokenizer changes, so that all content can be retokenized. The "rebuild" command is also useful when using the FTS4 content option after changes have been made to the original content table.

7.3. The "integrity-check" command

The "integrity-check" command causes SQLite to read and verify the accuracy of all inverted indices in an FTS3/4 table by comparing those inverted indices against the original content. The "integrity-check" command silently succeeds if the inverted indices are all ok, but will fail with an SQLITE_CORRUPT error if any problems are found.

The "integrity-check" command is similar in concept to PRAGMA integrity_check. In a working system, the "integrity-command" should always be successful. Possible causes of integrity-check failures include:

- The application has made changes to the FTS shadow tables directly, without using the FTS3/4 virtual table, causing the shadow tables to become out of sync with each other.
- Using the FTS4 content option and failing to manually keep the content in sync with the FTS4 inverted indices.
- Bugs in the FTS3/4 virtual table. (The "integrity-check" command was original conceived as part of the test suite for FTS3/4.)
- Corruption to the underlying SQLite database file. (See documentation on how to corrupt and SQLite database for additional information.)

7.4. The "merge=X,Y" command

The "merge=X,Y" command (where X and Y are integers) causes SQLite to do a limited amount of work toward merging the various inverted index b-trees of an FTS3/4 table together into one large b-tree. The X value is the target number of "blocks" to be merged, and Y is the minimum number of b-tree segments on a level required before merging will be applied to that level. The value of Y should be between 2 and 16 with a recommended value of 8. The value of X can be any positive integer but values on the order of 100 to 300 are recommended.

When an FTS table accumulates 16 b-tree segments at the same level, the next INSERT into that table will cause all 16 segments to be merged into a single b-tree segment at the next higher level. The effect of these level merges is that most INSERTs into an FTS table are very fast and take minimal memory, but an occasional INSERT is slow and generates a large transaction because of the need to do merging. This results in "spiky" performance of INSERTs.

To avoid spiky INSERT performance, an application can run the "merge=X,Y" command periodically, possibly in an idle thread or idle process, to ensure that the FTS table never accumulates too many b-tree segments at the same level. INSERT performance spikes can generally be avoided, and performance of FTS3/4 can be maximized, by running "merge=X,Y" after every few thousand document inserts. Each "merge=X,Y" command will run in a separate transaction (unless they are grouped together using BEGIN...COMMIT, of course). The transactions can be kept small by choosing a value for X in the range of 100 to 300. The idle thread that is running the merge commands can know when it is done by checking the difference in sqlite3_total_changes() before and after each "merge=X,Y" command and stopping the loop when the difference drops below two.

7.5. The "automerge=N" command

The "automerge=N" command (where N is an integer between 0 and 15, inclusive) is used to configure an FTS3/4 tables "automerge" parameter, which controls automatic incremental inverted index merging. The default automerge value for new tables is 0, meaning that automatic incremental merging is completely disabled. If the value of the automerge parameter is modified using the "automerge=N" command, the new parameter value is stored persistently in the database and is used by all subsequently established database connections.

Setting the automerge parameter to a non-zero value enables automatic incremental merging. This causes SQLite to do a small amount of inverted index merging after every INSERT operation. The amount of merging performed is designed so that the FTS3/4 table never reaches a point where it has 16 segments at the same level and hence has to do a large merge in order to complete an insert. In other words, automatic incremental merging is designed to prevent spiky INSERT performance.

The downside of automatic incremental merging is that it makes every INSERT, UPDATE, and DELETE operation on an FTS3/4 table run a little slower, since extra time must be used to do the incremental merge. For maximum performance, it is recommended that applications disable automatic incremental merge and instead use the "merge" command in an idle process to keep the inverted indices well merged. But if the structure of an application does not easily allow for idle processes, the use of automatic incremental merge is a very reasonable fallback solution.

The actual value of the automerge parameter determines the number of index segments merged simultaneously by an automatic inverted index merge. If the value is set to N, the system waits until there are at least N segments on a single level before beginning to incrementally merge them. Setting a lower value of N causes segments to be merged more quickly, which may speed up full-text queries and, if the workload contains UPDATE or DELETE operations as well as INSERTs, reduce the space on disk consumed by the full-text index. However, it also increases the amount of data written to disk.

For general use in cases where the workload contains few UPDATE or DELETE operations, a good choice for automerge is 8. If the workload contains many UPDATE or DELETE commands, or if query speed is a concern, it may be advantageous to reduce automerge to 2.

For reasons of backwards compatibility, the "automerge=1" command sets the automerge parameter to 8, not 1 (a value of 1 would make no sense anyway, as merging data from a single segment is a no-op).

8. Tokenizers

An FTS tokenizer is a set of rules for extracting terms from a document or basic FTS full-text query.

Unless a specific tokenizer is specified as part of the CREATE VIRTUAL TABLE statement used to create the FTS table, the default tokenizer, "simple", is used. The simple tokenizer extracts tokens from a document or basic FTS full-text query according to the following rules:

- A term is a contiguous sequence of eligible characters, where eligible characters are all alphanumeric characters and all characters with Unicode codepoint values greater than or equal to 128. All other characters are discarded when splitting a document into terms. Their only contribution is to separate adjacent terms.
- All uppercase characters within the ASCII range (Unicode codepoints less than 128), are transformed to their lowercase equivalents as part of the tokenization process. Thus, full-text queries are case-insensitive when using the simple tokenizer.

For example, when a document containing the text "Right now, they're very frustrated.", the terms extracted from the document and added to the full-text index are, in order, "right now they re very frustrated". Such a document would match a full-text query such as "MATCH 'Frustrated'", as the simple tokenizer transforms the term in the query to lowercase before searching the full-text index.

As well as the "simple" tokenizer, the FTS source code features a tokenizer that uses the Porter Stemming algorithm. This tokenizer uses the same rules to separate the input document into terms including folding all terms into lower case, but also uses the Porter Stemming algorithm to reduce related English language words to a common root. For example, using the same input document as in the paragraph above, the porter tokenizer extracts the following tokens: "right now thei veri frustrat". Even though some of these terms are not even English words, in some cases using them to build the full-text index is more useful than the more intelligible output produced by the simple tokenizer. Using the porter tokenizer, the document not only matches full-text queries such as "MATCH 'Frustrated'", but also queries such as "MATCH 'Frustration'", as the term "Frustration" is reduced by the Porter stemmer algorithm to "frustrat" - just as "Frustrated" is. So, when using the porter tokenizer, FTS is able to find not just exact matches for queried terms, but matches against similar English language terms. For more information on the Porter Stemmer algorithm, please refer to the page linked above.

Example illustrating the difference between the "simple" and "porter" tokenizers:

```
-- Create a table using the simple tokenizer. Insert a document into it.
CREATE VIRTUAL TABLE simple USING fts3(tokenize=simple);
INSERT INTO simple VALUES('Right now they''re very frustrated');

-- The first of the following two queries matches the document stored in
-- table "simple". The second does not.
SELECT * FROM simple WHERE simple MATCH 'Frustrated';
SELECT * FROM simple WHERE simple MATCH 'Frustration';

-- Create a table using the porter tokenizer. Insert the same document into it
CREATE VIRTUAL TABLE porter USING fts3(tokenize=porter);
INSERT INTO porter VALUES('Right now they''re very frustrated');

-- Both of the following queries match the document stored in table "porter".
SELECT * FROM porter WHERE porter MATCH 'Frustrated';
SELECT * FROM porter WHERE porter MATCH 'Frustration';
```

If this extension is compiled with the SQLITE_ENABLE_ICU pre-processor symbol defined, then there exists a built-in tokenizer named "icu" implemented using the ICU library. The first argument passed to the xCreate() method (see fts3_tokenizer.h) of this tokenizer may be an ICU locale identifier. For example "tr_TR" for Turkish as used in Turkey, or "en_AU" for English as used in Australia. For example:

```
CREATE VIRTUAL TABLE thai_text USING fts3(text, tokenize=icu th_TH)
```

The ICU tokenizer implementation is very simple. It splits the input text according to the ICU rules for finding word boundaries and discards any tokens that consist entirely of white-space. This may be suitable for some applications in some locales, but not all. If more complex processing is required, for example to implement stemming or discard punctuation, this can be done by creating a tokenizer implementation that uses the ICU tokenizer as part of its implementation.

The "unicode61" tokenizer is available beginning with SQLite version 3.7.13. Unicode61 works very much like "simple" except that it does simple unicode case folding according to rules in Unicode Version 6.1 and it recognizes unicode space and punctuation characters and uses those to separate tokens. The simple tokenizer only does case folding of ASCII characters and only recognizes ASCII space and punctuation characters as token separators.

By default, "unicode61" also removes all diacritics from Latin script characters. This behaviour can be overridden by adding the tokenizer argument "remove_diacritics=0". For example:

```
-- Create tables that remove diacritics from Latin script characters
-- as part of tokenization.
CREATE VIRTUAL TABLE txt1 USING fts4(tokenize=unicode61);
CREATE VIRTUAL TABLE txt2 USING fts4(tokenize=unicode61 "remove_diacritics=1");

-- Create a table that does not remove diacritics from Latin script
-- characters as part of tokenization.
CREATE VIRTUAL TABLE txt3 USING fts4(tokenize=unicode61 "remove_diacritics=0");
```

It is also possible to customize the set of codepoints that unicode61 treats as separator characters. The "separators=" option may be used to specify one or more extra characters that should be treated as separator characters, and the "tokenchars=" option may be used to specify one or more extra characters that should be treated as part of tokens instead of as separator characters. For example:

```
-- Create a table that uses the unicode61 tokenizer, but considers "."
-- and "=" characters to be part of tokens, and capital "X" characters to
-- function as separators.
CREATE VIRTUAL TABLE txt3 USING fts4(tokenize=unicode61 "tokenchars=.=" "separators=X");

-- Create a table that considers space characters (codepoint 32) to be
-- a token character
CREATE VIRTUAL TABLE txt4 USING fts4(tokenize=unicode61 "tokenchars= ");
```

If a character specified as part of the argument to "tokenchars=" is considered to be a token character by default, it is ignored. This is true even if it has been marked as a separator by an earlier "separators=" option. Similarly, if a character specified as part of a "separators=" option is treated as a separator character by default, it is ignored. If multiple "tokenchars=" or "separators=" options are specified, all are processed. For example:

```
-- Create a table that uses the unicode61 tokenizer, but considers "."
-- and "=" characters to be part of tokens, and capital "X" characters to
-- function as separators. Both of the "tokenchars=" options are processed
-- The "separators=" option ignores the "." passed to it, as "." is by
-- default a separator character, even though it has been marked as a token
-- character by an earlier "tokenchars=" option.
CREATE VIRTUAL TABLE txt5 USING fts4(
    tokenize=unicode61 "tokenchars=." "separators=X." "tokenchars=="
);
```

The arguments passed to the "tokenchars=" or "separators=" options are case-sensitive. In the example above, specifying that "X" is a separator character does not affect the way "x" is handled.

8.1. Custom (Application Defined) Tokenizers

In addition to providing built-in "simple", "porter" and (possibly) "icu" and "unicode61" tokenizers, FTS provides an interface for applications to implement and register custom tokenizers written in C. The interface used to create a new tokenizer is defined and described in the fts3_tokenizer.h source file.

Registering a new FTS tokenizer is similar to registering a new virtual table module with SQLite. The user passes a pointer to a structure containing pointers to various callback functions that make up the implementation of the new tokenizer type. For tokenizers, the structure (defined in fts3_tokenizer.h) is called "sqlite3_tokenizer_module".

FTS does not expose a C-function that users call to register new tokenizer types with a database handle. Instead, the pointer must be encoded as an SQL blob value and passed to FTS through the SQL engine by evaluating a special scalar function, "fts3_tokenizer()". The fts3_tokenizer() function may be called with one or two arguments, as follows:

```
SELECT fts3_tokenizer(&lt;tokenizer-name&gt;);
SELECT fts3_tokenizer(&lt;tokenizer-name&gt;, &lt;sqlite3_tokenizer_module ptr&gt;);
```

Where <tokenizer-name> is a string identifying the tokenizer and <sqlite3_tokenizer_module ptr> is a pointer to an sqlite3_tokenizer_module structure encoded as an SQL blob. If the second argument is present, it is registered as tokenizer <tokenizer-name> and a copy of it returned. If only one argument is passed, a pointer to the tokenizer implementation currently registered as <tokenizer-name> is returned, encoded as a blob. Or, if no such tokenizer exists, an SQL exception (error) is raised.

Because of security concerns, SQLite version 3.11.0 only enabled the second form of the fts3_tokenizer() function when the library is compiled with the -DSQLITE_ENABLE_FTS3_TOKENIZER option. In earlier versions it was always available.

Beginning with SQLite version 3.12.0, the second form of fts3_tokenizer() can also be activated at run-time by calling sqlite3_db_config(db,SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER,1,0).

SECURITY WARNING: If a version of the fts3/4 extension that supports the two-argument form of fts3_tokenizer() is deployed in an environment where malicious users can run arbitrary SQL, then those users should be prevented from invoking the two-argument fts3_tokenizer() function. This can be done using the authorization callback, or by disabling the two-argument fts3_tokenizer() interface using a call to sqlite3_db_config(db,SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER,0,0).

The following block contains an example of calling the fts3_tokenizer() function from C code:

```
/*
** Register a tokenizer implementation with FTS3 or FTS4.
*/
int registerTokenizer(
  sqlite3 *db,
  char *zName,
  const sqlite3_tokenizer_module *p
){
  int rc;
  sqlite3_stmt *pStmt;
  const char *zSql = "SELECT fts3_tokenizer(?, ?)";

  rc = sqlite3_prepare_v2(db, zSql, -1, &pStmt, 0);
  if( rc!=SQLITE_OK ){
    return rc;
  }

  sqlite3_bind_text(pStmt, 1, zName, -1, SQLITE_STATIC);
  sqlite3_bind_blob(pStmt, 2, &p, sizeof(p), SQLITE_STATIC);
  sqlite3_step(pStmt);

  return sqlite3_finalize(pStmt);
}

/*
** Query FTS for the tokenizer implementation named zName.
*/
int queryTokenizer(
  sqlite3 *db,
  char *zName,
  const sqlite3_tokenizer_module **pp
){
  int rc;
  sqlite3_stmt *pStmt;
  const char *zSql = "SELECT fts3_tokenizer(?)";

  *pp = 0;
  rc = sqlite3_prepare_v2(db, zSql, -1, &pStmt, 0);
  if( rc!=SQLITE_OK ){
    return rc;
  }

  sqlite3_bind_text(pStmt, 1, zName, -1, SQLITE_STATIC);
  if( SQLITE_ROW==sqlite3_step(pStmt) ){
    if( sqlite3_column_type(pStmt, 0)==SQLITE_BLOB ){
      memcpy(pp, sqlite3_column_blob(pStmt, 0), sizeof(*pp));
    }
  }

  return sqlite3_finalize(pStmt);
}
```

8.2. Querying Tokenizers

The "fts3tokenize" virtual table can be used to directly access any tokenizer. The following SQL demonstrates how to create an instance of the fts3tokenize virtual table:

```
CREATE VIRTUAL TABLE tok1 USING fts3tokenize('porter');
```

The name of the desired tokenizer should be substituted in place of 'porter' in the example, of course. If the tokenizer requires one or more arguments, they should be separated by commas in the fts3tokenize declaration (even though they are separated by spaces in declarations of regular fts4 tables). The following creates fts4 and fts3tokenize tables that use the same tokenizer:

```
CREATE VIRTUAL TABLE text1 USING fts4(tokenize=icu en_AU);
CREATE VIRTUAL TABLE tokens1 USING fts3tokenize(icu, en_AU);

CREATE VIRTUAL TABLE text2 USING fts4(tokenize=unicode61 "tokenchars=@." "separators=123"
CREATE VIRTUAL TABLE tokens2 USING fts3tokenize(unicode61, "tokenchars=@.", "separators=1
```

Once the virtual table is created, it can be queried as follows:

```
SELECT token, start, end, position
  FROM tok1
 WHERE input='This is a test sentence.';
```

The virtual table will return one row of output for each token in the input string. The "token" column is the text of the token. The "start" and "end" columns are the byte offset to the beginning and end of the token in the original input string. The "position" column is the sequence number of the token in the original input string. There is also an "input" column which is simply a copy of the input string that is specified in the WHERE clause. Note that a constraint of the form "input=?" must appear in the WHERE clause or else the virtual table will have no input to tokenize and will return no rows. The example above generates the following output:

```
thi&#124;0&#124;4&#124;0
is&#124;5&#124;7&#124;1
a&#124;8&#124;9&#124;2
test&#124;10&#124;14&#124;3
sentenc&#124;15&#124;23&#124;4
```

Notice that the tokens in the result set from the fts3tokenize virtual table have been transformed according to the rules of the tokenizer. Since this example used the "porter" tokenizer, the "This" token was converted into "thi". If the original text of the token is desired, it can be retrieved using the "start" and "end" columns with the substr() function. For example:

```
SELECT substr(input, start+1, end-start), token, position
  FROM tok1
 WHERE input='This is a test sentence.';
```

The fts3tokenize virtual table can be used on any tokenizer, regardless of whether or not there exists an FTS3 or FTS4 table that actually uses that tokenizer.

9. Data Structures

This section describes at a high-level the way the FTS module stores its index and content in the database. It is **not necessary to read or understand the material in this section in order to use FTS** in an application. However, it may be useful to application developers attempting to analyze and understand FTS performance characteristics, or to developers contemplating enhancements to the existing FTS feature set.

9.1. Shadow Tables

For each FTS virtual table in a database, three to five real (non-virtual) tables are created to store the underlying data. These real tables are called "shadow tables". The real tables are named "%_content", "%_segdir", "%_segments", "%_stat", and "%_docsize", where "%" is replaced by the name of the FTS virtual table.

The leftmost column of the "%*content" table is an INTEGER PRIMARY KEY field named "docid". Following this is one column for each column of the FTS virtual table as declared by the user, named by prepending the column name supplied by the user with "c_N*", where *N* is the index of the column within the table, numbered from left to right starting with 0. Data types supplied as part of the virtual table declaration are not used as part of the %_content table declaration. For example:

```
-- Virtual table declaration
CREATE VIRTUAL TABLE abc USING fts4(a NUMBER, b TEXT, c);

-- Corresponding %_content table declaration
CREATE TABLE abc_content(docid INTEGER PRIMARY KEY, c0a, c1b, c2c);
```

The %_content table contains the unadulterated data inserted by the user into the FTS virtual table by the user. If the user does not explicitly supply a "docid" value when inserting records, one is selected automatically by the system.

The %_stat and %_docsize tables are only created if the FTS table uses the FTS4 module, not FTS3. Furthermore, the %_docsize table is omitted if the FTS4 table is created with the "matchinfo=fts3" directive specified as part of the CREATE VIRTUAL TABLE statement. If they are created, the schema of the two tables is as follows:

```
CREATE TABLE %_stat(
  id INTEGER PRIMARY KEY,
  value BLOB
);

CREATE TABLE %_docsize(
  docid INTEGER PRIMARY KEY,
  size BLOB
);
```

For each row in the FTS table, the %*docsize table contains a corresponding row with the same "docid" value. The "size" field contains a blob consisting of _N* FTS varints, where *N* is the number of user-defined columns in the table. Each varint in the "size" blob is the number of tokens in the corresponding column of the associated row in the FTS table. The %*stat table always contains a single row with the "id" column set to 0. The "value" column contains a blob consisting of _N+1* FTS varints, where *N* is again the number of user-defined columns in the FTS table. The first varint in the blob is set to the total number of rows in the FTS table. The second and subsequent varints contain the total number of tokens stored in the corresponding column for all rows of the FTS table.

The two remaining tables, %_segments and %_segdir, are used to store the full-text index. Conceptually, this index is a lookup table that maps each term (word) to the set of docid values corresponding to records in the %_content table that contain one or more occurrences of the term. To retrieve all documents that contain a specified term, the FTS module queries this index to determine the set of docid values for records that contain the term, then retrieves the required documents from the %_content table. Regardless of the schema of the FTS virtual table, the %_segments and %_segdir tables are always created as follows:

```
CREATE TABLE %_segments(
  blockid INTEGER PRIMARY KEY,       -- B-tree node id
  block blob                         -- B-tree node data
);

CREATE TABLE %_segdir(
  level INTEGER,
  idx INTEGER,
  start_block INTEGER,               -- Blockid of first node in %_segments
  leaves_end_block INTEGER,          -- Blockid of last leaf node in %_segments
  end_block INTEGER,                 -- Blockid of last node in %_segments
  root BLOB,                         -- B-tree root node
  PRIMARY KEY(level, idx)
);
```

The schema depicted above is not designed to store the full-text index directly. Instead, it is used to one or more b-tree structures. There is one b-tree for each row in the %_segdir table. The %_segdir table row contains the root node and various meta-data associated with the b-tree structure, and the %_segments table contains all other (non-root) b-tree nodes. Each b-tree is referred to as a "segment". Once it has been created, a segment b-tree is never updated (although it may be deleted altogether).

The keys used by each segment b-tree are terms (words). As well as the key, each segment b-tree entry has an associated "doclist" (document list). A doclist consists of zero or more entries, where each entry consists of:

- A docid (document id), and
- A list of term offsets, one for each occurrence of the term within the document. A term

> offset indicates the number of tokens (words) that occur before the term in question, not the number of characters or bytes. For example, the term offset of the term "war" in the phrase "Ancestral voices prophesying war!" is 3.

Entries within a doclist are sorted by docid. Positions within a doclist entry are stored in ascending order.

The contents of the logical full-text index is found by merging the contents of all segment b-trees. If a term is present in more than one segment b-tree, then it maps to the union of each individual doclist. If, for a single term, the same docid occurs in more than one doclist, then only the doclist that is part of the most recently created segment b-tree is considered valid.

Multiple b-tree structures are used instead of a single b-tree to reduce the cost of inserting records into FTS tables. When a new record is inserted into an FTS table that already contains a lot of data, it is likely that many of the terms in the new record are already present in a large number of existing records. If a single b-tree were used, then large doclist structures would have to be loaded from the database, amended to include the new docid and term-offset list, then written back to the database. Using multiple b-tree tables allows this to be avoided by creating a new b-tree which can be merged with the existing b-tree (or b-trees) later on. Merging of b-tree structures can be performed as a background task, or once a certain number of separate b-tree structures have been accumulated. Of course, this scheme makes queries more expensive (as the FTS code may have to look up individual terms in more than one b-tree and merge the results), but it has been found that in practice this overhead is often negligible.

9.2. Variable Length Integer (varint) Format

Integer values stored as part of segment b-tree nodes are encoded using the FTS varint format. This encoding is similar, but **not identical**, to the SQLite varint format.

An encoded FTS varint consumes between one and ten bytes of space. The number of bytes required is determined by the sign and magnitude of the integer value encoded. More accurately, the number of bytes used to store the encoded integer depends on the position of the most significant set bit in the 64-bit twos-complement representation of the integer value. Negative values always have the most significant bit set (the sign bit), and so are always stored using the full ten bytes. Positive integer values may be stored using less space.

The final byte of an encoded FTS varint has its most significant bit cleared. All preceding bytes have the most significant bit set. Data is stored in the remaining seven least significant bits of each byte. The first byte of the encoded representation contains the least significant seven bits of the encoded integer value. The second byte of the encoded representation, if it is present, contains the seven next least significant bits of the integer value, and so on. The following table contains examples of encoded integer values:

Decimal	Hexadecimal	Encoded Representation
43	0x000000000000002B	0x2B
200815	0x000000000003106F	0x9C 0xA0 0x0C
-1	0xFFFFFFFFFFFFFFFF	0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0x01

9.3. Segment B-Tree Format

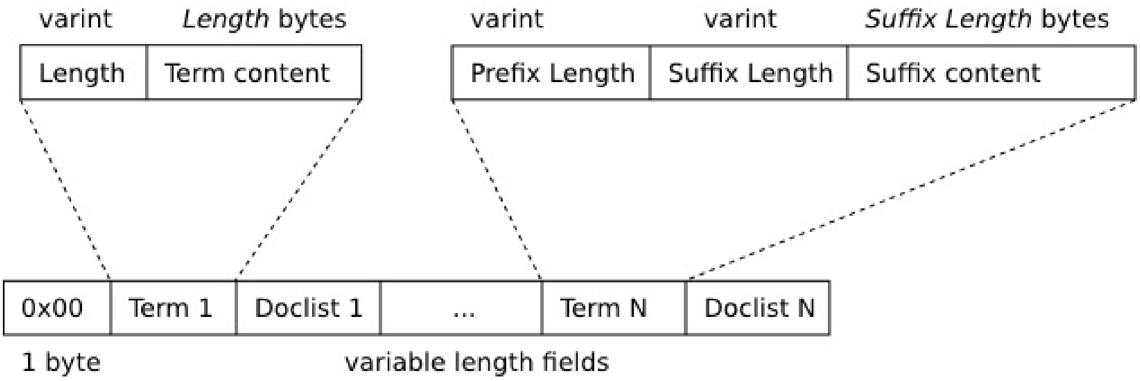
Segment b-trees are prefix-compressed b+-trees. There is one segment b-tree for each row in the %_segdir table (see above). The root node of the segment b-tree is stored as a blob in the "root" field of the corresponding row of the %_segdir table. All other nodes (if any exist) are stored in the "blob" column of the %_segments table. Nodes within the %_segments table are identified by the integer value in the blockid field of the corresponding row. The following table describes the fields of the %_segdir table:

Column	Interpretation
level	Between them, the contents of the "level" and "idx" fields define the relative age of the segment b-tree. The smaller the value stored in the "level" field, the more recently the segment b-tree was created. If two segment b-trees are of the same "level", the segment with the larger value stored in the "idx" column is more recent. The PRIMARY KEY constraint on the %_segdir table prevents any two segments from having the same value for both the "level" and "idx" fields.
idx	See above.
start_block	The blockid that corresponds to the node with the smallest blockid that belongs to this segment b-tree. Or zero if the entire segment b-tree fits on the root node. If it exists, this node is always a leaf node.
leaves_end_block	The blockid that corresponds to the leaf node with the largest blockid that belongs to this segment b-tree. Or zero if the entire segment b-tree fits on the root node.
end_block	This field may contain either an integer or a text field consisting of two integers separated by a space character (unicode codepoint 0x20).The first, or only, integer is the blockid that corresponds to the interior node with the largest blockid that belongs to this segment b-tree. Or zero if the entire segment b-tree fits on the root node. If it exists, this node is always an interior node.The second integer, if it is present, is the aggregate size of all data stored on leaf pages in bytes. If the value is negative, then the segment is the output of an unfinished incremental-merge operation, and the absolute value is current size in bytes.
root	Blob containing the root node of the segment b-tree.

Apart from the root node, the nodes that make up a single segment b-tree are always stored using a contiguous sequence of blockids. Furthermore, the nodes that make up a single level of the b-tree are themselves stored as a contiguous block, in b-tree order. The contiguous sequence of blockids used to store the b-tree leaves are allocated starting with the blockid value stored in the "start_block" column of the corresponding %_segdir row, and finishing at the blockid value stored in the "leaves_end_block" field of the same row. It is therefore possible to iterate through all the leaves of a segment b-tree, in key order, by traversing the %_segments table in blockid order from "start_block" to "leaves_end_block".

9.3.1. Segment B-Tree Leaf Nodes

The following diagram depicts the format of a segment b-tree leaf node.

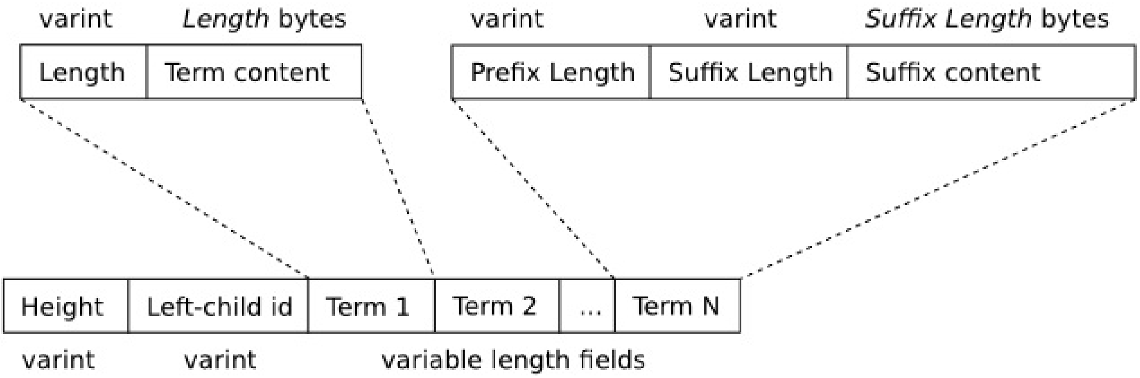


Segment B-Tree Leaf Node Format

The first term stored on each node ("Term 1" in the figure above) is stored verbatim. Each subsequent term is prefix-compressed with respect to its predecessor. Terms are stored within a page in sorted (memcmp) order.

9.3.2. Segment B-Tree Interior Nodes

The following diagram depicts the format of a segment b-tree interior (non-leaf) node.



Segment B-Tree Interior Node Format

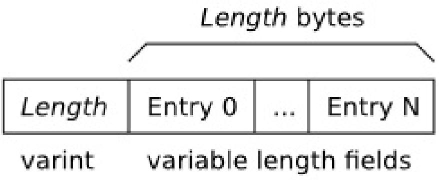
9.4. Doclist Format

A doclist consists of an array of 64-bit signed integers, serialized using the FTS varint format. Each doclist entry is made up of a series of two or more integers, as follows:

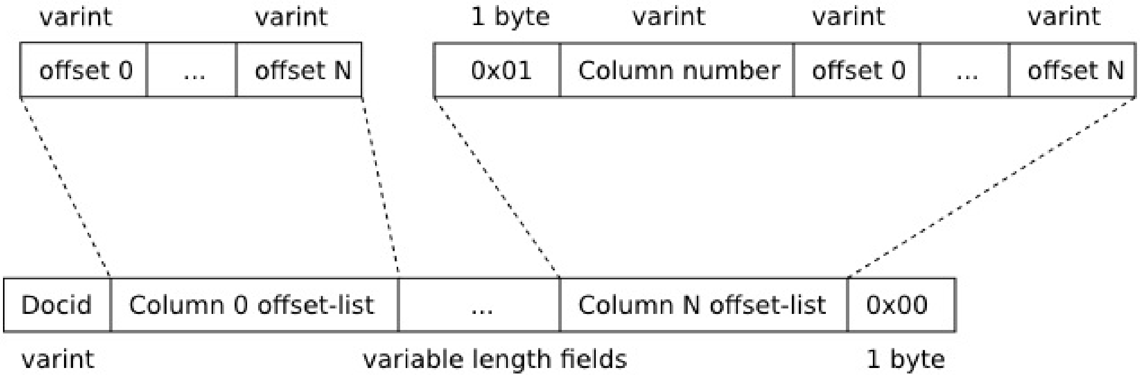
1. The docid value. The first entry in a doclist contains the literal docid value. The first field of each subsequent doclist entry contains the difference between the new docid and the previous one (always a positive number).
2. Zero or more term-offset lists. A term-offset list is present for each column of the FTS virtual table that contains the term. A term-offset list consists of the following:

    i. Constant value 1. This field is omitted for any term-offset list associated with column 0.
    ii. The column number (1 for the second leftmost column, etc.). This field is omitted for any term-offset list associated with column 0.
    iii. A list of term-offsets, sorted from smallest to largest. Instead of storing the term-offset value literally, each integer stored is the difference between the current term-offset and the previous one (or zero if the current term-offset is the first), plus 2.

3. Constant value 0.



FTS3 Doclist Format



FTS Doclist Entry Format

For doclists for which the term appears in more than one column of the FTS virtual table, term-offset lists within the doclist are stored in column number order. This ensures that the term-offset list associated with column 0 (if any) is always first, allowing the first two fields of the term-offset list to be omitted in this case.

Appendix A: Search Application Tips

FTS is primarily designed to support Boolean full-text queries - queries to find the set of documents that match a specified criteria. However, many (most?) search applications require that results are somehow ranked in order of "relevance", where "relevance" is defined as the likelihood that the user who performed the search is interested in a specific element of the returned set of documents. When using a search engine to find documents on the world wide web, the user expects that the most useful, or "relevant", documents will be returned as the first page of results, and that each subsequent page contains progressively less relevant results. Exactly how a machine can determine document relevance based on a users query is a complicated problem and the subject of much ongoing research.

One very simple scheme might be to count the number of instances of the users search terms in each result document. Those documents that contain many instances of the terms are considered more relevant than those with a small number of instances of each term. In an FTS application, the number of term instances in each result could be determined by counting the number of integers in the return value of the offsets function. The following example shows a query that could be used to obtain the ten most relevant results for a query entered by the user:

```
-- This example (and all others in this section) assumes the following schema
CREATE VIRTUAL TABLE documents USING fts3(title, content);

-- Assuming the application has supplied an SQLite user function named "countintegers"
-- that returns the number of space-separated integers contained in its only argument,
-- the following query could be used to return the titles of the 10 documents that contai
-- the greatest number of instances of the users query terms. Hopefully, these 10
-- documents will be those that the users considers more or less the most "relevant".
SELECT title FROM documents
  WHERE documents MATCH &lt;query&gt;
  ORDER BY countintegers(offsets(documents)) DESC
  LIMIT 10 OFFSET 0
```

The query above could be made to run faster by using the FTS matchinfo function to determine the number of query term instances that appear in each result. The matchinfo function is much more efficient than the offsets function. Furthermore, the matchinfo function provides extra information regarding the overall number of occurrences of each query term in the entire document set (not just the current row) and the number of documents in which each query term appears. This may be used (for example) to attach a higher weight to less common terms which may increase the overall computed relevancy of those results the user considers more interesting.

```
-- If the application supplies an SQLite user function called "rank" that
-- interprets the blob of data returned by matchinfo and returns a numeric
-- relevancy based on it, then the following SQL may be used to return the
-- titles of the 10 most relevant documents in the dataset for a users query.
SELECT title FROM documents
  WHERE documents MATCH &lt;query&gt;
  ORDER BY rank(matchinfo(documents)) DESC
  LIMIT 10 OFFSET 0
```

The SQL query in the example above uses less CPU than the first example in this section, but still has a non-obvious performance problem. SQLite satisfies this query by retrieving the value of the "title" column and matchinfo data from the FTS module for every row matched by the users query before it sorts and limits the results. Because of the way SQLite's virtual table interface works, retrieving the value of the "title" column requires loading the entire row from disk (including the "content" field, which may be quite large). This means that if the users query matches several thousand documents, many megabytes of "title" and "content" data may be loaded from disk into memory even though they will never be used for any purpose.

The SQL query in the following example block is one solution to this problem. In SQLite, when a sub-query used in a join contains a LIMIT clause, the results of the sub-query are calculated and stored in temporary table before the main query is executed. This means that SQLite will load only the docid and matchinfo data for each row matching the users query into memory, determine the docid values corresponding to the ten most relevant documents, then load only the title and content information for those 10 documents only. Because both the matchinfo and docid values are gleaned entirely from the full-text index, this results in dramatically less data being loaded from the database into memory.

```
SELECT title FROM documents JOIN (
    SELECT docid, rank(matchinfo(documents)) AS rank
    FROM documents
    WHERE documents MATCH &lt;query&gt;
    ORDER BY rank DESC
    LIMIT 10 OFFSET 0
) AS ranktable USING(docid)
ORDER BY ranktable.rank DESC
```

The next block of SQL enhances the query with solutions to two other problems that may arise in developing search applications using FTS:

1. The snippet function cannot be used with the above query. Because the outer query does not include a "WHERE ... MATCH" clause, the snippet function may not be used with it. One solution is to duplicate the WHERE clause used by the sub-query in the outer query. The overhead associated with this is usually negligible.

2. The relevancy of a document may depend on something other than just the data available in the return value of matchinfo. For example each document in the database may be assigned a static weight based on factors unrelated to its content (origin, author, age, number of references etc.). These values can be stored by the application in a separate table that can be joined against the documents table in the sub-query so that the rank function may access them.

This version of the query is very similar to that used by the sqlite.org documentation search application.

```
-- This table stores the static weight assigned to each document in FTS table
-- "documents". For each row in the documents table there is a corresponding row
-- with the same docid value in this table.
CREATE TABLE documents_data(docid INTEGER PRIMARY KEY, weight);

-- This query is similar to the one in the block above, except that:
--
--   1\. It returns a "snippet" of text along with the document title for display. So
--      that the snippet function may be used, the "WHERE ... MATCH ..." clause from
--      the sub-query is duplicated in the outer query.
--
--   2\. The sub-query joins the documents table with the document_data table, so that
--      implementation of the rank function has access to the static weight assigned
--      to each document.
SELECT title, snippet(documents) FROM documents JOIN (
    SELECT docid, rank(matchinfo(documents), documents_data.weight) AS rank
    FROM documents JOIN documents_data USING(docid)
    WHERE documents MATCH &lt;query&gt;
    ORDER BY rank DESC
    LIMIT 10 OFFSET 0
) AS ranktable USING(docid)
WHERE documents MATCH &lt;query&gt;
ORDER BY ranktable.rank DESC
```

All the example queries above return the ten most relevant query results. By modifying the values used with the OFFSET and LIMIT clauses, a query to return (say) the next ten most relevant results is easy to construct. This may be used to obtain the data required for a search applications second and subsequent pages of results.

The next block contains an example rank function that uses matchinfo data implemented in C. Instead of a single weight, it allows a weight to be externally assigned to each column of each document. It may be registered with SQLite like any other user function using sqlite3_create_function.

```
/*
** SQLite user defined function to use with matchinfo() to calculate the
** relevancy of an FTS match. The value returned is the relevancy score
** (a real value greater than or equal to zero). A larger value indicates
** a more relevant document.
**
** The overall relevancy returned is the sum of the relevancies of each
** column value in the FTS table. The relevancy of a column value is the
** sum of the following for each reportable phrase in the FTS query:
**
**   (&lt;hit count&gt; / &lt;global hit count&gt;) * &lt;column weight&gt;
**
```

```
** where &lt;hit count&gt; is the number of instances of the phrase in the
** column value of the current row and &lt;global hit count&gt; is the number
** of instances of the phrase in the same column of all rows in the FTS
** table. The &lt;column weight&gt; is a weighting factor assigned to each
** column by the caller (see below).
**
** The first argument to this function must be the return value of the FTS
** matchinfo() function. Following this must be one argument for each column
** of the FTS table containing a numeric weight factor for the corresponding
** column. Example:
**
**     CREATE VIRTUAL TABLE documents USING fts3(title, content)
**
** The following query returns the docids of documents that match the full-text
** query &lt;query&gt; sorted from most to least relevant. When calculating
** relevance, query term instances in the 'title' column are given twice the
** weighting of those in the 'content' column.
**
**     SELECT docid FROM documents
**     WHERE documents MATCH &lt;query&gt;
**     ORDER BY rank(matchinfo(documents), 1.0, 0.5) DESC
*/
static void rankfunc(sqlite3_context *pCtx, int nVal, sqlite3_value **apVal){
  int *aMatchinfo;                /* Return value of matchinfo() */
  int nCol;                       /* Number of columns in the table */
  int nPhrase;                    /* Number of phrases in the query */
  int iPhrase;                    /* Current phrase */
  double score = 0.0;             /* Value to return */

  assert( sizeof(int)==4 );

 /* Check that the number of arguments passed to this function is correct.
 ** If not, jump to wrong_number_args. Set aMatchinfo to point to the array
 ** of unsigned integer values returned by FTS function matchinfo. Set
 ** nPhrase to contain the number of reportable phrases in the users full-text
 ** query, and nCol to the number of columns in the table.
 */
  if( nVal&lt;1 ) goto wrong_number_args;
  aMatchinfo = (unsigned int *)sqlite3_value_blob(apVal[0]);
  nPhrase = aMatchinfo[0];
  nCol = aMatchinfo[1];
  if( nVal!=(1+nCol) ) goto wrong_number_args;

 /* Iterate through each phrase in the users query. */
  for(iPhrase=0; iPhrase&lt;nPhrase; iPhrase++){
    int iCol;                     /* Current column */

 /* Now iterate through each column in the users query. For each column,
 ** increment the relevancy score by:
 **
 **   (&lt;hit count&gt; / &lt;global hit count&gt;) * &lt;column weight&gt;
 **
 ** aPhraseinfo[] points to the start of the data for phrase iPhrase. So
 ** the hit count and global hit counts for each column are found in
 ** aPhraseinfo[iCol*3] and aPhraseinfo[iCol*3+1], respectively.
 */
    int *aPhraseinfo = &aMatchinfo[2 + iPhrase*nCol*3];
    for(iCol=0; iCol&lt;nCol; iCol++){
      int nHitCount = aPhraseinfo[3*iCol];
      int nGlobalHitCount = aPhraseinfo[3*iCol+1];
      double weight = sqlite3_value_double(apVal[iCol+1]);
      if( nHitCount&gt;0 ){
        score += ((double)nHitCount / (double)nGlobalHitCount) * weight;
      }
    }
  }

  sqlite3_result_double(pCtx, score);
  return;

 /* Jump here if the wrong number of arguments are passed to this function */
wrong_number_args:
  sqlite3_result_error(pCtx, "wrong number of arguments to function rank()", -1);
}
```

Indexes On Expressions

Normally, an SQL index references columns of a table. But an index can also be formed on expressions involving table columns.

As an example, consider the following table that tracks dollar-amount changes on various "accounts":

```
CREATE TABLE account_change(
  chng_id INTEGER PRIMARY KEY,
  acct_no INTEGER REFERENCES account,
  location INTEGER REFERENCES locations,
  amt INTEGER,  -- in cents
  authority TEXT,
  comment TEXT
);
CREATE INDEX acctchng_magnitude ON account_change(acct_no, abs(amt));
```

Each entry in the account_change table records a deposit or a withdrawl into an account. Deposits have a positive "amt" and withdrawls have a negative "amt".

The acctchng_magnitude index is over the account number ("acct_no") and on the absolute value of the amount. This index allows one to do efficient queries over the magnitude of a change to the account. For example, to list all changes to account number $xyz that are more than $100.00, one can say:

```
SELECT * FROM account_change WHERE acct_no=$xyz AND abs(amt)&gt;=10000;
```

Or, to list all changes to one particular account ($xyz) in order of decreasing magnitude, one can write:

```
SELECT * FROM account_change WHERE acct_no=$xyz
 ORDER BY abs(amt) DESC;
```

Both of the above example queries would work fine without the acctchng_magnitude index. The acctchng_magnitude index index merely helps the queries to run faster, especially on databases where there are many entries in the table for each account.

How To Use Indexes On Expressions

Use a CREATE INDEX statement to create a new index on one or more expressions just like you would to create an index on columns. The only difference is that expressions are listed as the elements to be indexed rather than column names.

The SQLite query planner will consider using an index on an expression when the expression that is indexed appears in the WHERE clause or in the ORDER BY clause of a query, *exactly* as it is written in the CREATE INDEX statement. The query planner does not do algebra. In order to match WHERE clause constraints and ORDER BY terms to indexes, SQLite requires that the expressions be the same, except for minor syntactic differences such as white-space changes. So if you have:

```
CREATE TABLE t2(x,y,z);
CREATE INDEX t2xy ON t2(x+y);
```

And then you run the query:

```
SELECT * FROM t2 WHERE y+x=22;
```

Then the index will not be used because the expression on the CREATE INDEX statement (x+y) is not the same as the expression as it appears in the query (y+x). The two expressions might be mathematically equivalent, but the SQLite query planner insists that they be the same, not merely equivalent. Consider rewriting the query thusly:

```
SELECT * FROM t2 WHERE x+y=22;
```

This second query will likely use the index because now the expression in the WHERE clause (x+y) matches the expression in the index exactly.

Restrictions

There are certain reasonable restrictions on expressions that appear in CREATE INDEX statements:

1. Expressions in CREATE INDEX statements may only refer to columns of the table being indexed, not to columns in other tables.

2. Expressions in CREATE INDEX statmeent may contain function calls, but only to functions whose output is always determined completely by its input parameters (a.k.a.: "deterministic" functions). Obviously, functions like random() will not work well in an index. But also functions like sqlite_version(), though they are constant across any one database connection, are not constant across the life of the underlying database file, and hence may not be used in a CREATE INDEX statement.

   Note that application-defined SQL functions are by default considered non-deterministic and may not be used in a CREATE INDEX statement unless the SQLITE_DETERMINISTIC flag is used when the function is registered.

3. Expressions in CREATE INDEX statements may not use subqueries.
4. Expressions may only be used in CREATE INDEX statements, not within UNIQUE or PRIMARY KEY constraints within the CREATE TABLE statement.

Compatibility

The ability to index expressions was added to SQLite with version 3.9.0 in October of 2015. A database that uses an index on expressions will not be usable by earlier versions of SQLite.

Internal Versus External BLOBs in SQLite

If you have a database of large BLOBs, do you get better read performance when you store the complete BLOB content directly in the database or is it faster to store each BLOB in a separate file and store just the corresponding filename in the database?

To try to answer this, we ran 49 test cases with various BLOB sizes and SQLite page sizes on a Linux workstation (Ubuntu circa 2011 with the Ext4 filesystem on a fast SATA disk). For each test case, a database was created that contains 100MB of BLOB content. The sizes of the BLOBs ranged from 10KB to 1MB. The number of BLOBs varied in order to keep the total BLOB content at about 100MB. (Hence, 100 BLOBs for the 1MB size and 10000 BLOBs for the 10K size and so forth.) SQLite version 3.7.8 was used.

The matrix below shows the time needed to read BLOBs stored in separate files divided by the time needed to read BLOBs stored entirely in the database. Hence, for numbers larger than 1.0, it is faster to store the BLOBs directly in the database. For numbers smaller than 1.0, it is faster to store the BLOBs in separate files.

In every case, the pager cache size was adjusted to keep the amount of cache memory at about 2MB. For example, a 2000 page cache was used for 1024 byte pages and a 31 page cache was used for 65536 byte pages. The BLOB values were read in a random order.

Database Page Size	BLOB size						
10k	20k	50k	100k	200k	500k	1m	
1024	1.535	1.020	0.608	0.456	0.330	0.247	0.233
2048	2.004	1.437	0.870	0.636	0.483	0.372	0.340
4096	2.261	1.886	1.173	0.890	0.701	0.526	0.487
8192	2.240	1.866	1.334	1.035	0.830	0.625	0.720
16384	2.439	1.757	1.292	1.023	0.829	0.820	0.598
32768	1.878	1.843	1.296	0.981	0.976	0.675	0.613
65536	1.256	1.255	1.339	0.983	0.769	0.687	0.609

We deduce the following rules of thumb from the matrix above:

- A database page size of 8192 or 16384 gives the best performance for large BLOB I/O.
- For BLOBs smaller than 100KB, reads are faster when the BLOBs are stored directly in the database file. For BLOBs larger than 100KB, reads from a separate file are faster.

Of course, your mileage may vary depending on hardware, filesystem, and operating system. Double-check these figures on target hardware before committing to a particular design.

Limits In SQLite

"Limits" in the context of this article means sizes or quantities that can not be exceeded. We are concerned with things like the maximum number of bytes in a BLOB or the maximum number of columns in a table.

SQLite was originally designed with a policy of avoiding arbitrary limits. Of course, every program that runs on a machine with finite memory and disk space has limits of some kind. But in SQLite, those limits were not well defined. The policy was that if it would fit in memory and you could count it with a 32-bit integer, then it should work.

Unfortunately, the no-limits policy has been shown to create problems. Because the upper bounds were not well defined, they were not tested, and bugs (including possible security exploits) were often found when pushing SQLite to extremes. For this reason, newer versions of SQLite have well-defined limits and those limits are tested as part of the test suite.

This article defines what the limits of SQLite are and how they can be customized for specific applications. The default settings for limits are normally quite large and adequate for almost every application. Some applications may want to increase a limit here or there, but we expect such needs to be rare. More commonly, an application might want to recompile SQLite with much lower limits to avoid excess resource utilization in the event of bug in higher-level SQL statement generators or to help thwart attackers who inject malicious SQL statements.

Some limits can be changed at run-time on a per-connection basis using the sqlite3_limit() interface with one of the limit categories defined for that interface. Run-time limits are designed for applications that have multiple databases, some of which are for internal use only and others which can be influenced or controlled by potentially hostile external agents. For example, a web browser application might use an internal database to track historical page views but have one or more separate databases that are created and controlled by javascript applications that are downloaded from the internet. The sqlite3_limit() interface allows internal databases managed by trusted code to be unconstrained while simultaneously placing tight limitations on databases created or controlled by untrusted external code in order to help prevent a denial of service attack.

1. **Maximum length of a string or BLOB**

   The maximum number of bytes in a string or BLOB in SQLite is defined by the preprocessor macro SQLITE_MAX_LENGTH. The default value of this macro is 1 billion (1 thousand million or 1,000,000,000). You can raise or lower this value at compile-time using a command-line option like this:

```
-DSQLITE_MAX_LENGTH=123456789
```

The current implementation will only support a string or BLOB length up to 2<small><sup>31</sup></small>-1 or 2147483647. And some built-in functions such as hex() might fail well before that point. In security-sensitive applications it is best not to try to increase the maximum string and blob length. In fact, you might do well to lower the maximum string and blob length to something more in the range of a few million if that is possible.

During part of SQLite's INSERT and SELECT processing, the complete content of each row in the database is encoded as a single BLOB. So the SQLITE_MAX_LENGTH parameter also determines the maximum number of bytes in a row.

The maximum string or BLOB length can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_LENGTH,size) interface.

2. **Maximum Number Of Columns**

The SQLITE_MAX_COLUMN compile-time parameter is used to set an upper bound on:

- The number of columns in a table
- The number of columns in an index
- The number of columns in a view
- The number of terms in the SET clause of an UPDATE statement
- The number of columns in the result set of a SELECT statement
- The number of terms in a GROUP BY or ORDER BY clause
- The number of values in an INSERT statement

The default setting for SQLITE_MAX_COLUMN is 2000. You can change it at compile time to values as large as 32767. On the other hand, many experienced database designers will argue that a well-normalized database will never need more than 100 columns in a table.

In most applications, the number of columns is small - a few dozen. There are places in the SQLite code generator that use algorithms that are $O(N^2)$ where N is the number of columns. So if you redefine SQLITE_MAX_COLUMN to be a really huge number and you generate SQL that uses a large number of columns, you may find that sqlite3_prepare_v2() runs slowly.

The maximum number of columns can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_COLUMN,size) interface.

3. **Maximum Length Of An SQL Statement**

The maximum number of bytes in the text of an SQL statement is limited to SQLITE_MAX_SQL_LENGTH which defaults to 1000000. You can redefine this limit to be as large as the smaller of SQLITE_MAX_LENGTH and 1073741824.

If an SQL statement is limited to be a million bytes in length, then obviously you will not be able to insert multi-million byte strings by embedding them as literals inside of INSERT statements. But you should not do that anyway. Use host parameters for your data. Prepare short SQL statements like this:

```
INSERT INTO tab1 VALUES(?,?,?);
```

Then use the sqlite3_bind_XXXX() functions to bind your large string values to the SQL statement. The use of binding obviates the need to escape quote characters in the string, reducing the risk of SQL injection attacks. It is also runs faster since the large string does not need to be parsed or copied as much.

The maximum length of an SQL statement can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_SQL_LENGTH,size) interface.

4. Maximum Number Of Tables In A Join

SQLite does not support joins containing more than 64 tables. This limit arises from the fact that the SQLite code generator uses bitmaps with one bit per join-table in the query optimizer.

SQLite uses an efficient query planner algorithm and so even a large join can be prepared quickly. Hence, there is no mechanism to raise or lower the limit on the number of tables in a join.

5. Maximum Depth Of An Expression Tree

SQLite parses expressions into a tree for processing. During code generation, SQLite walks this tree recursively. The depth of expression trees is therefore limited in order to avoid using too much stack space.

The SQLITE_MAX_EXPR_DEPTH parameter determines the maximum expression tree depth. If the value is 0, then no limit is enforced. The current implementation has a default value of 1000.

The maximum depth of an expression tree can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_EXPR_DEPTH,size) interface if the SQLITE_MAX_EXPR_DEPTH is initially positive. In other words, the maximum expression depth can be lowered at run-time if there is already a compile-time limit on the expression depth. If SQLITE_MAX_EXPR_DEPTH is set to 0 at compile time (if the depth of expressions is unlimited) then the sqlite3_limit(db,SQLITE_LIMIT_EXPR_DEPTH,size) is a no-op.

6. **Maximum Number Of Arguments On A Function**

   The SQLITE_MAX_FUNCTION_ARG parameter determines the maximum number of parameters that can be passed to an SQL function. The default value of this limit is 100. SQLite should work with functions that have thousands of parameters. However, we suspect that anybody who tries to invoke a function with more than a few parameters is really trying to find security exploits in systems that use SQLite, not do useful work, and so for that reason we have set this parameter relatively low.

   The number of arguments to a function is sometimes stored in a signed character. So there is a hard upper bound on SQLITE_MAX_FUNCTION_ARG of 127.

   The maximum number of arguments in a function can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_FUNCTION_ARG,size) interface.

7. **Maximum Number Of Terms In A Compound SELECT Statement**

   A compound SELECT statement is two or more SELECT statements connected by operators UNION, UNION ALL, EXCEPT, or INTERSECT. We call each individual SELECT statement within a compound SELECT a "term".

   The code generator in SQLite processes compound SELECT statements using a recursive algorithm. In order to limit the size of the stack, we therefore limit the number of terms in a compound SELECT. The maximum number of terms is SQLITE_MAX_COMPOUND_SELECT which defaults to 500. We think this is a generous allotment since in practice we almost never see the number of terms in a compound select exceed single digits.

   The maximum number of compound SELECT terms can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_COMPOUND_SELECT,size) interface.

8. **Maximum Length Of A LIKE Or GLOB Pattern**

   The pattern matching algorithm used in the default LIKE and GLOB implementation of SQLite can exhibit O(N^2) performance (where N is the number of characters in the pattern) for certain pathological cases. To avoid denial-of-service attacks from miscreants who are able to specify their own LIKE or GLOB patterns, the length of the LIKE or GLOB pattern is limited to SQLITE_MAX_LIKE_PATTERN_LENGTH bytes. The default value of this limit is 50000. A modern workstation can evaluate even a pathological LIKE or GLOB pattern of 50000 bytes relatively quickly. The denial of service problem only comes into play when the pattern length gets into millions of bytes.

Nevertheless, since most useful LIKE or GLOB patterns are at most a few dozen bytes in length, paranoid application developers may want to reduce this parameter to something in the range of a few hundred if they know that external users are able to generate arbitrary patterns.

The maximum length of a LIKE or GLOB pattern can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_LIKE_PATTERN_LENGTH,size) interface.

9. **Maximum Number Of Host Parameters In A Single SQL Statement**

A host parameter is a place-holder in an SQL statement that is filled in using one of the sqlite3_bind_XXXX() interfaces. Many SQL programmers are familiar with using a question mark ("?") as a host parameter. SQLite also supports named host parameters prefaced by ":", "$", or "@" and numbered host parameters of the form "?123".

Each host parameter in an SQLite statement is assigned a number. The numbers normally begin with 1 and increase by one with each new parameter. However, when the "?123" form is used, the host parameter number is the number that follows the question mark.

SQLite allocates space to hold all host parameters between 1 and the largest host parameter number used. Hence, an SQL statement that contains a host parameter like ?1000000000 would require gigabytes of storage. This could easily overwhelm the resources of the host machine. To prevent excessive memory allocations, the maximum value of a host parameter number is SQLITE_MAX_VARIABLE_NUMBER, which defaults to 999.

The maximum host parameter number can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_VARIABLE_NUMBER,size) interface.

10. **Maximum Depth Of Trigger Recursion**

SQLite limits the depth of recursion of triggers in order to prevent a statement involving recursive triggers from using an unbounded amount of memory.

Prior to SQLite version 3.6.18, triggers were not recursive and so this limit was meaningless. Beginning with version 3.6.18, recursive triggers were supported but had to be explicitly enabled using the PRAGMA recursive_triggers statement. Beginning with version 3.7.0, recursive triggers are enabled by default but can be manually disabled using PRAGMA recursive_triggers. The SQLITE_MAX_TRIGGER_DEPTH is only meaningful if recursive triggers are enabled.

The default maximum trigger recursion depth is 1000.

11. **Maximum Number Of Attached Databases**

The ATTACH statement is an SQLite extension that allows two or more databases to be associated to the same database connection and to operate as if they were a single database. The number of simultaneously attached databases is limited to SQLITE_MAX_ATTACHED which is set to 10 by default. The maximum number of attached databases cannot be increased above 125.

The maximum number of attached databases can be lowered at run-time using the sqlite3_limit(db,SQLITE_LIMIT_ATTACHED,size) interface.

12. Maximum Number Of Pages In A Database File

SQLite is able to limit the size of a database file to prevent the database file from growing too large and consuming too much disk space. The SQLITE_MAX_PAGE_COUNT parameter, which is normally set to 1073741823, is the maximum number of pages allowed in a single database file. An attempt to insert new data that would cause the database file to grow larger than this will return SQLITE_FULL.

The largest possible setting for SQLITE_MAX_PAGE_COUNT is 2147483646. When used with the maximum page size of 65536, this gives a maximum SQLite database size of about 140 terabytes.

The max_page_count PRAGMA can be used to raise or lower this limit at run-time.

13. Maximum Number Of Rows In A Table

The theoretical maximum number of rows in a table is 2<sup><small>64</small></sup> (18446744073709551616 or about 1.8e+19). This limit is unreachable since the maximum database size of 140 terabytes will be reached first. A 140 terabytes database can hold no more than approximately 1e+13 rows, and then only if there are no indices and if each row contains very little data.

14. Maximum Database Size

Every database consists of one or more "pages". Within a single database, every page is the same size, but different database can have page sizes that are powers of two between 512 and 65536, inclusive. The maximum size of a database file is 2147483646 pages. At the maximum page size of 65536 bytes, this translates into a maximum database size of approximately 1.4e+14 bytes (140 terabytes, or 128 tebibytes, or 140,000 gigabytes or 128,000 gibibytes).

This particular upper bound is untested since the developers do not have access to hardware capable of reaching this limit. However, tests do verify that SQLite behaves correctly and sanely when a database reaches the maximum file size of the underlying filesystem (which is usually much less than the maximum theoretical database size) and when a database is unable to grow due to disk space exhaustion.

15. Maximum Number Of Tables In A Schema

Each table and index requires at least one page in the database file. An "index" in the previous sentence means an index created explicitly using a CREATE INDEX statement or implicit indices created by UNIQUE and PRIMARY KEY constraints. Since the maximum number of pages in a database file is 2147483646 (a little over 2 billion) this is also then an upper bound on the number of tables and indices in a schema.

Whenever a database is opened, the entire schema is scanned and parsed and a parse tree for the schema is held in memory. That means that database connection startup time and initial memory usage is proportional to the size of the schema.

Memory-Mapped I/O

The default mechanism by which SQLite accesses and updates database disk files is the xRead() and xWrite() methods of the sqlite3_io_methods VFS object. These methods are typically implemented as "read()" and "write()" system calls which cause the operating system to copy disk content between the kernel buffer cache and user space.

Beginning with version 3.7.17, SQLite has the option of accessing disk content directly using memory-mapped I/O and the new xFetch() and xUnfetch() methods on sqlite3_io_methods.

There are advantages and disadvantages to using memory-mapped I/O. Advantages include:

1. Many operations, especially I/O intensive operations, can be much faster since content does need to be copied between kernel space and user space. In some cases, performance can nearly double.

2. The SQLite library may need less RAM since it shares pages with the operating-system page cache and does not always need its own copy of working pages.

But there are also disadvantages:

1. A stray pointer or buffer overflow in the application program might change the content of mapped memory, potentially corrupting the database file. Bugs in application program that overwrite memory assigned to the SQLite library could, in theory, also cause corruption in read/write mode, but that would require that the overwrite be followed by a call to xWrite() and in practice the buggy application usually errors out or crashes before that point. With memory-mapped I/O, the database corruption occurs immediately and is thus more of a risk.

2. An I/O error on a memory-mapped file cannot be caught and dealt with by SQLite. Instead, the I/O error causes a signal which, if not caught by the application, results in a program crash.

3. The operating system must have a unified buffer cache in order for the memory-mapped I/O extension to work correctly, especially in situations where two processes are accessing the same database file and one process is using memory-mapped I/O while the other is not. Not all operating systems have a unified buffer cache. In some operating systems that claim to have a unified buffer cache, the implementation is buggy and can lead to corrupt databases.

4. Performance does not always increase with memory-mapped I/O. In fact, it is possible to construct test cases where performance is reduced by the use of memory-mapped I/O, though this is hard to do.

5. Windows is unable to truncate a memory-mapped file. Hence, on Windows, if an operation such as VACUUM or auto_vacuum tries to reduce the size of a memory-mapped database file, the size reduction attempt will silently fail, leaving unused space at the end of the database file. No data is lost due to this problem, and the unused space will be reused again the next time the database grows. However if a version of SQLite prior to 3.7.0 runs PRAGMA integrity_check on such a database, it will (incorrectly) report database corruption due to the unused space at the end. Or if a version of SQLite prior to 3.7.0 writes to the database while it still has unused space at the end, it may make that unused space inaccessible and unavailable for reuse until after the next VACUUM.

Because of the potential disadvantages, memory-mapped I/O is turned off by default. To activate memory-mapped I/O, use the mmap_size pragma and set the mmap_size to some large number, usually 256MB or larger, depending on how much address space your application can spare. The rest is automatic. The PRAGMA mmap_size statement will be a silent no-op on systems that do not support memory-mapped I/O.

How Memory-Mapped I/O Works

To read a page of database content using the legacy xRead() method, SQLite first allocates a page-size chunk of heap memory then invokes the xRead() method which causes the database page content to be copied into the newly allocated heap memory. This involves (at a minimum) a copy of the entire page.

But if SQLite wants to access a page of the database file and memory mapped I/O is enabled, it first calls the xFetch() method. The xFetch() method asks the operating system to return a pointer to the requested page, if possible. If the requested page has been or can be mapped into the application address space, then xFetch returns a pointer to that page for SQLite to use without having to copy anything. Skipping the copy step is what makes memory mapped I/O faster.

SQLite does not assume that the xFetch() method will work. If a call to xFetch() returns a NULL pointer (indicating that the requested page is not currently mapped into the applications address space) then SQLite silently falls back to using xRead(). An error is only reported if xRead() also fails.

When updating the database file, SQLite always makes a copy of the page content into heap memory before modifying the page. This is necessary since the changes are not supposed to be visible to other processes until after the transaction commits and so the changes must occur in private memory. After all needed changes are completed, xWrite() is used to move the content back into the database file. The current xWrite() implementations for both unix and windows check to see if section of the file being written is mapped into the applications address space, and if it is the write operation is implemented using memcpy() rather than invoking a "write()" system call, but that is just an implementation detail. A memory copy occurs either way. So the use of memory mapped I/O does not significantly change the performance of database changes. Memory mapped I/O is mostly a benefit for queries.

Configuring Memory-Mapped I/O

The "mmap_size" is the maximum number of bytes of the database file that SQLite will try to map into the process address space at one time. The mmap_size applies separately to each database file, so the total amount of process address space that could potentially be used is the mmap_size times the number of open database files.

To activate memory-mapped I/O, an application can set the mmap_size to some large value. For example:

```
PRAGMA mmap_size=268435456;
```

To disable memory-mapped I/O, simply set the mmap_size to zero:

```
PRAGMA mmap_size=0;
```

If mmap_size is set to N then all current implementations map the first N bytes of the database file and use legacy xRead() calls for any content beyond N bytes. If the database file is smaller than N bytes, then the entire file is mapped. In the future, new OS interfaces could, in theory, map regions of the file other than the first N bytes, but no such implementation currently exists.

The mmap_size is set separately for each database file using the "PRAGMA mmap_size" statement. The usual default mmap_size is zero, meaning that memory mapped I/O is disabled by default. However, the default mmap_size can be increased either at compile-time using the SQLITE_DEFAULT_MMAP_SIZE macro or at start-time using the sqlite3_config(SQLITE_CONFIG_MMAP_SIZE,...) interface.

SQLite also maintains a hard upper bound on the mmap_size. Attempts to increase the mmap_size above this hard upper bound (using PRAGMA mmap_size) will automatically cap the mmap_size at the hard upper bound. If the hard upper bound is zero, then memory mapped I/O is impossible. The hard upper bound can be set at compile-time using the SQLITE_MAX_MMAP_SIZE macro. If SQLITE_MAX_MMAP_SIZE is set to zero, then the code used to implement memory mapped I/O is omitted from the build. The hard upper bound is automatically set to zero on certain platforms (ex: OpenBSD) where memory mapped I/O does not work due to the lack of a unified buffer cache.

If the hard upper bound on mmap_size is non-zero at compilation time, it may still be reduced or zeroed at start-time using the sqlite3_config(SQLITE_CONFIG_MMAP_SIZE,X,Y) interface. The X and Y parameters must both be 64-bit signed integers. The X parameter is the default mmap_size of the process and the Y is the new hard upper bound. The hard upper bound cannot be increased above its compile-time setting using SQLITE_CONFIG_MMAP_SIZE but it can be reduced or zeroed.

SQLite And Multiple Threads

SQLite support three different threading modes:

1. **Single-thread**. In this mode, all mutexes are disabled and SQLite is unsafe to use in more than a single thread at once.
2. **Multi-thread**. In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
3. **Serialized**. In serialized mode, SQLite can be safely used by multiple threads with no restriction.

The threading mode can be selected at compile-time (when the SQLite library is being compiled from source code) or at start-time (when the application that intends to use SQLite is initializing) or at run-time (when a new SQLite database connection is being created). Generally speaking, run-time overrides start-time and start-time overrides compile-time. Except, single-thread mode cannot be overridden once selected.

The default mode is serialized.

Compile-time selection of threading mode

Use the SQLITE_THREADSAFE compile-time parameter to selected the threading mode. If no SQLITE_THREADSAFE compile-time parameter is present, then serialized mode is used. This can be made explicit with -DSQLITE_THREADSAFE=1. With -DSQLITE_THREADSAFE=0 the threading mode is single-thread. With -DSQLITE_THREADSAFE=2 the threading mode is multi-thread.

The return value of the sqlite3_threadsafe() interface is determined by the compile-time threading mode selection. If single-thread mode is selected at compile-time, then sqlite3_threadsafe() returns false. If either the multi-thread or serialized modes are selected, then sqlite3_threadsafe() returns true. The sqlite3_threadsafe() interface predates the multi-thread mode and start-time and run-time mode selection and so is unable to distinguish between multi-thread and serialized mode nor is it able to report start-time or run-time mode changes.

If single-thread mode is selected at compile-time, then critical mutexing logic is omitted from the build and it is impossible to enable either multi-thread or serialized modes at start-time or run-time.

Start-time selection of threading mode

Assuming that the compile-time threading mode is not single-thread, then the threading mode can be changed during initialization using the sqlite3_config() interface. The SQLITE_CONFIG_SINGLETHREAD verb puts SQLite into single-thread mode, the SQLITE_CONFIG_MULTITHREAD verb sets multi-thread mode, and the SQLITE_CONFIG_SERIALIZED verb sets serialized mode.

Run-time selection of threading mode

If single-thread mode has not been selected at compile-time or start-time, then individual database connections can be created as either multi-thread or serialized. It is not possible to downgrade an individual database connection to single-thread mode. Nor is it possible to escalate an individual database connection if the compile-time or start-time mode is single-thread.

The threading mode for an individual database connection is determined by flags given as the third argument to sqlite3_open_v2(). The SQLITE_OPEN_NOMUTEX flag causes the database connection to be in the multi-thread mode and the SQLITE_OPEN_FULLMUTEX flag causes the connection to be in serialized mode. If neither flag is specified or if sqlite3_open() or sqlite3_open16() are used instead of sqlite3_open_v2(), then the default mode determined by the compile-time and start-time settings is used.

NULL Handling in SQLite Versus Other Database Engines

The goal is to make SQLite handle NULLs in a standards-compliant way. But the descriptions in the SQL standards on how to handle NULLs seem ambiguous. It is not clear from the standards documents exactly how NULLs should be handled in all circumstances.

So instead of going by the standards documents, various popular SQL engines were tested to see how they handle NULLs. The idea was to make SQLite work like all the other engines. An SQL test script was developed and run by volunteers on various SQL RDBMSes and the results of those tests were used to deduce how each engine processed NULL values. The original tests were run in May of 2002. A copy of the test script is found at the end of this document.

SQLite was originally coded in such a way that the answer to all questions in the chart below would be "Yes". But the experiments run on other SQL engines showed that none of them worked this way. So SQLite was modified to work the same as Oracle, PostgreSQL, and DB2. This involved making NULLs indistinct for the purposes of the SELECT DISTINCT statement and for the UNION operator in a SELECT. NULLs are still distinct in a UNIQUE column. This seems somewhat arbitrary, but the desire to be compatible with other engines outweighed that objection.

It is possible to make SQLite treat NULLs as distinct for the purposes of the SELECT DISTINCT and UNION. To do so, one should change the value of the NULL_ALWAYS_DISTINCT #define in the `sqliteInt.h` source file and recompile.

> *Update 2003-07-13:* Since this document was originally written some of the database engines tested have been updated and users have been kind enough to send in corrections to the chart below. The original data showed a wide variety of behaviors, but over time the range of behaviors has converged toward the PostgreSQL/Oracle model. The only significant difference is that Informix and MS-SQL both treat NULLs as indistinct in a UNIQUE column.
>
> The fact that NULLs are distinct for UNIQUE columns but are indistinct for SELECT DISTINCT and UNION continues to be puzzling. It seems that NULLs should be either distinct everywhere or nowhere. And the SQL standards documents suggest that NULLs should be distinct everywhere. Yet as of this writing, no SQL engine tested treats NULLs as distinct in a SELECT DISTINCT statement or in a UNION.

The following table shows the results of the NULL handling experiments.

	SQLite	PostgreSQL	Oracle	Informix	DB2	MS-SQL	OCEL
Adding anything to null gives null	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multiplying null by zero gives null	Yes	Yes	Yes	Yes	Yes	Yes	Yes
nulls are distinct in a UNIQUE column	Yes	Yes	Yes	No	(Note 4)	No	Yes
nulls are distinct in SELECT DISTINCT	No	No	No	No	No	No	No
nulls are distinct in a UNION	No	No	No	No	No	No	No
"CASE WHEN null THEN 1 ELSE 0 END" is 0?	Yes	Yes	Yes	Yes	Yes	Yes	Yes
"null OR true" is true	Yes	Yes	Yes	Yes	Yes	Yes	Yes
"not (null AND false)" is true	Yes	Yes	Yes	Yes	Yes	Yes	Yes

	MySQL 3.23.41	MySQL 4.0.16	Firebird	SQL Anywhere	Borland Interbase
Adding anything to null gives null	Yes	Yes	Yes	Yes	Yes
Multiplying null by zero gives null	Yes	Yes	Yes	Yes	Yes
nulls are distinct in a UNIQUE column	Yes	Yes	Yes	(Note 4)	(Note 4)
nulls are distinct in SELECT DISTINCT	No	No	No (Note 1)	No	No
nulls are distinct in a UNION	(Note 3)	No	No (Note 1)	No	No
"CASE WHEN null THEN 1 ELSE 0 END" is 0?	Yes	Yes	Yes	Yes	(Note 5)
"null OR true" is true	Yes	Yes	Yes	Yes	Yes
"not (null AND false)" is true	No	Yes	Yes	Yes	Yes

Notes:	
1.	Older versions of firebird omits all NULLs from SELECT DISTINCT and from UNION.
2.	Test data unavailable.
3.	MySQL version 3.23.41 does not support UNION.
4.	DB2, SQL Anywhere, and Borland Interbase do not allow NULLs in a UNIQUE column.
5.	Borland Interbase does not support CASE expressions.

The following script was used to gather information for the table above.

```
-- I have about decided that SQL's treatment of NULLs is capricious and cannot be
-- deduced by logic.  It must be discovered by experiment.  To that end, I have
-- prepared the following script to test how various SQL databases deal with NULL.
-- My aim is to use the information gather from this script to make SQLite as much
-- like other databases as possible.
--
-- If you could please run this script in your database engine and mail the results
-- to me at drh@hwaci.com, that will be a big help.  Please be sure to identify the
-- database engine you use for this test.  Thanks.
--
-- If you have to change anything to get this script to run with your database
-- engine, please send your revised script together with your results.
--

-- Create a test table with data
create table t1(a int, b int, c int);
insert into t1 values(1,0,0);
insert into t1 values(2,0,1);
insert into t1 values(3,1,0);
insert into t1 values(4,1,1);
insert into t1 values(5,null,0);
insert into t1 values(6,null,1);
insert into t1 values(7,null,null);

-- Check to see what CASE does with NULLs in its test expressions
select a, case when b<>0 then 1 else 0 end from t1;
select a+10, case when not b<>0 then 1 else 0 end from t1;
select a+20, case when b<>0 and c<>0 then 1 else 0 end from t1;
select a+30, case when not (b<>0 and c<>0) then 1 else 0 end from t1;
select a+40, case when b<>0 or c<>0 then 1 else 0 end from t1;
select a+50, case when not (b<>0 or c<>0) then 1 else 0 end from t1;
select a+60, case b when c then 1 else 0 end from t1;
select a+70, case c when b then 1 else 0 end from t1;

-- What happens when you multiple a NULL by zero?
select a+80, b*0 from t1;
select a+90, b*c from t1;

-- What happens to NULL for other operators?
select a+100, b+c from t1;

-- Test the treatment of aggregate operators
select count(*), count(b), sum(b), avg(b), min(b), max(b) from t1;

-- Check the behavior of NULLs in WHERE clauses
select a+110 from t1 where b<10; select="" a+120="" from="" t1="" where="" not="" b="">10
select a+130 from t1 where b<10 or="" c="1;" select="" a+140="" from="" t1="" where="" b<
```

Partial Indexes

1.0 Introduction

A partial index is an index over a subset of the rows of a table.

In ordinary indexes, there is exactly one entry in the index for every row in the table. In partial indexes, only some subset of the rows in the table have corresponding index entries. For example, a partial index might omit entries for which the column being indexed is NULL. When used judiciously, partial indexes can result in smaller database files and improvements in both query and write performance.

2.0 Creating Partial Indexes

Create a partial index by adding a WHERE clause to the end of an ordinary CREATE INDEX statement.

create-index-stmt: <button id="x1437" onclick="hideorshow("x1437","x1438")">hide</button>

expr: <button id="x1439" onclick="hideorshow("x1439","x1440")">show</button>

literal-value: <button id="x1441" onclick="hideorshow("x1441","x1442")">show</button>

raise-function: <button id="x1443" onclick="hideorshow("x1443","x1444")">show</button>

select-stmt: <button id="x1445" onclick="hideorshow("x1445","x1446")">show</button>

common-table-expression: <button id="x1447" onclick="hideorshow("x1447","x1448")">show</button>

compound-operator: <button id="x1449" onclick="hideorshow("x1449","x1450")">show</button>

join-clause: <button id="x1451" onclick="hideorshow("x1451","x1452")">show</button>

join-constraint: <button id="x1453" onclick="hideorshow("x1453","x1454")">show</button>

join-operator: <button id="x1455" onclick="hideorshow("x1455","x1456")">show</button>

ordering-term: <button id="x1457" onclick="hideorshow("x1457","x1458")">show</button>

result-column: <button id="x1459" onclick="hideorshow("x1459","x1460")">show</button>

table-or-subquery: <button id="x1461" onclick="hideorshow("x1461","x1462")">show</button>

type-name: <button id="x1463" onclick="hideorshow("x1463","x1464")">show</button>

signed-number: <button id="x1465" onclick="hideorshow("x1465","x1466")">show</button>

> **indexed-column:** <button id="x1467" onclick="hideorshow("x1467","x1468")">show</button>

Any index that includes the WHERE clause at the end is considered to be a partial index. Indexes that omit the WHERE clause (or indexes that are created by UNIQUE or PRIMARY KEY constraints inside of CREATE TABLE statements) are ordinary full indexes.

The expression following the WHERE clause may contain operators, literal values, and names of columns in the table being indexed. The WHERE clause may *not* contains subqueries, references to other tables, functions, or bound parameters. The LIKE, GLOB, MATCH, and REGEXP operators in SQLite are implemented as functions by the same name. Since functions are prohibited in the WHERE clause of a CREATE INDEX statement, so too are the LIKE, GLOB, MATCH, and REGEXP operators.

Only rows of the table for which the WHERE clause evaluates to true are included in the index. If the WHERE clause expression evaluates to NULL or to false for some rows of the table, then those rows are omitted from the index.

The columns referenced in the WHERE clause of a partial index can be any of the columns in the table, not just columns that happen to be indexed. However, it is very common for the WHERE clause expression of a partial index to be a simple expression on the column being indexed. The following is a typical example:

> CREATE INDEX po_parent ON purchaseorder(parent_po) WHERE parent_po IS NOT NULL;

In the example above, if most purchase orders do not have a "parent" purchase order, then most parent_po values will be NULL. That means only a small subset of the rows in the purchaseorder table will be indexed. Hence the index will take up much less space. And changes to the original purchaseorder table will run faster since the po_parent index only needs to be updated for those exceptional rows where parent_po is not NULL. But the index is still useful for querying. In particular, if one wants to know all "children" of a particular purchase order "?1", the query would be:

> SELECT po_num FROM purchaseorder WHERE parent_po=?1;

The query above will use the po_parent index to help find the answer, since the po_parent index contains entries for all rows of interest. Note that since po_parent is smaller than a full index, the query will likely run faster too.

2.1 Unique Partial Indexes

A partial index definition may include the UNIQUE keyword. If it does, then SQLite requires every entry *in the index* to be unique. This provides a mechanism for enforcing uniqueness across some subset of the rows in a table.

For example, suppose you have a database of the members of a large organization where each person is assigned to a particular "team". Each team has a "leader" who is also a member of that team. The table might look something like this:

```
CREATE TABLE person(
  person_id       INTEGER PRIMARY KEY,
  team_id         INTEGER REFERENCES team,
  is_team_leader  BOOLEAN,
  -- other fields elided
);
```

The team_id field cannot be unique because there usually multiple people on the same team. One cannot make the combination of team_id and is_team_leader unique since there are usually multiple non-leaders on each team. The solution to enforcing one leader per team is to create a unique index on team_id but restricted to those entries for which is_team_leader is true:

```
CREATE UNIQUE INDEX team_leader ON person(team_id) WHERE is_team_leader;
```

Coincidentally, that same index is useful for locating the team leader of a particular team:

```
SELECT person_id FROM person WHERE is_team_leader AND team_id=?1;
```

3.0 Queries Using Partial Indexes

Let X be the expression in the WHERE clause of a partial index, and let W be the WHERE clause of a query that uses the table that is indexed. Then, the query is permitted to use the partial index if W⇒X, where the ⇒ operator (usually pronounced "implies") is the logic operator equivalent to "X or not W". Hence, determining whether or not a partial index is usable in a particular query reduces to proving a theorem in first-order logic.

SQLite does not have a sophisticated theorem prover with which to determine W⇒X. Instead, SQLite uses two simple rules to find the common cases where W⇒X is true, and it assumes all the other cases are false. The rules used by SQLite are these:

1. If W is AND-connected terms and X is OR-connected terms and if any term of W appears as a term of X, then the partial index is usable.

   For example, let the index be

   ```
   CREATE INDEX ex1 ON tab1(a,b) WHERE a=5 OR b=6;
   ```

And let the query be:

```
SELECT * FROM tab1 WHERE b=6 AND a=7; -- uses partial index
```

Then the index is usable by the query because the "b=6" term appears in both the index definition and in the query. Remember: terms in the index should be OR-connected and terms in the query should be AND-connected.

The terms in W and X must match exactly. SQLite does not do algebra to try to get them to look the same. The term "b=6" does not match "b=3+3" or "b-6=0" or "b BETWEEN 6 AND 6". "b=6" will match to "6=b" as long as "b=6" is on the index and "6=b" is in the query. If a term of the form "6=b" appears in the index, it will never match anything.

2. If a term in X is of the form "z IS NOT NULL" and if a term in W is a comparison operator on "z" other than "IS", then those terms match.

   Example: Let the index be

   ```
   CREATE INDEX ex2 ON tab2(b,c) WHERE c IS NOT NULL;
   ```

   Then any query that uses operators =, <, >, <=, >=, <>, or IN on column "c" would be usable with the partial index because those comparison operators are only true if "c" is not NULL. So the following query could use the partial index:

   ```
   SELECT * FROM tab2 WHERE b=456 AND c<>0; -- uses partial index
   ```

   But the next query can not use the partial index:

   ```
   SELECT * FROM tab2 WHERE b=456; -- cannot use partial index
   ```

   The latter query can not use the partial index because there might be rows in the table with b=456 and where c is NULL. But those rows would not be in the partial index.

These two rules describe how the query planner for SQLite works as of this writing (2013-08-01). And the rules above will always be honored. However, future versions of SQLite might incorporate a better theorem prover that can find other cases where W⇒X is true and thus may find more instances where partial indexes are useful.

4.0 Supported Versions

Partial indexes have been supported in SQLite since version 3.8.0.

Database files that contain partial indices are not readable or writable by versions of SQLite prior to 3.8.0. However, a database file created by SQLite 3.8.0 is still readable and writable by prior versions as long as its schema contains no partial indexes. A database that is unreadable by legacy versions of SQLite can be made readable simply by running DROP INDEX on the partial indexes.

The SQLite R*Tree Module

1.0 Overview

An R-Tree is a special index that is designed for doing range queries. R-Trees are most commonly used in geospatial systems where each entry is a rectangle with minimum and maximum X and Y coordinates. Given a query rectangle, an R-Tree is able to quickly find all entries that are contained within the query rectangle or which overlap the query rectangle. This idea is easily extended to three dimensions for use in CAD systems. R-Trees also find use in time-domain range look-ups. For example, suppose a database records the starting and ending times for a large number of events. A R-Tree is able to quickly find all events that were active at any time during a given time interval, or all events that started during a particular time interval, or all events that both started and ended within a given time interval. And so forth.

The R-Tree concept originated with Toni Guttman: *R-Trees: A Dynamic Index Structure for Spatial Searching*, Proc. 1984 ACM SIGMOD International Conference on Management of Data, pp. 47-57. The implementation found in SQLite is a refinement of Guttman's original idea, commonly called "R*Trees", that was described by Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger: _The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles._ SIGMOD Conference 1990: 322-331.

2.0 Compiling The R*Tree Module

The source code to the SQLite R*Tree module is included as part of the amalgamation but is disabled by default. To enable the R*Tree module, simply compile with the SQLITE_ENABLE_RTREE C-preprocessor macro defined. With many compilers, this is accomplished by adding the option "-DSQLITE_ENABLE_RTREE=1" to the compiler command-line.

3.0 Using the R*Tree Module

The SQLite R*Tree module is implemented as a virtual table. Each R*Tree index is a virtual table with an odd number of columns between 3 and 11. The first column is always a 64-bit signed integer primary key. The other columns are pairs, one pair per dimension, containing the minimum and maximum values for that dimension, respectively. A 1-dimensional R*Tree thus has 3 columns. A 2-dimensional R*Tree has 5 columns. A 3-dimensional R*Tree has 7 *columns. A 4-dimensional R*Tree has 9 columns. And a 5-dimensional R*Tree has 11 columns. The SQLite R*Tree implementation does not support R*Trees wider than 5 dimensions.

The first column of an SQLite R*Tree is similar to an integer primary key column of a normal SQLite table. It may only store a 64-bit signed integer value. Inserting a NULL value into this column causes SQLite to automatically generate a new unique primary key value. If an attempt is made to insert any other non-integer value into this column, the r-tree module silently converts it to an integer before writing it into the database.

The min/max-value pair columns are stored as 32-bit floating point values for "rtree" virtual tables or as 32-bit signed integers in "rtree_i32" virtual tables. Unlike regular SQLite tables which can store data in a variety of datatypes and formats, the R*Tree rigidly enforce these storage types. If any other type of value is inserted into such a column, the r-tree module silently converts it to the required type before writing the new record to the database.

3.1 Creating An R*Tree Index

A new R*Tree index is created as follows:

```
CREATE VIRTUAL TABLE &lt;name&gt; USING rtree(&lt;column-names&gt;);
```

The <*name*> is the name your application chooses for the R*Tree index and <*column-names*> is a comma separated list of between 3 and 11 columns. The virtual <name> table creates three "shadow" tables to actually store its content. The names of these shadow tables are:

> <*name*>_**node** <*name*>_**rowid** <*name*>_**parent**

The shadow tables are ordinary SQLite data tables. You can query them directly if you like, though this unlikely to reveal anything particularly useful. And you can UPDATE, DELETE, INSERT or even DROP the shadow tables, though doing so will corrupt your R*Tree index. So it is best to simply ignore the shadow tables. Recognize that they hold your R*Tree index information and let it go as that.

As an example, consider creating a two-dimensional R*Tree index for use in spatial queries:

```
CREATE VIRTUAL TABLE demo_index USING rtree(
   id,              -- Integer primary key
   minX, maxX,      -- Minimum and maximum X coordinate
   minY, maxY       -- Minimum and maximum Y coordinate
);
```

3.2 Populating An R*Tree Index

The usual INSERT, UPDATE, and DELETE commands work on an R*Tree index just like on regular tables. So to insert some data into our sample R*Tree index, we can do something like this:

```
INSERT INTO demo_index VALUES(
    1,                   -- Primary key -- SQLite.org headquarters
    -80.7749, -80.7747,  -- Longitude range
    35.3776, 35.3778     -- Latitude range
);
INSERT INTO demo_index VALUES(
    2,                   -- NC 12th Congressional District in 2010
    -81.0, -79.6,
    35.0, 36.2
);
```

The entries above might represent (for example) a bounding box around the main office for SQLite.org and bounding box around the 12th Congressional District of North Carolina (prior to the 2011 redistricting) in which SQLite.org was located.

3.3 Querying An R*Tree Index

Any valid query will work against an R*Tree index. But the R*Tree implementation is designed to make two kinds of queries especially efficient. First, queries against the primary key are efficient:

```
SELECT * FROM demo_index WHERE id=1;
```

Of course, an ordinary SQLite table will also do a query against its integer primary key efficiently, so the previous is no big deal. The real reason for using an R*Tree is so that you can efficiently do inequality queries against the coordinate ranges. To find all elements of the index that are contained within the vicinity of Charlotte, North Carolina, one might do:

```
SELECT id FROM demo_index
 WHERE minX&gt;=-81.08 AND maxX&lt;=-80.58 and="" miny=""&gt;=35.00  AND maxY&lt;=35
```

The query above would very quickly locate the id of 1 even if the R*Tree contained millions of entries. The previous is an example of a "contained-within" query. The R*Tree also supports "overlapping" queries. For example, to find all bounding boxes that overlap the Charlotte area:

```
SELECT id FROM demo_index
 WHERE maxX&gt;=-81.08 AND minX&lt;=-80.58 and="" maxy=""&gt;=35.00  AND minY&lt;=35
```

This second query would find both entry 1 (the SQLite.org office) which is entirely contained within the query box and also the 12th Congressional District which extends well outside the query box but still overlaps the query box.

Note that it is not necessary for all coordinates in an R*Tree index to be constrained in order for the index search to be efficient. One might, for example, want to query all objects that overlap with the 35th parallel:

```
SELECT id FROM demo_index
 WHERE maxY&gt;=35.0  AND minY&lt;=35.0; &lt;="" pre=""&gt;
```

But, generally speaking, the more constraints that the R*Tree module has to work with, and the smaller the bounding box, the faster the results will come back.

3.3 Roundoff Error

By default, coordinates are stored in an R*Tree using 32-bit floating point values. When a coordinate cannot be exactly represented by a 32-bit floating point number, the lower-bound coordinates are rounded down and the upper-bound coordinates are rounded up. Thus, bounding boxes might be slightly larger than specified, but will never be any smaller. This is exactly what is desired for doing the more common "overlapping" queries where the application wants to find every entry in the R*Tree that overlaps a query bounding box. Rounding the entry bounding boxes outward might cause a few extra entries to appears in an overlapping query if the edge of the entry bounding box corresponds to an edge of the query bounding box. But the overlapping query will never miss a valid table entry.

However, for a "contained-within" style query, rounding the bounding boxes outward might cause some entries to be excluded from the result set if the edge of the entry bounding box corresponds to the edge of the query bounding box. To guard against this, applications should expand their contained-within query boxes slightly (by 0.000012%) by rounding down the lower coordinates and rounding up the top coordinates, in each dimension.

4.0 Using R*Trees Effectively

The only information that an R*Tree index stores about an object is its integer ID and its bounding box. Additional information needs to be stored in separate tables and related to the R*Tree index using the primary key. For the example above, one might create an auxiliary table as follows:

```
CREATE TABLE demo_data(
  id INTEGER PRIMARY KEY,  -- primary key
  objname TEXT,            -- name of the object
  objtype TEXT,            -- object type
  boundary BLOB            -- detailed boundary of object
);
```

In this example, the demo_data.boundary field is intended to hold some kind of binary representation of the precise boundaries of the object. The R*Tree index only holds an axis-aligned rectangular boundary for the object. The R*Tree boundary is just an approximation of the true object boundary. So what typically happens is that the R*Tree index is used to narrow a search down to a list of candidate objects and then more detailed and expensive computations are done on each candidate to find if the candidate truly meets the search criteria.

> **Key Point:** An R*Tree index does not normally provide the exact answer but merely reduces the set of potential answers from millions to dozens.

Suppose the demo_data.boundary field holds some proprietary data description of a complex two-dimensional boundary for an object and suppose that the application has used the sqlite3_create_function() interface to created application-defined functions "contained_in" and "overlaps" accepting two demo_data.boundary objects and return true or false. One may assume that "contained_in" and "overlaps" are relatively slow functions that we do not want to invoke too frequently. Then an efficient way to find the name of all objects located within the North Carolina 12th District, one may be to run a query like this:

```
SELECT objname FROM demo_data, demo_index
 WHERE demo_data.id=demo_index.id
   AND contained_in(demo_data.boundary, :boundary)
   AND minX&gt;=-81.0 AND maxX&lt;=-79.6 and="" miny=""&gt;=35.0 AND maxY&lt;=36.2;
```

In the query above, one would presumably bind the binary BLOB description of the precise boundary of the 12th district to the ":boundary" parameter.

Notice how the query above works: The R*Tree index runs in the outer loop to find entries that are contained within the bounding box of longitude -81..-79.6 and latitude 35.0..36.2. For each object identifier found, SQLite looks up the corresponding entry in the demo_data table. It then uses the boundary field from the demo_data table as a parameter to the contained_in() function and if that function returns true, the objname field from the demo_data table is returned as the next row of query result.

One would get the same answer without the use of the R*Tree index using the following simpler query:

```
SELECT objname FROM demo_data
 WHERE contained_in(demo_data.boundary, :boundary);
```

The problem with this latter query is that it must apply the contained_in() function to millions of entries in the demo_data table. The use of the R*Tree in the penultimate query reduces the number of calls to contained_in() function to a small subset of the entire table. The R*Tree index did not find the exact answer itself, it merely limited the search space.

5.0 Integer-Valued R-Trees

The default virtual table ("rtree") normally stores coordinates as single-precision (4-byte) floating point numbers. If integer coordinates are desired, declare the table using "rtree_i32" instead:

```
CREATE VIRTUAL TABLE intrtree USING rtree_i32(id,x0,x1,y0,y1,z0,z1);
```

An rtree_i32 stores coordinates as 32-bit signed integers. But it still using floating point computations internally as part of the r-tree algorithm. For applications running on processors without hardware floating point, it might be desirable to have a pure integer implementation of r-trees. This is accomplished by compiling with the SQLITE_RTREE_INT_ONLY option. When SQLITE_RTREE_INT_ONLY is used, both the "rtree" and the "rtree_i32" virtual tables store 32-bit integers and only integer values are used for internal computations.

6.0 Custom R-Tree Queries

By using standard SQL expressions in the WHERE clause of a SELECT query, a programmer can query for all R*Tree entries that have a spatial relationship (they intersect with or are contained within) a particular bounding-box. Custom R*Tree queries, using the MATCH operator in the WHERE clause of a SELECT, allow the programmer to query for the set of R*Tree entries that intersect any arbitrary region or shape, not just a box. This capability is useful, for example, in computing the subset of objects in the R*Tree that are visible from a camera positioned in 3-D space.

Regions for custom R*Tree queries are defined by R*Tree geometry callbacks implemented by the application and registered with SQLite via a call to one of the following two APIs:

```
int sqlite3_rtree_query_callback(
  sqlite3 *db,
  const char *zQueryFunc,
  int (*xQueryFunc)(sqlite3_rtree_query_info*),
  void *pContext,
  void (*xDestructor)(void*)
);
int sqlite3_rtree_geometry_callback(
  sqlite3 *db,
  const char *zGeom,
  int (*xGeom)(sqlite3_rtree_geometry *, int nCoord, double *aCoord, int *pRes),
  void *pContext
);
```

The sqlite3_rtree_query_callback() became available with SQLite version 3.8.5 and is the preferred interface. The sqlite3_rtree_geometry_callback() is an older and less flexible interface that is supported for backwards compatibility.

A call to one of the above APIs creates a new SQL function named by the second parameter (zQueryFunc or zGeom). When that SQL function appears on the right-hand side of the MATCH operator and the left-hand side of the MATCH operator is any column in the R*Tree virtual table, then the callback defined by the third argument (xQueryFunc or xGeom) is invoked to determine if a particular object or subtree overlaps the desired region.

For example, a query like the following might be used to find all R*Tree entries that overlap with a circle centered a 45.3,22.9 with a radius of 5.0:

```
SELECT id FROM demo_index WHERE id MATCH circle(45.3, 22.9, 5.0)
```

The SQL syntax for custom queries is the same regardless of which interface, sqlite3_rtree_geometry_callback() or sqlite3_rtree_query_callback(), is used to register the SQL function. However, the newer query-style callbacks give the application greater control over how the query proceeds.

6.1 The Legacy xGeom Callback

The legacy xGeom callback is invoked with four arguments. The first argument is a pointer to an sqlite3_rtree_geometry structure which provides information about how the SQL function was invoked. The second argument is the number of coordinates in each r-tree entry, and is always the same for any given R*Tree. The number of coordinates is 2 for a 1-dimensional R*Tree, 4 for a 2-dimensional R*Tree, 6 for a 3-dimensional R*Tree, and so forth. The third argument, aCoord[], is an array of nCoord coordinates that defines a bounding box to be tested. The last argument is a pointer into which the callback result should be written. The result is zero if the bounding-box defined by aCoord[] is completely outside the region defined by the xGeom callback and the result is non-zero if the bounding-box is inside or overlaps with the xGeom region. The xGeom callback should normally return SQLITE_OK. If xGeom returns anything other than SQLITE_OK, then the r-tree query will abort with an error.

The sqlite3_rtree_geometry structure that the first argument to the xGeom callback points to has a structure shown below. The exact same sqlite3_rtree_geometry structure is used for every callback for same MATCH operator in the same query. The contents of the sqlite3_rtree_geometry structure are initialized by SQLite but are not subsequently modified. The callback is free to make changes to the pUser and xDelUser elements of the structure if desired.

```
typedef struct sqlite3_rtree_geometry sqlite3_rtree_geometry;
struct sqlite3_rtree_geometry {
  void *pContext;                 /* Copy of pContext passed to s_r_g_c() */
  int nParam;                     /* Size of array aParam */
  double *aParam;                 /* Parameters passed to SQL geom function */
  void *pUser;                    /* Callback implementation user data */
  void (*xDelUser)(void *);       /* Called by SQLite to clean up pUser */
};
```

The pContext member of the sqlite3_rtree_geometry structure is always set to a copy of the pContext argument passed to sqlite3_rtree_geometry_callback() when the callback is registered. The aParam[] array (size nParam) contains the parameter values passed to the SQL function on the right-hand side of the MATCH operator. In the example "circle" query above, nParam would be set to 3 and the aParam[] array would contain the three values 45.3, 22.9 and 5.0.

The pUser and xDelUser members of the sqlite3_rtree_geometry structure are initially set to NULL. The pUser variable may be set by the callback implementation to any arbitrary value that may be useful to subsequent invocations of the callback within the same query (for example, a pointer to a complicated data structure used to test for region intersection). If the xDelUser variable is set to a non-NULL value, then after the query has finished running SQLite automatically invokes it with the value of the pUser variable as the only argument. In other words, xDelUser may be set to a destructor function for the pUser value.

The xGeom callback always does a depth-first search of the r-tree.

6.2 The New xQueryFunc Callback

The newer xQueryFunc callback receives more information from the r-tree query engine on each call, and it sends more information back to the query engine before it returns. To help keep the interface manageable, the xQueryFunc callback sends and receives information from the query engine as fields in the sqlite3_rtree_query_info structure:

```
struct sqlite3_rtree_query_info {
  void *pContext;                   /* pContext from when function registered */
  int nParam;                       /* Number of function parameters */
  sqlite3_rtree_dbl *aParam;        /* value of function parameters */
  void *pUser;                      /* callback can use this, if desired */
  void (*xDelUser)(void*);          /* function to free pUser */
  sqlite3_rtree_dbl *aCoord;        /* Coordinates of node or entry to check */
  unsigned int *anQueue;            /* Number of pending entries in the queue */
  int nCoord;                       /* Number of coordinates */
  int iLevel;                       /* Level of current node or entry */
  int mxLevel;                      /* The largest iLevel value in the tree */
  sqlite3_int64 iRowid;             /* Rowid for current entry */
  sqlite3_rtree_dbl rParentScore;   /* Score of parent node */
  int eParentWithin;                /* Visibility of parent node */
  int eWithin;                      /* OUT: Visiblity */
  sqlite3_rtree_dbl rScore;         /* OUT: Write the score here */
};
```

The first five fields of the sqlite3_rtree_query_info structure are identical to the sqlite3_rtree_geometry structure, and have exactly the same meaning. The sqlite3_rtree_query_info structure also contains nCoord and aCoord fields which have the same meaning as the parameter of the same name in the xGeom callback.

The xQueryFunc must set the eWithin field of sqlite3_rtree_query_info to on of the values NOT_WITHIN, PARTLY_WITHIN, or FULLY_WITHIN depending on whether or not the bounding box defined by aCoord[] is completely outside the region, overlaps the region, or is completely inside the region, respectively. In addition, the xQueryFunc must set the rScore field to a non-negative value that indicates the order in which subtrees and entries of the query should be analyzed and returned. Smaller scores are processed first.

As its name implies, an R*Tree is organized as a tree. Each node of the tree is a bounding box. The root of the tree is a bounding box that encapsulates all elements of the tree. Beneath the root are a number of subtrees (typically 20 or more) each with their own smaller bounding boxes and each containing some subset of the R*Tree entries. The subtrees may have sub-subtrees, and so forth until finally one reaches the leaves of the tree which are the actual R*Tree entries.

An R*Tree query is initialized by making the root node the only entry in a priority queue sorted by rScore. The query proceeds by extracting the entry from the priority queue that has the lowest score. If that entry is a leaf (meaning that it is an actual R*Tree entry and not a subtree) then that entry is returned as one row of the query result. If the extracted priority queue entry is a node (a subtree), then sub-subtrees or leaves contained within that entry are passed to the xQueryFunc callback, one by one. Those subelements for which the xQueryFunc callback sets eWithin to PARTLY_WITHIN or FULLY_WITHIN are added to the priority queue using the score supplied by the callback. Subelements that return NOT_WITHIN are discarded. The query runs until the priority queue is empty.

Every leaf entry and node (subtree) within the R*Tree has an integer "level". The leaves have a level of 0. The first containing subtree of the leaves has a level of 1. The root of the R*Tree has the largest level value. The mxLevel entry in the sqlite3_rtree_query_info structure is the level value for the root of the R*Tree. The iLevel entry in sqlite3_rtree_query_info gives the level for the object being interrogated.

Most R*Tree queries use a depth-first search. This is accomplished by setting the rScore equal to iLevel. A depth-first search is usually preferred since it minimizes the number of elements in the priority queue, which reduces memory requirements and speeds processing. However, some application may prefer a breadth-first search, which can be accomplished by setting rScore to mxLevel-iLevel. By creating more complex formulas for rScore, applications can exercise detailed control over the order in which subtree are searched and leaf R*Tree entries are returned. For example, in an application with many millions of R*Tree entries, the rScore might be arranged so that the largest or most significant entries are returned first, allowing the application to display the most important information quickly, and filling in smaller and less important details as they become available.

Other information fields of the sqlite3_rtree_query_info structure are available for use by the xQueryFunc callback, if desired. The iRowid field is the rowid (the first of the 3 to 11 columns in the R*Tree) for the element being considered. iRowid is only valid for leaves. The eParentWithin and rParentScore values are copies of the eWithin and rScore values from the containing subtree of the current row. The anQueue field is an array of mxLevel+1 unsigned integers that tell the current number of elements in the priority queue at each level.

6.3 Additional Considerations for Custom Queries

The MATCH operator of a custom R*Tree query function must be a top-level AND-connected term of the WHERE clause, or else it will not be usable by the R*Tree query optimizer and the query will not be runnable. If the MATCH operator is connected to other terms of the WHERE clause via an OR operator, for example, the query will fail with an error.

Two or more MATCH operators are allowed in the same WHERE clause, as long as they are connected by AND operators. However, the R*Tree query engine only contains a single priority queue. The priority assigned to each node in the search is the lowest priority returned by any of the MATCH operators.

Run-Time Loadable Extensions

SQLite has the ability to load extensions (including new application-defined SQL functions, collating sequences, virtual tables, and VFSes) at run-time. This feature allows the code for extensions to be developed and tested separately from the application and then loaded on an as-needed basis.

Extensions can also be statically linked with the application. The code template shown below will work just as well as a statically linked extension as it does as a run-time loadable extension except that you should give the entry point function ("sqlite3_extension_init") a different name to avoid name collisions if your application contains two or more extensions.

Loading An Extension

An SQLite extension is a shared library or DLL. To load it, you need to supply SQLite with the name of the file containing the shared library or DLL and an entry point to initialize the extension. In C code, this information is supplied using the sqlite3_load_extension() API. See the documentation on that routine for additional information.

Note that different operating systems use different filename suffixes for their shared libraries. Windows use ".dll", Mac uses ".dylib", and most unixes other than mac use ".so". If you want to make your code portable, you can omit the suffix from the shared library filename and the appropriate suffix will be added automatically by the sqlite3_load_extension() interface.

There is also an SQL function that can be used to load extensions: load_extension(X,Y). It works just like the sqlite3_load_extension() C interface.

Both methods for loading an extension allow you to specify the name of an entry point for the extension. You can leave this argument blank - passing in a NULL pointer for the sqlite3_load_extension() C-language interface or omitting the second argument for the load_extension() SQL interface - and the extension loader logic will attempt to figure out the entry point on its own. It will first try the generic extension name "sqlite3_extension_init". If that does not work, it constructs a entry point using the template "sqlite3_X_init" where the X is replaced by the lowercase equivalent of every ASCII character in the filename after the last "/" and before the first following "." omitting the first three characters if they happen to be "lib". So, for example, if the filename is "/usr/lib/libmathfunc-4.8.so" the entry point name would be "sqlite3_mathfunc_init". Or if the filename is "./SpellFixExt.dll" then the entry point would be called "sqlite3_spellfixext_init".

For security reasons, extension loading is turned off by default. In order to use either the C-language or SQL extension loading functions, one must first enable extension loading using the sqlite3_enable_load_extension() C-language API in your application.

From the command-line shell, extensions can be loaded using the ".load" dot-command. For example:

```
.load ./YourCode
```

Note that the command-line shell program has already enabled extension loading for you (by calling the sqlite3_enable_load_extension() interface as part of its setup) so the command above works without any special switches, setup, or other complications.

The ".load" command with one argument invokes sqlite3_load_extension() with the zProc parameter set to NULL, causing SQLite to first look for an entry point named "sqlite3_extension_init" and then "sqlite3_X_init" where "X" is derived from the filename. If your extension has an entry point with a different name, simply supply that name as the second argument. For example:

```
.load ./YourCode nonstandard_entry_point
```

Compiling A Loadable Extension

Loadable extensions are C-code. To compile them on most unix-like operating systems, the usual command is something like this:

```
gcc -g -fPIC -shared YourCode.c -o YourCode.so
```

Macs are unix-like, but they do not follow the usual shared library conventions. To compile a shared library on a Mac, use a command like this:

```
gcc -g -fPIC -dynamiclib YourCode.c -o YourCode.dylib
```

If when you try to load your library you get back an error message that says "mach-o, but wrong architecture" then you might need to add command-line options "-arch i386" or "arch x86_64" to gcc, depending on how your application is built.

To compile on Windows using MSVC, a command similar to the following will usually work:

```
cl YourCode.c -link -dll -out:YourCode.dll
```

To compile for Windows using MinGW, the command line is just like it is for unix except that the output file suffix is changed to ".dll" and the -fPIC argument is omitted:

```
gcc -g -shared YourCode.c -o YourCode.dll
```

Programming Loadable Extensions

A template loadable extension contains the following three elements:

1. Use "`#include &lt;sqlite3ext.h&gt;`" at the top of your source code files instead of "`#include &lt;sqlite3.h&gt;`".
2. Put the macro "`SQLITE_EXTENSION_INIT1`" on a line by itself right after the "`#include &lt;sqlite3ext.h&gt;`" line.
3. Add an extension loading entry point routine that looks like something the following:

   ```
   #ifdef _WIN32
   __declspec(dllexport)
   #endif
   int sqlite3_extension_init( /* &lt;== change="" this="" name,="" maybe="" *="" s
   ```

   You will do well to customize the name of your entry point to correspond to the name of the shared library you will be generating, rather than using the generic "sqlite3_extension_init" name. Giving your extension a custom entry point name will enable you to statically link two or more extensions into the same program without a linker conflict, if you later decide to use static linking rather than run-time linking. If your shared library ends up being named "YourCode.so" or "YourCode.dll" or "YourCode.dylib" as shown in the compiler examples above, then the correct entry point name would be "sqlite3_yourcode_init".

Here is a complete template extension that you can copy/paste to get started:

```
/* Add your header comment here */
#include &lt;sqlite3ext.h&gt; /* Do not use &lt;sqlite3.h&gt;! */
SQLITE_EXTENSION_INIT1

/* Insert your extension code here */

#ifdef _WIN32
__declspec(dllexport)
#endif
/* TODO: Change the entry point name so that "extension" is replaced by
** text derived from the shared library filename as follows:  Copy every
** ASCII alphabetic character from the filename after the last "/" through
** the next following ".", converting each character to lowercase, and
** discarding the first three characters if they are "lib".
*/
int sqlite3_extension_init(
  sqlite3 *db,
  char **pzErrMsg,
  const sqlite3_api_routines *pApi
){
  int rc = SQLITE_OK;
  SQLITE_EXTENSION_INIT2(pApi);
  /* Insert here calls to
  **     sqlite3_create_function_v2(),
  **     sqlite3_create_collation_v2(),
  **     sqlite3_create_module_v2(), and/or
  **     sqlite3_vfs_register()
  ** to register the new features that your extension adds.
  */
  return rc;
}
```

Many examples of complete and working loadable extensions can be seen in the SQLite source tree in the ext/misc subdirectory.

Statically Linking A Run-Time Loadable Extension

The exact same source code can be used for both a run-time loadable shared library or DLL and as a module that is statically linked with your application. This provides flexibility and allows you to reuse the same code in different ways.

To statically link your extension, simply add the -DSQLITE_CORE compile-time option. The SQLITE_CORE macro causes the SQLITE_EXTENSION_INIT1 and SQLITE_EXTENSION_INIT2 macros to become no-ops. Then modify your application to invoke the entry point directly, passing in a NULL pointer as the third "pApi" parameter.

It is particularly important to use an entry point name that is based on the extension filename, rather than the generic "sqlite3_extension_init" entry point name, if you will be statically linking two or more extensions. If you use the generic name, there will be multiple definitions of the same symbol and the link will fail.

If you will be opening multiple database connections in your application, rather than invoking the extension entry points for each database connection separately, you might want to consider using the sqlite3_auto_extension() interface to register your extensions and to cause them to be automatically started as each database connection is opened. You only have to register each extension once, and you can do so near the beginning of your main() routine. Using the sqlite3_auto_extension() interface to register your extensions makes your extensions work as if they were built into the core SQLite - they are automatically there whenever you open a new database connection without needing to be initialized. Just be sure to complete any configuration you need to accomplish using sqlite3_config() before registering your extensions, since the sqlite3_auto_extension() interface implicitly calls sqlite3_initialize().

Implementation Details

SQLite implements run-time extension loading using the xDlOpen(), xDlError(), xDlSym(), and xDlClose() methods of the sqlite3_vfs object. These methods are implemented using the dlopen() library on unix (which explains why SQLite commonly need to be linked against the "-ldl" library on unix systems) and using LoadLibrary() API on Windows. In a custom VFS for unusual systems, these methods can all be omitted, in which case the run-time extension loading mechanism will not work (though you will still be able to statically link the extension code, assuming the entry pointers are uniquely named). SQLite can be compiled with SQLITE_OMIT_LOAD_EXTENSION to omit the extension loading code from the build.

Shared Cache Mode

1.0 SQLite Shared-Cache Mode

Starting with version 3.3.0, SQLite includes a special "shared-cache" mode (disabled by default) intended for use in embedded servers. If shared-cache mode is enabled and a thread establishes multiple connections to the same database, the connections share a single data and schema cache. This can significantly reduce the quantity of memory and IO required by the system.

In version 3.5.0, shared-cache mode was modified so that the same cache can be shared across an entire process rather than just within a single thread. Prior to this change, there were restrictions on passing database connections between threads. Those restrictions were dropped in 3.5.0 update. This document describes shared-cache mode as of version 3.5.0.

Shared-cache mode changes the semantics of the locking model in some cases. The details are described by this document. A basic understanding of the normal SQLite locking model (see File Locking And Concurrency In SQLite Version 3 for details) is assumed.

2.0 Shared-Cache Locking Model

Externally, from the point of view of another process or thread, two or more database connections using a shared-cache appear as a single connection. The locking protocol used to arbitrate between multiple shared-caches or regular database users is described elsewhere.

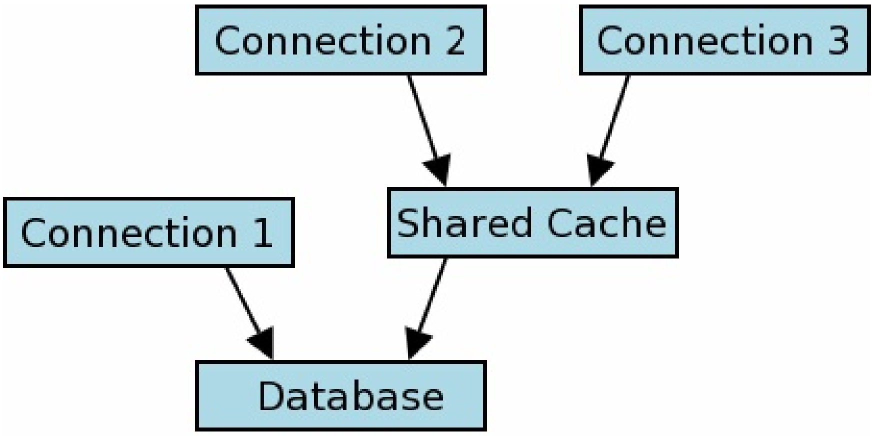


Figure 1

Figure 1 depicts an example runtime configuration where three database connections have been established. Connection 1 is a normal SQLite database connection. Connections 2 and 3 share a cache The normal locking protocol is used to serialize database access between connection 1 and the shared cache. The internal protocol used to serialize (or not, see "Read-Uncommitted Isolation Mode" below) access to the shared-cache by connections 2 and 3 is described in the remainder of this section.

There are three levels to the shared-cache locking model, transaction level locking, table level locking and schema level locking. They are described in the following three sub-sections.

2.1 Transaction Level Locking

SQLite connections can open two kinds of transactions, read and write transactions. This is not done explicitly, a transaction is implicitly a read-transaction until it first writes to a database table, at which point it becomes a write-transaction.

At most one connection to a single shared cache may open a write transaction at any one time. This may co-exist with any number of read transactions.

2.2 Table Level Locking

When two or more connections use a shared-cache, locks are used to serialize concurrent access attempts on a per-table basis. Tables support two types of locks, "read-locks" and "write-locks". Locks are granted to connections - at any one time, each database connection has either a read-lock, write-lock or no lock on each database table.

At any one time, a single table may have any number of active read-locks or a single active write lock. To read data a table, a connection must first obtain a read-lock. To write to a table, a connection must obtain a write-lock on that table. If a required table lock cannot be obtained, the query fails and SQLITE_LOCKED is returned to the caller.

Once a connection obtains a table lock, it is not released until the current transaction (read or write) is concluded.

2.2.1 Read-Uncommitted Isolation Mode

The behaviour described above may be modified slightly by using the read_uncommitted pragma to change the isolation level from serialized (the default), to read-uncommitted.

A database connection in read-uncommitted mode does not attempt to obtain read-locks before reading from database tables as described above. This can lead to inconsistent query results if another database connection modifies a table while it is being read, but it also means that a read-transaction opened by a connection in read-uncommitted mode can neither block nor be blocked by any other connection.

Read-uncommitted mode has no effect on the locks required to write to database tables (i.e. read-uncommitted connections must still obtain write-locks and hence database writes may still block or be blocked). Also, read-uncommitted mode has no effect on the *sqlite_master* locks required by the rules enumerated below (see section "Schema (sqlite_master) Level Locking").

```
/* Set the value of the read-uncommitted flag:
**
**   True  -&gt; Set the connection to read-uncommitted mode.
**   False -&gt; Set the connection to serialized (the default) mode.
*/
PRAGMA read_uncommitted = &lt;boolean&gt;;

/* Retrieve the current value of the read-uncommitted flag */
PRAGMA read_uncommitted;
```

2.3 Schema (sqlite_master) Level Locking

The *sqlite_master* table supports shared-cache read and write locks in the same way as all other database tables (see description above). The following special rules also apply:

- A connection must obtain a read-lock on *sqlite_master* before accessing any database tables or obtaining any other read or write locks.
- Before executing a statement that modifies the database schema (i.e. a CREATE or DROP TABLE statement), a connection must obtain a write-lock on *sqlite_master*.
- A connection may not compile an SQL statement if any other connection is holding a write-lock on the *sqlite_master* table of any attached database (including the default database, "main").

3.0 Thread Related Issues

In SQLite versions 3.3.0 through 3.4.2 when shared-cache mode is enabled, a database connection may only be used by the thread that called sqlite3_open() to create it. And a connection could only share cache with another connection in the same thread. These restrictions were dropped beginning with SQLite version 3.5.0.

4.0 Shared Cache And Virtual Tables

In older versions of SQLite, shared cache mode could not be used together with virtual tables. This restriction was removed in SQLite version 3.6.17.

5.0 Enabling Shared-Cache Mode

Shared-cache mode is enabled on a per-process basis. Using the C interface, the following API can be used to globally enable or disable shared-cache mode:

```
int sqlite3_enable_shared_cache(int);
```

Each call to sqlite3_enable_shared_cache() affects subsequent database connections created using sqlite3_open(), sqlite3_open16(), or sqlite3_open_v2(). Database connections that already exist are unaffected. Each call to sqlite3_enable_shared_cache() overrides all previous calls within the same process.

Individual database connections created using sqlite3_open_v2() can choose to participate or not participate in shared cache mode by using the SQLITE_OPEN_SHAREDCACHE or SQLITE_OPEN_PRIVATECACHE flags the third parameter. The use of either of these flags overrides the global shared cache mode setting established by sqlite3_enable_shared_cache(). No more than one of the flags should be used; if both SQLITE_OPEN_SHAREDCACHE and SQLITE_OPEN_PRIVATECACHE flags are used in the third argument to sqlite3_open_v2() then the behavior is undefined.

When URI filenames are used, the "cache" query parameter can be used to specify whether or not the database will use shared cache. Use "cache=shared" to enable shared cache and "cache=private" to disable shared cache. The ability to use URI query parameters to specify the cache sharing behavior of a database connection allows cache sharing to be controlled in ATTACH statements. For example:

```
ATTACH 'file:aux.db?cache=shared' AS aux;
```

6.0 Shared Cache And In-Memory Databases

Beginning with SQLite version 3.7.13, shared cache can be used on in-memory databases, provided that the database is created using a URI filename. For backwards compatibility, shared cache is always disable for in-memory databases if the unadorned name ":memory:" is used to open the database. Prior to version 3.7.13, shared cache was always disabled for in-memory databases regardless of the database name used, current system shared cache setting, or query parameters or flags.

Enabling shared-cache for an in-memory database allows two or more database connections in the same process to have access to the same in-memory database. An in-memory database in shared cache is automatically deleted and memory is reclaimed when the last connection to that database closes.

Using the sqlite3_unlock_notify() API

```
/* This example uses the pthreads API */
#include <pthread.h>

/*
** A pointer to an instance of this structure is passed as the user-context
** pointer when registering for an unlock-notify callback.
*/
typedef struct UnlockNotification UnlockNotification;
struct UnlockNotification {
  int fired;                         /* True after unlock event has occurred */
  pthread_cond_t cond;               /* Condition variable to wait on */
  pthread_mutex_t mutex;             /* Mutex to protect structure */
};

/*
** This function is an unlock-notify callback registered with SQLite.
*/
static void unlock_notify_cb(void **apArg, int nArg){
  int i;
  for(i=0; i<nArg; i++){
    UnlockNotification *p = (UnlockNotification *)apArg[i];
    pthread_mutex_lock(&p->mutex);
    p->fired = 1;
    pthread_cond_signal(&p->cond);
    pthread_mutex_unlock(&p->mutex);
  }
}

/*
** This function assumes that an SQLite API call (either [sqlite3_prepare_v2](c3ref/prepa
** or [sqlite3_step](c3ref/step.html)()) has just returned SQLITE_LOCKED. The argument is
** associated database connection.
**
** This function calls [sqlite3_unlock_notify](c3ref/unlock_notify.html)() to register fo
** unlock-notify callback, then blocks until that callback is delivered
** and returns SQLITE_OK. The caller should then retry the failed operation.
**
** Or, if [sqlite3_unlock_notify](c3ref/unlock_notify.html)() indicates that to block wou
** the system, then this function returns SQLITE_LOCKED immediately. In
** this case the caller should not retry the operation and should roll
** back the current transaction (if any).
*/
static int wait_for_unlock_notify(sqlite3 *db){
  int rc;
  UnlockNotification un;

  /* Initialize the UnlockNotification structure. */
  un.fired = 0;
  pthread_mutex_init(&un.mutex, 0);
  pthread_cond_init(&un.cond, 0);

  /* Register for an unlock-notify callback. */
  rc = sqlite3_unlock_notify(db, unlock_notify_cb, (void *)&un);
  assert( rc==SQLITE_LOCKED || rc==SQLITE_OK );

  /* The call to [sqlite3_unlock_notify](c3ref/unlock_notify.html)() always returns eithe
 ** or SQLITE_OK.
 **
 ** If SQLITE_LOCKED was returned, then the system is deadlocked. In this
 ** case this function needs to return SQLITE_LOCKED to the caller so
 ** that the current transaction can be rolled back. Otherwise, block
 ** until the unlock-notify callback is invoked, then return SQLITE_OK.
  */
  if( rc==SQLITE_OK ){
    pthread_mutex_lock(&un.mutex);
    if( !un.fired ){
      pthread_cond_wait(&un.cond, &un.mutex);
    }
    pthread_mutex_unlock(&un.mutex);
  }

  /* Destroy the mutex and condition variables. */
  pthread_cond_destroy(&un.cond);
  pthread_mutex_destroy(&un.mutex);

  return rc;
}

/*
** This function is a wrapper around the SQLite function [sqlite3_step](c3ref/step.html)(
** It functions in the same way as step(), except that if a required
** shared-cache lock cannot be obtained, this function may block waiting for
** the lock to become available. In this scenario the normal API step()
** function always returns SQLITE_LOCKED.
**
** If this function returns SQLITE_LOCKED, the caller should rollback
** the current transaction (if any) and try again later. Otherwise, the
** system may become deadlocked.
*/
int sqlite3_blocking_step(sqlite3_stmt *pStmt){
  int rc;
  while( SQLITE_LOCKED==(rc = sqlite3_step(pStmt)) ){
    rc = wait_for_unlock_notify(sqlite3_db_handle(pStmt));
    if( rc!=SQLITE_OK ) break;
    sqlite3_reset(pStmt);
  }
  return rc;
}

/*
** This function is a wrapper around the SQLite function [sqlite3_prepare_v2](c3ref/prepa
** It functions in the same way as prepare_v2(), except that if a required
** shared-cache lock cannot be obtained, this function may block waiting for
** the lock to become available. In this scenario the normal API prepare_v2()
** function always returns SQLITE_LOCKED.
**
** If this function returns SQLITE_LOCKED, the caller should rollback
** the current transaction (if any) and try again later. Otherwise, the
** system may become deadlocked.
*/
int sqlite3_blocking_prepare_v2(
  sqlite3 *db,              /* Database handle. */
  const char *zSql,         /* UTF-8 encoded SQL statement. */
  int nSql,                 /* Length of zSql in bytes. */
  sqlite3_stmt **ppStmt,    /* OUT: A pointer to the prepared statement */
  const char **pz           /* OUT: End of parsed string */
){
  int rc;
  while( SQLITE_LOCKED==(rc = sqlite3_prepare_v2(db, zSql, nSql, ppStmt, pz)) ){
    rc = wait_for_unlock_notify(db);
    if( rc!=SQLITE_OK ) break;
  }
  return rc;
}
```

When two or more connections access the same database in shared-cache mode, read and write (shared and exclusive) locks on individual tables are used to ensure that concurrently executing transactions are kept isolated. Before writing to a table, a write (exclusive) lock must be obtained on that table. Before reading, a read (shared) lock must be obtained. A connection releases all held table locks when it concludes its transaction. If a connection cannot obtain a required lock, then the call to sqlite3_step() returns SQLITE_LOCKED.

Although it is less common, a call to sqlite3_prepare() or sqlite3_prepare_v2() may also return SQLITE_LOCKED if it cannot obtain a read-lock on the sqlite_master table of each attached database. These APIs need to read the schema data contained in the sqlite_master table in order to compile SQL statements to sqlite3_stmt* objects.

This article presents a technique using the SQLite sqlite3_unlock_notify() interface such that calls to sqlite3_step() and sqlite3_prepare_v2() block until the required locks are available instead of returning SQLITE_LOCKED immediately. If the sqlite3_blocking_step() or sqlite3_blocking_prepare_v2() functions presented to the left return SQLITE_LOCKED, this indicates that to block would deadlock the system.

The sqlite3_unlock_notify() API, which is only available if the library is compiled with the pre-processor symbol SQLITE_ENABLE_UNLOCK_NOTIFY defined, is documented here. This article is not a substitute for reading the full API documentation!

The sqlite3_unlock_notify() interface is designed for use in systems that have a separate thread assigned to each database connection. There is nothing in the implementation that prevents a single thread from running multiple database connections. However, the sqlite3_unlock_notify() interface only works on a single connection at a time, so the lock resolution logic presented here will only work for a single database connection per thread.

The sqlite3_unlock_notify() API

After a call to sqlite3_step() or sqlite3_prepare_v2() returns SQLITE_LOCKED, the sqlite3_unlock_notify() API may be invoked to register for an unlock-notify callback. The unlock-notify callback is invoked by SQLite after the database connection holding the table-lock that prevented the call to sqlite3_step() or sqlite3_prepare_v2() from succeeding has finished its transaction and released all locks. For example, if a call to sqlite3_step() is an attempt to read from table X, and some other connection Y is holding a write-lock on table X, then sqlite3_step() will return SQLITE_LOCKED. If sqlite3_unlock_notify() is then called, the unlock-notify callback will be invoked after connection Y's transaction is concluded. The connection that the unlock-notify callback is waiting on, in this case connection Y, is known as the "blocking connection".

If a call to sqlite3_step() that attempts to write to a database table returns SQLITE_LOCKED, then more than one other connection may be holding a read-lock on the database table in question. In this case SQLite simply selects one of those other connections arbitrarily and issues the unlock-notify callback when that connection's transaction is finished. Whether the call to sqlite3_step() was blocked by one or many connections, when the corresponding unlock-notify callback is issued it is not guaranteed that the required lock is available, only that it may be.

When the unlock-notify callback is issued, it is issued from within a call to sqlite3_step() (or sqlite3_close()) associated with the blocking connection. It is illegal to invoke any sqlite3_XXX() API functions from within an unlock-notify callback. The expected use is that the unlock-notify callback will signal some other waiting thread or schedule some action to take place later.

The algorithm used by the sqlite3_blocking_step() function is as follows:

1. Call sqlite3_step() on the supplied statement handle. If the call returns anything other than SQLITE_LOCKED, then return this value to the caller. Otherwise, continue.

2. Invoke sqlite3_unlock_notify() on the database connection handle associated with the supplied statement handle to register for an unlock-notify callback. If the call to unlock_notify() returns SQLITE_LOCKED, then return this value to the caller.

3. Block until the unlock-notify callback is invoked by another thread.

4. Call sqlite3_reset() on the statement handle. Since an SQLITE_LOCKED error may only occur on the first call to sqlite3_step() (it is not possible for one call to sqlite3_step() to return SQLITE_ROW and then the next SQLITE_LOCKED), the statement handle may be reset at this point without affecting the results of the query from the point of view of the caller. If sqlite3_reset() were not called at this point, the next call to sqlite3_step() would return SQLITE_MISUSE.

5. Return to step 1.

The algorithm used by the sqlite3_blocking_prepare_v2() function is similar, except that step 4 (resetting the statement handle) is omitted.

Writer Starvation

Multiple connections may hold a read-lock simultaneously. If many threads are acquiring overlapping read-locks, it might be the case that at least one thread is always holding a read lock. Then a table waiting for a write-lock will wait forever. This scenario is called "writer starvation."

SQLite helps applications avoid writer starvation. After any attempt to obtain a write-lock on a table fails (because one or more other connections are holding read-locks), all attempts to open new transactions on the shared-cache fail until one of the following is true:

- The current writer concludes its transaction, OR
- The number of open read-transactions on the shared-cache drops to zero.

Failed attempts to open new read-transactions return SQLITE_LOCKED to the caller. If the caller then calls sqlite3_unlock_notify() to register for an unlock-notify callback, the blocking connection is the connection that currently has an open write-transaction on the shared-cache. This prevents writer-starvation since if no new read-transactions may be opened and assuming all existing read-transactions are eventually concluded, the writer will eventually have an opportunity to obtain the required write-lock.

The pthreads API

By the time sqlite3_unlock_notify() is invoked by wait_for_unlock_notify(), it is possible that the blocking connection that prevented the sqlite3_step() or sqlite3_prepare_v2() call from succeeding has already finished its transaction. In this case, the unlock-notify callback is invoked immediately, before sqlite3_unlock_notify() returns. Or, it is possible that the unlock-notify callback is invoked by a second thread after sqlite3_unlock_notify() is called but before the thread starts waiting to be asynchronously signaled.

Exactly how such a potential race-condition is handled depends on the threads and synchronization primitives interface used by the application. This example uses pthreads, the interface provided by modern UNIX-like systems, including Linux.

The pthreads interface provides the pthread_cond_wait() function. This function allows the caller to simultaneously release a mutex and start waiting for an asynchronous signal. Using this function, a "fired" flag and a mutex, the race-condition described above may be eliminated as follows:

When the unlock-notify callback is invoked, which may be before the thread that called sqlite3_unlock_notify() begins waiting for the asynchronous signal, it does the following:

1. Obtains the mutex.
2. Sets the "fired" flag to true.
3. Attempts to signal a waiting thread.
4. Releases the mutex.

When the wait_for_unlock_notify() thread is ready to begin waiting for the unlock-notify callback to arrive, it:

1. Obtains the mutex.
2. Checks if the "fired" flag has been set. If so, the unlock-notify callback has already been invoked. Release the mutex and continue.
3. aAtomically releases the mutex and begins waiting for the asynchronous signal. When the signal arrives, continue.

This way, it doesn't matter if the unlock-notify callback has already been invoked, or is being invoked, when the wait_for_unlock_notify() thread begins blocking.

Possible Enhancements

The code in this article could be improved in at least two ways:

- It could manage thread priorities.
- It could handle a special case of SQLITE_LOCKED that can occur when dropping a table or index.

Even though the sqlite3_unlock_notify() function only allows the caller to specify a single user-context pointer, an unlock-notify callback is passed an array of such context pointers. This is because if when a blocking connection concludes its transaction, if there is more than one unlock-notify registered to call the same C function, the context-pointers are marshaled into an array and a single callback issued. If each thread were assigned a priority, then instead of just signaling the threads in arbitrary order as this implementation does, higher priority threads could be signaled before lower priority threads.

If a "DROP TABLE" or "DROP INDEX" SQL command is executed, and the same database connection currently has one or more actively executing SELECT statements, then SQLITE_LOCKED is returned. If sqlite3_unlock_notify() is called in this case, then the specified callback will be invoked immediately. Re-attempting the "DROP TABLE" or "DROP INDEX" statement will return another SQLITE_LOCKED error. In the implementation of sqlite3_blocking_step() shown to the left, this could cause an infinite loop.

The caller could distinguish between this special "DROP TABLE|INDEX" case and other cases by using extended error codes. When it is appropriate to call sqlite3_unlock_notify(), the extended error code is SQLITE_LOCKED_SHAREDCACHE. Otherwise, in the "DROP TABLE|INDEX" case, it is just plain SQLITE_LOCKED. Another solution might be to limit the number of times that any single query could be reattempted (to say 100). Although this might be less efficient than one might wish, the situation in question is not likely to occur often.

URI Filenames

1.0 URI Filenames In SQLite

Beginning with version 3.7.7, the SQLite database file argument to the sqlite3_open(), sqlite3_open16(), and sqlite3_open_v2() interfaces and to the ATTACH command can be specified either as an ordinary filename or as a Uniform Resource Identifier or URI. The advantage of using a URI filename is that query parameters on the URI can be used to control details of the newly created database connection. For example, an alternative VFS can be specified using a "vfs=" query parameter. Or the database can be opened read-only by using "mode=ro" as a query parameter.

2.0 Backwards Compatibility

In order to maintain full backwards compatibility for legacy applications, the URI filename capability is disabled by default. URI filenames can be enabled or disabled using the SQLITE_USE_URI=1 or SQLITE_USE_URI=0 compile-time options. The compile-time setting for URI filenames can be changed at start-time using the sqlite3_config(SQLITE_CONFIG_URI,1) or sqlite3_config(SQLITE_CONFIG_URI,0) configuration calls. Regardless of the compile-time or start-time settings, URI filenames can be enabled for individual database connections by including the SQLITE_OPEN_URI bit in the set of bits passed as the F parameter to sqlite3_open_v2(N,P,F,V).

If URI filenames are recognized when the database connection is originally opened, then URI filenames will also be recognized on ATTACH statements. Similarly, if URI filenames are not recognized when the database connection is first opened, they will not be recognized by ATTACH.

Since SQLite always interprets any filename that does not begin with "`file:`" as an ordinary filename regardless of the URI setting, and because it is very unusual to have an actual file begin with "`file:`", it is safe for most applications to enable URI processing even if URI filenames are not currently being used.

3.0 URI Format

According to RFC 3986, a URI consists of a scheme, an authority, a path, a query string, and a fragment. The scheme is always required. One of either the authority or the path is also always required. The query string and fragment are optional.

SQLite uses the "`file:`" URI syntax to identify database files. SQLite strives to interpret file: URIs in exactly the same way as popular web-browsers such as Firefox, Chrome, Safari, Internet Explorer, and Opera, and command-line programs such as Windows "start" and the Mac OS-X "open" command. A succinct summary of the URI parsing rules follows:

- The scheme of the URI must be "`file:`". Any other scheme results in the input being treated as an ordinary filename.
- The authority may be omitted, may be blank, or may be "`localhost`". Any other authority results in an error. Exception: If SQLite is compiled with SQLITE_ALLOW_URI_AUTHORITY then any authority value other than "localhost" is passed through to the underlying operating system as a UNC filename.
- The path is optional if the authority is present. If the authority is omitted then the path is required.
- The query string is optional. If the query string is present, then all query parameters are passed through into the xOpen method of the underlying VFS.
- The fragment is optional. If present, it is ignored.

Zero or more escape sequences of the form "***%HH***" (where ***H*** represents any hexadecimal digit) can occur in the path, query string, or fragment.

A filename that is not a well-formed URI is interpreted as an ordinary filename.

URIs are processed as UTF8 text. The filename argument sqlite3_open16() is converted from UTF16 native byte order into UTF8 prior to processing.

3.1 The URI Path

The path component of the URI specifies the disk file that is the SQLite database to be opened. If the path component is omitted, then the database is stored in a temporary file that will be automatically deleted when the database connection closes. If the authority section is present, then the path is always an absolute pathname. If the authority section is omitted, then the path is an absolute pathname if it begins with the "/" character (ASCII code 0x2f) and is a relative pathname otherwise. On windows, if the absolute path begins with "***/X:/***" where ***X*** is any single ASCII alphabetic character ("a" through "z" or "A" through "Z") then the "***X:***" is understood to be the drive letter of the volume containing the file, not the toplevel directory.

An ordinary filename can usually be converted into an equivalent URI by the steps shown below. The one exception is that a relative windows pathname with a drive letter cannot be converted directly into a URI; it must be changed into an absolute pathname first.

1. Convert all "`?`" characters into "`%3f`".
2. Convert all "`#`" characters into "`%23`".

3. On windows only, convert all "`\`" characters into "`/`".
4. Convert all sequences of two or more "`/`" characters into a single "`/`" character.
5. On windows only, if the filename begins with a drive letter, prepend a single "`/`" character.
6. Prepend the "`file:`" scheme.

3.2 Query String

A URI filename can optionally be followed by a query string. The query string consists of text following the first "`?`" character but excluding the optional fragment that begins with "`#`". The query string is divided into key/value pairs. We usually refer to these key/value pairs as "query parameters". Key/value pairs are separated by a single "`&`" character. The key comes first and is separated from the value by a single "`=`" character. Both key and value may contain **%HH** escape sequences.

The text of query parameters is appended to the filename argument of the xOpen method of the VFS. Any %HH escape sequences in the query parameters are resolved prior to being appended to the xOpen filename. A single zero-byte separates the xOpen filename argument from the key of the first query parameters, each key and value, and each subsequent key from the prior value. The list of query parameters appended to the xOpen filename is terminated by a single zero-length key. Note that the value of a query parameter can be an empty string.

3.3 Recognized Query Parameters

Some query parameters are interpreted by the SQLite core and used to modify the characteristics of the new connection. All query parameters are always passed through into the xOpen method of the VFS even if they are previously read and interpreted by the SQLite core.

The following query parameters are recognized by SQLite as of version 3.8.0. Other query parameters might be added to this set in future releases.

vfs=*NAME*

The vfs query parameter causes the database connection to be opened using the VFS called *NAME*. The open attempt fails if *NAME* is not the name of a VFS that is built into SQLite or that has been previously registered using sqlite3_vfs_register().

mode=ro mode=rw mode=rwc mode=memory

The mode query parameter determines if the new database is opened read-only, read-write, read-write and created if it does not exist, or that the database is a pure in-memory database that never interacts with disk, respectively.

cache=shared cache=private

The cache query parameter determines if the new database is opened using shared cache mode or with a private cache.

psow=0 psow=1

The psow query parameter overrides the powersafe overwrite property of the database file being opened. The psow query parameter works with the default windows and unix VFSes but might be a no-op for other proprietary or non-standard VFSes.

nolock=1

The nolock query parameter is a boolean that disables all calls to the xLock, xUnlock, and xCheckReservedLock methods of the VFS when true. The nolock query parameter might be used, for example, when trying to access a file on a filesystem that does not support file locking. Caution: If two or more database connections try to interact with the same SQLite database and one or more of those connections has enabled "nolock", then database corruption can result. The "nolock" query parameter should only be used if the application can guarantee that writes to the database are serialized.

immutable=1

The immutable query parameter is a boolean that signals to SQLite that the underlying database file is held on read-only media and cannot be modified, even by another process with elevated privileges. SQLite always opens immutable database files read-only and it skips all file locking and change detection on immutable database files. If these query parameter (or the SQLITE_IOCAP_IMMUTABLE bit in xDeviceCharacteristics) asserts that a database file is immutable and that file changes anyhow, then SQLite might return incorrect query results and/or SQLITE_CORRUPT errors.

4.0 See Also

- URI filenames in sqlite3_open()
- URI filename examples

The WITHOUT ROWID Optimization

1.0 Introduction

By default, every row in SQLite has a special column, usually called the "rowid", that uniquely identifies that row within the table. However if the phrase "WITHOUT ROWID" is added to the end of a CREATE TABLE statement, then the special "rowid" column is omitted. There are sometimes space and performance advantages to omitting the rowid.

1.1 Syntax

To create a WITHOUT ROWID table, simply add the keywords "WITHOUT ROWID" to the end of the CREATE TABLE statement. For example:

```
CREATE TABLE IF NOT EXISTS wordcount(
  word TEXT PRIMARY KEY,
  cnt INTEGER
) WITHOUT ROWID;
```

As with all SQL syntax, the case of the keywords does not matter. One can write "WITHOUT rowid" or "without rowid" or "WiThOuT rOwId" and it will mean the same thing.

Every WITHOUT ROWID table must have a PRIMARY KEY. An error is raised if a CREATE TABLE statement with the WITHOUT ROWID clause lacks a PRIMARY KEY.

In most contexts, the special "rowid" column of normal tables can also be called "oid" or "*rowid*". However, only "rowid" works as the keyword in the CREATE TABLE statement.

1.2 Compatibility

SQLite version 3.8.2 or later is necessary in order to use a WITHOUT ROWID table. An attempt to open a database that contains one or more WITHOUT ROWID tables using an earlier version of SQLite will result in a "malformed database schema" error.

1.3 Quirks

WITHOUT ROWID is found only in SQLite and is not compatible with any other SQL database engine, as far as we know. In an elegant system, all tables would behave as WITHOUT ROWID tables even without the WITHOUT ROWID keyword. However, when SQLite was first designed, it used only integer rowids for row keys to simplify the implementation. This approach worked well for many years. But as the demands on SQLite grew, the need for tables in which the PRIMARY KEY really did correspond to the underlying row key grew more acute. The WITHOUT ROWID concept was added in order to meet that need without breaking backwards compatibility with the billions of SQLite databases already in use at the time (circa 2013).

2.0 Differences From Ordinary Rowid Tables

The WITHOUT ROWID syntax is an optimization. It provides no new capabilities. Anything that can be done using a WITHOUT ROWID table can also be done in exactly the same way, and exactly the same syntax, using an ordinary rowid table. The only advantage of a WITHOUT ROWID table is that it can sometimes use less disk space and/or perform a little faster than an ordinary rowid table.

For the most part, ordinary rowid tables and WITHOUT ROWID tables are interchangeable. But there are some additional restrictions on WITHOUT ROWID tables that do not apply to ordinary rowid tables:

1. **Every WITHOUT ROWID table must have a PRIMARY KEY.** An attempt to create a WITHOUT ROWID table without a PRIMARY KEY results in an error.

2. **The special behaviors associated "INTEGER PRIMARY KEY" do not apply on WITHOUT ROWID tables.** In an ordinary table, "INTEGER PRIMARY KEY" means that the column is an alias for the rowid. But since there is no rowid in a WITHOUT ROWID table, that special meaning no longer applies. An "INTEGER PRIMARY KEY" column in a WITHOUT ROWID table works like an "INT PRIMARY KEY" column in an ordinary table: It is a PRIMARY KEY that has integer affinity.

3. **AUTOINCREMENT does not work on WITHOUT ROWID tables.** The AUTOINCREMENT mechanism assumes the presence of a rowid and so it does not work on a WITHOUT ROWID table. An error is raised if the "AUTOINCREMENT" keyword is used in the CREATE TABLE statement for a WITHOUT ROWID table.

4. **NOT NULL is enforced on every column of the PRIMARY KEY in a WITHOUT ROWID table.** This is in accordance with the SQL standard. Each column of a PRIMARY KEY is supposed to be individually NOT NULL. However, NOT NULL was not enforced on PRIMARY KEY columns by early versions of SQLite due to a bug. By the time that this bug was discovered, so many SQLite databases were already in circulation that the decision was made not to fix this bug for fear of breaking compatibility. So, ordinary rowid tables in SQLite violate the SQL standard and allow NULL values in PRIMARY KEY fields. But WITHOUT ROWID tables do follow the standard and will throw an error on any attempt to insert a NULL into a PRIMARY KEY column.

5. **The sqlite3_last_insert_rowid() function does not work for WITHOUT ROWID tables.** Inserts into a WITHOUT ROWID do not change the value returned by the sqlite3_last_insert_rowid() function. The last_insert_rowid() SQL function is also unaffected since it is just a wrapper around sqlite3_last_insert_rowid().

6. **The incremental blob I/O mechanism does not work for WITHOUT ROWID tables.** Incremental BLOB I/O uses the rowid to create an sqlite3_blob object for doing the direct I/O. However, WITHOUT ROWID tables do not have a rowid, and so there is no way to create an sqlite3_blob object for a WITHOUT ROWID table.

7. **The sqlite3_update_hook() interface does not fire callbacks for changes to a WITHOUT ROWID table.** Part of the callback from sqlite3_update_hook() is the rowid of the table row that has changed. However, WITHOUT ROWID tables do not have a rowid. Hence, the update hook is not invoked when a WITHOUT ROWID table changes.

   Note that since the session extension uses the update hook, that means that the session extension will not work correctly on a database that includes WITHOUT ROWID tables.

3.0 Benefits Of WITHOUT ROWID Tables

A WITHOUT ROWID table is an optimization that can reduce storage and processing requirements.

In an ordinary SQLite table, the PRIMARY KEY is really just a UNIQUE index. The key used to look up records on disk is the rowid. The special "INTEGER PRIMARY KEY" column type in ordinary SQLite tables causes the column to be an alias for the rowid, and so an INTEGER PRIMARY KEY is a true PRIMARY KEY. But any other kind of PRIMARY KEYs, including "INT PRIMARY KEY" are just unique indexes in an ordinary rowid table.

Consider a table (shown below) intended to store a vocabulary of words together with a count of the number of occurrences of each word in some text corpus:

```
CREATE TABLE IF NOT EXISTS wordcount(
  word TEXT PRIMARY KEY,
  cnt INTEGER
);
```

As an ordinary SQLite table, "wordcount" is implemented as two separate B-Trees. The main table uses the hidden rowid value as the key and stores the "word" and "cnt" columns as data. The "TEXT PRIMARY KEY" phrase of the CREATE TABLE statement causes the creation of an unique index on the "word" column. This index is a separate B-Tree that uses "word" and the "rowid" as the key and stores no data at all. Note that the complete text of every "word" is stored twice: once in the main table and again in the index.

Consider querying this table to find the number of occurrences of the word "xyzzy".:

```
SELECT cnt FROM wordcount WHERE word='xyzzy';
```

This query first has to search the index B-Tree looking for any entry that contains the matching value for "word". When an entry is found in the index, the rowid is extracted and used to search the main table. Then the "cnt" value is read out of the main table and returned. Hence, two separate binary searches are required to fulfill the request.

A WITHOUT ROWID table uses a different data design for the equivalent table.

```
CREATE TABLE IF NOT EXISTS wordcount(
  word TEXT PRIMARY KEY,
  cnt INTEGER
) WITHOUT ROWID;
```

In this latter table, there is only a single B-Tree which uses the "word" column as its key and the "cnt" column as its data. (Technicality: the low-level implementation actually stores both "word" and "cnt" in the "key" area of the B-Tree. But unless you are looking at the low-level byte encoding of the database file, that fact is unimportant.) Because there is only a single B-Tree, the text of the "word" column is only stored once in the database. Furthermore, querying the "cnt" value for a specific "word" only involves a single binary search into the main B-Tree, since the "cnt" value can be retrieved directly from the record found by that first search and without the need to do a second binary search on the rowid.

Thus, in some cases, a WITHOUT ROWID table can use about half the amount of disk space and can operate nearly twice as fast. Of course, in a real-world schema, there will typically be secondary indices and/or UNIQUE constraints, and the situation is more complicated. But even then, there can often be space and performance advantages to using WITHOUT ROWID on tables that have non-integer or composite PRIMARY KEYs.

4.0 When To Use WITHOUT ROWID

The WITHOUT ROWID optimization is likely to be helpful for tables that have non-integer or composite (multi-column) PRIMARY KEYs and that do not store large strings or BLOBs.

WITHOUT ROWID tables will work correctly (that is to say, they provide the correct answer) for tables with a single INTEGER PRIMARY KEY. However, ordinary rowid tables will run faster in that case. Hence, it is good design to avoid creating WITHOUT ROWID tables with single-column PRIMARY KEYs of type INTEGER.

WITHOUT ROWID tables work best when individual rows are not too large. A good rule-of-thumb is that the average size of a single row in a WITHOUT ROWID table should be less than about 1/20th the size of a database page. That means that rows should not contain more than about 50 bytes each for a 1KiB page size or about 200 bytes each for 4KiB page size. WITHOUT ROWID tables will work (in the sense that they get the correct answer) for arbitrarily large rows - up to 2GB in size - but traditional rowid tables tend to work faster for large row sizes. This is because rowid tables are implemented as B*-Trees where all content is stored in the leaves of the tree, whereas WITHOUT ROWID tables are implemented using ordinary B-Trees with content stored on both leaves and intermediate nodes. Storing content in intermediate nodes mean that each intermediate node entry takes up more space on the page and thus reduces the fan-out, increasing the search cost.

The "sqlite3_analyzer.exe" utility program, available as source code in the SQLite source tree or as a precompiled binary on the SQLite Download page, can be used to measure the average sizes of table rows in an existing SQLite database.

Note that except for a few corner-case differences detailed above, WITHOUT ROWID tables and rowid tables work the same. They both generate the same answers given the same SQL statements. So it is a simple matter to run experiments on an application, late in the development cycle, to test whether or not the use of WITHOUT ROWID tables will be helpful. A good strategy is to simply not worry about WITHOUT ROWID until near the end of product development, then go back and run tests to see if adding WITHOUT ROWID to tables with non-integer PRIMARY KEYs helps or hurts performance, and retaining the WITHOUT ROWID only in those cases where it helps.

Write-Ahead Logging

The default method by which SQLite implements atomic commit and rollback is a rollback journal. Beginning with version 3.7.0, a new "Write-Ahead Log" option (hereafter referred to as "WAL") is available.

There are advantages and disadvantages to using WAL instead of a rollback journal. Advantages include:

1. WAL is significantly faster in most scenarios.
2. WAL provides more concurrency as readers do not block writers and a writer does not block readers. Reading and writing can proceed concurrently.
3. Disk I/O operations tends to be more sequential using WAL.
4. WAL uses many fewer fsync() operations and is thus less vulnerable to problems on systems where the fsync() system call is broken.

But there are also disadvantages:

1. WAL normally requires that the VFS support shared-memory primitives. (Exception: WAL without shared memory) The built-in unix and windows VFSes support this but third-party extension VFSes for custom operating systems might not.
2. All processes using a database must be on the same host computer; WAL does not work over a network filesystem.
3. Transactions that involve changes against multiple ATTACHed databases are atomic for each individual database, but are not atomic across all databases as a set.
4. It is not possible to change the database page size after entering WAL mode, either on an empty database or by using VACUUM or by restoring from a backup using the backup API. You must be in a rollback journal mode to change the page size.
5. It is not possible to open read-only WAL databases. The opening process must have write privileges for "`-shm`" wal-index shared memory file associated with the database, if that file exists, or else write access on the directory containing the database file if the "`-shm`" file does not exist.
6. WAL might be very slightly slower (perhaps 1% or 2% slower) than the traditional rollback-journal approach in applications that do mostly reads and seldom write.
7. There is an additional quasi-persistent "`-wal`" file and "`-shm`" shared memory file associated with each database, which can make SQLite less appealing for use as an application file-format.
8. There is the extra operation of checkpointing which, though automatic by default, is still something that application developers need to be mindful of.
9. ~~WAL works best with smaller transactions. WAL does not work well for very large~~

~~transactions. For transactions larger than about 100 megabytes, traditional rollback journal modes will likely be faster. For transactions in excess of a gigabyte, WAL mode may fail with an I/O or disk-full error. It is recommended that one of the rollback journal modes be used for transactions larger than a few dozen megabytes.~~ Beginning with version 3.11.0, WAL mode works as efficiently with large transactions as does rollback mode.

How WAL Works

The traditional rollback journal works by writing a copy of the original unchanged database content into a separate rollback journal file and then writing changes directly into the database file. In the event of a crash or ROLLBACK, the original content contained in the rollback journal is played back into the database file to revert the database file to its original state. The COMMIT occurs when the rollback journal is deleted.

The WAL approach inverts this. The original content is preserved in the database file and the changes are appended into a separate WAL file. A COMMIT occurs when a special record indicating a commit is appended to the WAL. Thus a COMMIT can happen without ever writing to the original database, which allows readers to continue operating from the original unaltered database while changes are simultaneously being committed into the WAL. Multiple transactions can be appended to the end of a single WAL file.

Checkpointing

Of course, one wants to eventually transfer all the transactions that are appended in the WAL file back into the original database. Moving the WAL file transactions back into the database is called a "*checkpoint*".

Another way to think about the difference between rollback and write-ahead log is that in the rollback-journal approach, there are two primitive operations, reading and writing, whereas with a write-ahead log there are now three primitive operations: reading, writing, and checkpointing.

By default, SQLite does a checkpoint automatically when the WAL file reaches a threshold size of 1000 pages. (The SQLITE_DEFAULT_WAL_AUTOCHECKPOINT compile-time option can be used to specify a different default.) Applications using WAL do not have to do anything in order to for these checkpoints to occur. But if they want to, applications can adjust the automatic checkpoint threshold. Or they can turn off the automatic checkpoints and run checkpoints during idle moments or in a separate thread or process.

Concurrency

When a read operation begins on a WAL-mode database, it first remembers the location of the last valid commit record in the WAL. Call this point the "end mark". Because the WAL can be growing and adding new commit records while various readers connect to the database, each reader can potentially have its own end mark. But for any particular reader, the end mark is unchanged for the duration of the transaction, thus ensuring that a single read transaction only sees the database content as it existed at a single point in time.

When a reader needs a page of content, it first checks the WAL to see if that page appears there, and if so it pulls in the last copy of the page that occurs in the WAL prior to the reader's end mark. If no copy of the page exists in the WAL prior to the reader's end mark, then the page is read from the original database file. Readers can exist in separate processes, so to avoid forcing every reader to scan the entire WAL looking for pages (the WAL file can grow to multiple megabytes, depending on how often checkpoints are run), a data structure called the "wal-index" is maintained in shared memory which helps readers locate pages in the WAL quickly and with a minimum of I/O. The wal-index greatly improves the performance of readers, but the use of shared memory means that all readers must exist on the same machine. This is why the write-ahead log implementation will not work on a network filesystem.

Writers merely append new content to the end of the WAL file. Because writers do nothing that would interfere with the actions of readers, writers and readers can run at the same time. However, since there is only one WAL file, there can only be one writer at a time.

A checkpoint operation takes content from the WAL file and transfers it back into the original database file. A checkpoint can run concurrently with readers, however the checkpoint must stop when it reaches a page in the WAL that is past the read mark of any current reader. The checkpoint has to stop at that point because otherwise it might overwrite part of the database file that the reader is actively using. The checkpoint remembers (in the wal-index) how far it got and will resume transferring content from the WAL to the database from where it left off on the next invocation.

Thus a long-running read transaction can prevent a checkpointer from making progress. But presumably every read transactions will eventually end and the checkpointer will be able to continue.

Whenever a write operation occurs, the writer checks how much progress the checkpointer has made, and if the entire WAL has been transferred into the database and synced and if no readers are making use of the WAL, then the writer will rewind the WAL back to the beginning and start putting new transactions at the beginning of the WAL. This mechanism prevents a WAL file from growing without bound.

Performance Considerations

Write transactions are very fast since they only involve writing the content once (versus twice for rollback-journal transactions) and because the writes are all sequential. Further, syncing the content to the disk is not required, as long as the application is willing to sacrifice durability following a power loss or hard reboot. (Writers sync the WAL on every transaction commit if PRAGMA synchronous is set to FULL but omit this sync if PRAGMA synchronous is set to NORMAL.)

On the other hand, read performance deteriorates as the WAL file grows in size since each reader must check the WAL file for the content and the time needed to check the WAL file is proportional to the size of the WAL file. The wal-index helps find content in the WAL file much faster, but performance still falls off with increasing WAL file size. Hence, to maintain good read performance it is important to keep the WAL file size down by running checkpoints at regular intervals.

Checkpointing does require sync operations in order to avoid the possibility of database corruption following a power loss or hard reboot. The WAL must be synced to persistent storage prior to moving content from the WAL into the database and the database file must by synced prior to resetting the WAL. Checkpoint also requires more seeking. The checkpointer makes an effort to do as many sequential page writes to the database as it can (the pages are transferred from WAL to database in ascending order) but even then there will typically be many seek operations interspersed among the page writes. These factors combine to make checkpoints slower than write transactions.

The default strategy is to allow successive write transactions to grow the WAL until the WAL becomes about 1000 pages in size, then to run a checkpoint operation for each subsequent COMMIT until the WAL is reset to be smaller than 1000 pages. By default, the checkpoint will be run automatically by the same thread that does the COMMIT that pushes the WAL over its size limit. This has the effect of causing most COMMIT operations to be very fast but an occasional COMMIT (those that trigger a checkpoint) to be much slower. If that effect is undesirable, then the application can disable automatic checkpointing and run the periodic checkpoints in a separate thread, or separate process. (Links to commands and interfaces to accomplish this are shown below.)

Note that with PRAGMA synchronous set to NORMAL, the checkpoint is the only operation to issue an I/O barrier or sync operation (fsync() on unix or FlushFileBuffers() on windows). If an application therefore runs checkpoint in a separate thread or process, the main thread or process that is doing database queries and updates will never block on a sync operation. This helps to prevent "latch-up" in applications running on a busy disk drive. The downside to this configuration is that transactions are no longer durable and might rollback following a power failure or hard reset.

Notice too that there is a tradeoff between average read performance and average write performance. To maximize the read performance, one wants to keep the WAL as small as possible and hence run checkpoints frequently, perhaps as often as every COMMIT. To maximize write performance, one wants to amortize the cost of each checkpoint over as many writes as possible, meaning that one wants to run checkpoints infrequently and let the WAL grow as large as possible before each checkpoint. The decision of how often to run checkpoints may therefore vary from one application to another depending on the relative read and write performance requirements of the application. The default strategy is to run a checkpoint once the WAL reaches 1000 pages and this strategy seems to work well in test applications on workstations, but other strategies might work better on different platforms or for different workloads.

Activating And Configuring WAL Mode

An SQLite database connection defaults to journal_mode=DELETE. To convert to WAL mode, use the following pragma:

```
PRAGMA journal_mode=WAL;
```

The journal_mode pragma returns a string which is the new journal mode. On success, the pragma will return the string "`wal`". If the conversion to WAL could not be completed (for example, if the VFS does not support the necessary shared-memory primitives) then the journaling mode will be unchanged and the string returned from the primitive will be the prior journaling mode (for example "`delete`").

Automatic Checkpoint

By default, SQLite will automatically checkpoint whenever a COMMIT occurs that causes the WAL file to be 1000 pages or more in size, or when the last database connection on a database file closes. The default configuration is intended to work well for most applications. But programs that want more control can force a checkpoint using the wal_checkpoint pragma or by calling the sqlite3_wal_checkpoint() C interface. The automatic checkpoint threshold can be changed or automatic checkpointing can be completely disabled using the wal_autocheckpoint pragma or by calling the sqlite3_wal_autocheckpoint() C interface. A program can also use sqlite3_wal_hook() to register a callback to be invoked whenever any transaction commits to the WAL. This callback can then invoke sqlite3_wal_checkpoint() or sqlite3_wal_checkpoint_v2() based on whatever criteria it thinks is appropriate. (The automatic checkpoint mechanism is implemented as a simple wrapper around sqlite3_wal_hook().)

Application-Initiated Checkpoints

An application can initiate a checkpoint using any writable database connection on the database simply by invoking sqlite3_wal_checkpoint() or sqlite3_wal_checkpoint_v2(). There are three subtypes of checkpoints that vary in their aggressiveness: PASSIVE, FULL, and RESTART. The default checkpoint style is PASSIVE, which does as much work as it can without interfering with other database connections, and which might not run to completion if there are concurrent readers or writers. All checkpoints initiated by sqlite3_wal_checkpoint() and by the automatic checkpoint mechanism are PASSIVE. FULL and RESTART checkpoints try harder to run the checkpoint to completion and can only be initiated by a call to sqlite3_wal_checkpoint_v2(). See the sqlite3_wal_checkpoint_v2() documentation for additional information on FULL and RESET checkpoints.

Persistence of WAL mode

Unlike the other journaling modes, PRAGMA journal_mode=WAL is persistent. If a process sets WAL mode, then closes and reopens the database, the database will come back in WAL mode. In contrast, if a process sets (for example) PRAGMA journal_mode=TRUNCATE and then closes and reopens the database will come back up in the default rollback mode of DELETE rather than the previous TRUNCATE setting.

The persistence of WAL mode means that applications can be converted to using SQLite in WAL mode without making any changes to the application itself. One has merely to run "`PRAGMA journal_mode=WAL;`" on the database file(s) using the command-line shell or other utility, then restart the application.

The WAL journal mode will be set on all connections to the same database file if it is set on any one connection.

Read-Only Databases

No SQLite database (regardless of whether or not it is WAL mode) is readable if it is located on read-only media and it requires recovery. So, for example, if an application crashes and leaves an SQLite database with a hot journal, that database cannot be opened unless the opening process has write privilege on the database file, the directory containing the database file, and the hot journal. This is because the incomplete transaction left over from the crash must be rolled back prior to reading the database and that rollback cannot occur without write permission on all files and the directory containing them.

A database in WAL mode cannot generally be opened from read-only media because even ordinary reads in WAL mode require recovery-like operations.

An efficient implementation of the WAL read algorithm requires that there exist a hash table in shared memory over the content of the WAL file. This hash table is called the wal-index. The wal-index is in shared memory, and so technically it does not have to have a name in the host computer filesystem. Custom VFS implementations are free to implement shared memory in any way they see fit, but the default unix and windows drivers that come built-in with SQLite implement shared memory using mmapped files named using the suffix "`-shm`" and located in the same directory as the database file. The wal-index must be rebuilt upon first access, even by readers, and so in order to open the WAL database, write access is required on the "`-shm`" shared memory file if the file exists, or else write access is required on the directory containing the database so that the wal-index can be created if it does not already exist. This does not preclude custom VFS implementations that implement shared memory differently from being able to access read-only WAL databases, but it does prevent the default unix and windows backends from accessing WAL databases on read-only media.

Hence, SQLite databases should always be converted to PRAGMA journal_mode=DELETE prior to being transferred to read-only media.

Also, if multiple processes are to access a WAL mode database, then all processes should run under user or group IDs that give them write access to the database files, the WAL file, the shared memory `-shm` file, and the containing directory.

Avoiding Excessively Large WAL Files

In normal cases, new content is appended to the WAL file until the WAL file accumulates about 1000 pages (and is thus about 4MB in size) at which point a checkpoint is automatically run and the WAL file is recycled. The checkpoint does not normally truncate the WAL file (unless the journal_size_limit pragma is set). Instead, it merely causes SQLite to start overwriting the WAL file from the beginning. This is done because it is normally faster to overwrite an existing file than to append. When the last connection to a database closes, that connection does one last checkpoint and then deletes the WAL and its associated shared-memory file, to clean up the disk.

So in the vast majority of cases, applications need not worry about the WAL file at all. SQLite will automatically take care of it. But it is possible to get SQLite into a state where the WAL file will grow without bound, causing excess disk space usage and slow queries speeds. The following bullets enumerate some of the ways that this can happen and how to avoid them.

- **Disabling the automatic checkpoint mechanism.** In its default configuration, SQLite will checkpoint the WAL file at the conclusion of any transaction when the WAL file is more than 1000 pages long. However, compile-time and run-time options exist that can disable or defer this automatic checkpoint. If an application disables the automatic checkpoint, then there is nothing to prevent the WAL file from growing excessively.

- **Checkpoint starvation.** A checkpoint is only able to run to completion, and reset the WAL file, if there are no other database connections using the WAL file. If another connection has a read transaction open, then the checkpoint cannot reset the WAL file because doing so might delete content out from under the reader. The checkpoint will do as much work as it can without upsetting the reader, but it cannot run to completion. The checkpoint will start up again where it left off after the next write transaction. This repeats until some checkpoint is able to complete.

  However, if a database has many concurrent overlapping readers and there is always at least one active reader, then no checkpoints will be able to complete and hence the WAL file will grow without bound.

  This scenario can be avoided by ensuring that there are "reader gaps": times when no processes are reading from the database and that checkpoints are attempted during those times. In applications with many concurrent readers, one might also consider running manual checkpoints with the SQLITE_CHECKPOINT_RESTART or SQLITE_CHECKPOINT_TRUNCATE option which will ensure that the checkpoint runs to completion before returning. The disadvantage of using SQLITE_CHECKPOINT_RESTART and SQLITE_CHECKPOINT_TRUNCATE is that readers might block while the checkpoint is running.

- **Very large write transactions.** A checkpoint can only complete when no other transactions are running, which means the WAL file cannot be reset in the middle of a write transaction. So a large change to a large database might result in a large WAL file. The WAL file will be checkpointed once the write transaction completes (assuming there are no other readers blocking it) but in the meantime, the file can grow very big.

  As of SQLite version 3.11.0, the WAL file for a single transaction should be proportional in size to the transaction itself. Pages that are changed by the transaction should only be written into the WAL file once. However, with older versions of SQLite, the same page might be written into the WAL file multiple times if the transaction grows larger than the page cache.

Implementation Of Shared-Memory For The WAL-Index

The wal-index is implemented using an ordinary file that is mmapped for robustness. Early (pre-release) implementations of WAL mode stored the wal-index in volatile shared-memory, such as files created in /dev/shm on Linux or /tmp on other unix systems. The problem with that approach is that processes with a different root directory (changed via chroot) will see different files and hence use different shared memory areas, leading to database corruption.

Other methods for creating nameless shared memory blocks are not portable across the various flavors of unix. And we could not find any method to create nameless shared memory blocks on windows. The only way we have found to guarantee that all processes accessing the same database file use the same shared memory is to create the shared memory by mmapping a file in the same directory as the database itself.

Using an ordinary disk file to provide shared memory has the disadvantage that it might actually do unnecessary disk I/O by writing the shared memory to disk. However, the developers do not think this is a major concern since the wal-index rarely exceeds 32 KiB in size and is never synced. Furthermore, the wal-index backing file is deleted when the last database connection disconnects, which often prevents any real disk I/O from ever happening.

Specialized applications for which the default implementation of shared memory is unacceptable can devise alternative methods via a custom VFS. For example, if it is known that a particular database will only be accessed by threads within a single process, the wal-index can be implemented using heap memory instead of true shared memory.

Use of WAL Without Shared-Memory

Beginning in SQLite version 3.7.4, WAL databases can be created, read, and written even if shared memory is unavailable as long as the locking_mode is set to EXCLUSIVE before the first attempted access. In other words, a process can interact with a WAL database without using shared memory if that process is guaranteed to be the only process accessing the database. This feature allows WAL databases to be created, read, and written by legacy VFSes that lack the "version 2" shared-memory methods xShmMap, xShmLock, xShmBarrier, and xShmUnmap on the sqlite3_io_methods object.

If EXCLUSIVE locking mode is set prior to the first WAL-mode database access, then SQLite never attempts to call any of the shared-memory methods and hence no shared-memory wal-index is ever created. In that case, the database connection remains in EXCLUSIVE mode as long as the journal mode is WAL; attempts to change the locking mode using "`PRAGMA locking_mode=NORMAL;`" are no-ops. The only way to change out of EXCLUSIVE locking mode is to first change out of WAL journal mode.

If NORMAL locking mode is in effect for the first WAL-mode database access, then the shared-memory wal-index is created. This means that the underlying VFS must support the "version 2" shared-memory. If the VFS does not support shared-memory methods, then the attempt to open a database that is already in WAL mode, or the attempt convert a database into WAL mode, will fail. As long as exactly one connection is using a shared-memory wal-index, the locking mode can be changed freely between NORMAL and EXCLUSIVE. It is only when the shared-memory wal-index is omitted, when the locking mode is EXCLUSIVE prior to the first WAL-mode database access, that the locking mode is stuck in EXCLUSIVE.

Backwards Compatibility

The database file format is unchanged for WAL mode. However, the WAL file and the wal-index are new concepts and so older versions of SQLite will not know how to recover a crashed SQLite database that was operating in WAL mode when the crash occurred. To prevent older versions of SQLite (prior to version 3.7.0, 2010-07-22) from trying to recover a WAL-mode database (and making matters worse) the database file format version numbers (bytes 18 and 19 in the database header) are increased from 1 to 2 in WAL mode. Thus, if an older version of SQLite attempts to connect to an SQLite database that is operating in WAL mode, it will report an error along the lines of "file is encrypted or is not a database".

One can explicitly change out of WAL mode using a pragma such as this:

```
PRAGMA journal_mode=DELETE;
```

Deliberately changing out of WAL mode changes the database file format version numbers back to 1 so that older versions of SQLite can once again access the database file.

Advocacy

SQLite As An Application File Format

Executive Summary

An SQLite database file with a defined schema often makes an excellent application file format. Here are a dozen reasons why this is so:

1. Simplified Application Development
2. Single-File Documents
3. High-Level Query Language
4. Accessible Content
5. Cross-Platform
6. Atomic Transactions
7. Incremental And Continuous Updates
8. Easily Extensible
9. Performance
10. Concurrent Use By Multiple Processes
11. Multiple Programming Languages
12. Better Applications

Each of these points will be described in more detail below, after first considering more closely the meaning of "application file format". See also the short version of this whitepaper.

What Is An Application File Format?

An "application file format" is the file format used to persist application state to disk or to exchange information between programs. There are thousands of application file formats in use today. Here are just a few examples:

- DOC - Word Perfect and Microsoft Office documents
- DWG - AutoCAD drawings
- PDF - Portable Document Format from Adobe
- XLS - Microsoft Excel Spreadsheet
- GIT - Git source code repository
- EPUB - The Electronic Publication format used by non-Kindle eBooks
- ODT - The Open Document format used by OpenOffice and others
- PPT - Microsoft PowerPoint presentations
- ODP - The Open Document presentation format used by OpenOffice and others

We make a distinction between a "file format" and an "application format". A file format is used to store a single object. So, for example, a GIF or JPEG file stores a single image, and an XHTML file stores text, so those are "file formats" and not "application formats". An EPUB file, in contrast, stores both text and images (as contained XHTML and GIF/JPEG files) and so it is considered a "application format". This article is about "application formats".

The boundary between a file format and an application format is fuzzy. This article calls JPEG a file format, but for an image editor, JPEG might be considered the application format. Much depends on context. For this article, let us say that a file format stores a single object and an application format stores many different objects and their relationships to one another.

Most application formats fit into one of these three categories:

1. **Fully Custom Formats.** Custom formats are specifically designed for a single application. DOC, DWG, PDF, XLS, and PPT are examples of custom formats. Custom formats are usually contained within a single file, for ease of transport. They are also usually binary, though the DWG format is a notable exception. Custom file formats require specialized application code to read and write and are not normally accessible from commonly available tools such as unix command-line programs and text editors. In other words, custom formats are usually "opaque blobs". To access the content of a custom application file format, one needs a tool specifically engineered to read and/or write that format.

2. **Pile-of-Files Formats.** Sometimes the application state is stored as a hierarchy of files. Git is a prime example of this, though the phenomenon occurs frequently in one-off and bespoke applications. A pile-of-files format essentially uses the filesystem as a key/value database, storing small chunks of information into separate files. This gives the advantage of making the content more accessible to common utility programs such as text editors or "awk" or "grep". But even if many of the files in a pile-of-files format are easily readable, there are usually some files that have their own custom format (example: Git "Packfiles") and are hence "opaque blobs" that are not readable or writable without specialized tools. It is also much less convenient to move a pile-of-files from one place or machine to another, than it is to move a single file. And it is hard to make a pile-of-files document into an email attachment, for example. Finally, a pile-of-files format breaks the "document metaphor": there is no one file that a user can point to that is "the document".

3. **Wrapped Pile-of-Files Formats.** Some applications use a Pile-of-Files that is then encapsulated into some kind of single-file container, usually a ZIP archive. EPUB, ODT,and ODP are examples of this approach. An EPUB book is really just a ZIP archive that contains various XHTML files for the text of book chapters, GIF and JPEG images for the artwork, and a specialized catalog file that tells the eBook reader how all the

> XML and image files fit together. OpenOffice documents (ODT and ODP) are also ZIP archives containing XML and images that represent their content as well as "catalog" files that show the interrelationships between the component parts.
>
> A wrapped pile-of-files format is a compromise between a full custom file format and a pure pile-of-files format. A wrapped pile-of-files format is not an opaque blob in the same sense as a custom format, since the component parts can still be accessed using any common ZIP archiver, but the format is not quite as accessible as a pure pile-of-files format because one does still need the ZIP archiver, and one cannot normally use command-line tools like "find" on the file hierarchy without first un-zipping it. On the other hand, a wrapped pile-of-files format does preserve the document metaphor by putting all content into a single disk file. And because it is compressed, the wrapped pile-of-files format tends to be more compact.
>
> As with custom file formats, and unlike pure pile-of-file formats, a wrapped pile-of-files format is not as easy to edit, since usually the entire file must be rewritten in order to change any component part.

The purpose of this document is to argue in favor of a fourth new catagory of application file format: An SQLite database file.

SQLite As The Application File Format

Any application state that can be recorded in a pile-of-files can also be recorded in an SQLite database with a simple key/value schema like this:

```
CREATE TABLE files(filename TEXT PRIMARY KEY, content BLOB);
```

If the content is compressed, then such an SQLite database is the same size (±1%) as an equivalent ZIP archive, and it has the advantage of being able to update individual "files" without rewriting the entire document.

But an SQLite database is not limited to a simple key/value structure like a pile-of-files database. An SQLite database can have dozens or hundreds or thousands of different of tables, with dozens or hundreds or thousands of fields per table, each with different datatypes and constraints and particular meanings, all cross-referencing each other, appropriately and automatically indexed for rapid retrieval, and all stored efficiently and compactly in a single disk file. And all of this structure is succinctly documented for humans by the SQL schema.

In other words, an SQLite database can do everything that a pile-of-files or wrapped pile-of-files format can do, plus much more, and with greater lucidity. An SQLite database is a more versatile container than key/value filesystem or a ZIP archive. (For a detailed example, see the OpenOffice case study essay.)

The power of an SQLite database could, in theory, be achieved using a custom file format. But any custom file format that is as expressive as a relational database would likely require an enormous design specification and many tens or hundreds of thousands of lines of code to implement. And the end result would be an "opaque blob" that is inaccessible without specialized tools.

Hence, in comparison to other approaches, the use of an SQLite database as an application file format has compelling advantages. Here are a few of these advantages, enumerated and expounded:

1. **Simplified Application Development.** No new code is needed for reading or writing the application file. One has merely to link against the SQLite library, or include the single "sqlite3.c" source file with the rest of the application C code, and SQLite will take care of all of the application file I/O. This can reduce application code size by many thousands of lines, with corresponding saving in development and maintenance costs.

   SQLite is one of the most used software libraries in the world. There are literally tens of billions of SQLite database files in use daily, on smartphones and gadgets and in desktop applications. SQLite is carefully tested and proven reliable. It is not a component that needs much tuning or debugging, allowing developers to stay focused on application logic.

2. **Single-File Documents.** An SQLite database is contained in a single file, which is easily copied or moved or attached. The "document" metaphor is preserved.

   SQLite does not have any file naming requirements and so the application can use any custom file suffix that it wants to help identify the file as "belonging" to the application. SQLite database files contain a 4-byte Application ID in their headers that can be set to an application-defined value and then used to identify the "type" of the document for utility programs such as file(1), further enhancing the document metaphor.

3. **High-Level Query Language.** SQLite is a complete relational database engine, which means that the application can access content using high-level queries. Application developers need not spend time thinking about "how" to retrieve the information they need from a document. Developers write SQL that expresses "what" information they want and let the database engine to figure out how to best retrieve that content. This helps developers operate "heads up" and remain focused on solving the user's problem, and avoid time spent "heads down" fiddling with low-level file formatting details.

A pile-of-files format can be viewed as a key/value database. A key/value database is better than no database at all. But without transactions or indices or a high-level query language or a proper schema, it much harder and more error prone to use a key/value database than a relational database.

4. **Accessible Content.** Information held in an SQLite database file is accessible using commonly available open-source command-line tools - tools that are installed by default on Mac and Linux systems and that are freely available as a self-contained EXE file on Windows. Unlike custom file formats, application-specific programs are not required to read or write content in an SQLite database. An SQLite database file is not an opaque blob. It is true that command-line tools such as text editors or "grep" or "awk" are not useful on an SQLite database, but the SQL query language is a much more powerful and convenient way for examining the content, so the inability to use "grep" and "awk" and the like is not seen as a loss.

   An SQLite database is a well-defined and well-documented file format that is in widespread use by literally millions of applications and is backwards compatible to its inception in 2004 and which promises to continue to be compatible in years to come. The longevity of SQLite database files is particularly important to bespoke applications, since it allows the document content to be accessed years or decades in the future, long after all traces of the original application have been lost. Data lives longer than code.

5. **Cross-Platform.** SQLite database files are portable between 32-bit and 64-bit machines and between big-endian and little-endian architectures and between any of the various flavors of Windows and Unix-like operating systems. The application using an SQLite application file format can store binary numeric data without having to worry about the byte-order of integers or floating point numbers. Text content can be read or written as UTF-8, UTF-16LE, or UTF-16BE and SQLite will automatically perform any necessary translations on-the-fly.

6. **Atomic Transactions.** Writes to an SQLite database are atomic. They either happen completely or not at all, even during system crashes or power failures. So there is no danger of corrupting a document just because the power happened to go out at the same instant that a change was being written to disk.

   SQLite is transactional, meaning that multiple changes can be grouped together such that either all or none of them occur, and so that the changes can be rolled back if a problem is found prior to commit. This allows an application to make a change incrementally, then run various sanity and consistency checks on the resulting data prior to committing the changes to disk. The Fossil DVCS uses this technique to verify that no repository history has been lost prior to each change.

7. **Incremental And Continuous Updates.** When writing to an SQLite database file, only those parts of the file that actually change are written out to disk. This makes the writing happen faster and saves wear on SSDs. This is an enormous advantage over custom and wrapped pile-of-files formats, both of which usually require a rewrite of the entire document in order to change a single byte. Pure pile-of-files formats can also do incremental updates to some extent, though the granularity of writes is usually larger with pile-of-file formats (a single file) than with SQLite (a single page).

   SQLite also supports continuous update. Instead of collecting changes in memory and then writing them to disk only on a File/Save action, changes can be written back to the disk as they occur. This avoids loss of work on a system crash or power failure. An automated undo/redo stack, managed using triggers, can be kept in the on-disk database, meaning that undo/redo can occur across session boundaries.

8. **Easily Extensible.** As an application grows, new features can be added to an SQLite application file format simply by adding new tables to the schema or by adding new columns to existing tables. Adding columns or tables does not change the meaning of prior queries, so with a modicum of care to ensuring that the meaning of legacy columns and tables are preserved, backwards compatibility is maintained.

   It is possible to extend custom or pile-of-files formats too, of course, but doing is often much harder. If indices are added, then all application code that changes the corresponding tables must be located and modified to keep those indices up-to-date. If columns are added, then all application code that accesses the corresponding table must be located and modified to take into account the new columns.

9. **Performance.** In many cases, an SQLite application file format will be faster than a custom or pile-of-files format. In the case of a custom format, SQLite often dramatically improves start-up times because instead of having to read and parse the entire document into memory, the application can do queries to extract only the information needed for the initial screen. As the application progresses, it only needs to load as much material as is needed to draw the next screen, and can discard information from prior screens that is no longer in use. This helps keep the memory footprint of the application under control.

   A pile-of-files format can be read incrementally just like SQLite. But many developers are surprised to learn that SQLite can read and write smaller BLOBs (less than about 100KB in size) from its database faster than those same blobs can be read or written as separate files from the filesystem. (See Internal Versus External BLOBs for further information.) There is overhead associated with operating a relational database engine, however one should not assume that direct file I/O is faster than SQLite database I/O, as often it is not.

In either case, if performance problems do arise in an SQLite application those problems can often be resolved by adding one or two CREATE INDEX statements to the schema or perhaps running ANALYZE one time and without having to touch a single line of application code. But if a performance problem comes up in a custom or pile-of-files format, the fix will often require extensive changes to application code to add and maintain new indices or to extract information using different algorithms.

10. **Concurrent Use By Multiple Processes.** SQLite automatically coordinates concurrent access to the same document from multiple threads and/or processes. Two or more applications can connect and read from the same document at the same time. Writes are serialized, but as writes normally only take milliseconds, applications simply take turns writing. SQLite automatically ensures that the low-level format of the document is uncorrupted. Accomplishing the same with a custom or pile-of-files format, in contrast, requires extensive support in the application. And the application logic needed to support concurrency is a notorious bug-magnet.

11. **Multiple Programming Languages.** Though SQLite is itself written in ANSI-C, interfaces exist for just about every other programming language you can think of: C++, C#, Objective-C, Java, Tcl, Perl, Python, Ruby, Erlang, JavaScript, and so forth. So programmers can develop in whatever language they are most comfortable with and which best matches the needs of the project.

    An SQLite application file format is a great choice in cases where there is a collection or "federation" of separate programs, often written in different languages and by different development teams. This comes up commonly in research or laboratory environments where one team is responsible for data acquisition and other teams are responsible for various stages of analysis. Each team can use whatever hardware, operating system, programming language and development methodology that they are most comfortable with, and as long as all programs use an SQLite database with a common schema, they can all interoperate.

12. **Better Applications.** If the application file format is an SQLite database, the complete documentation for that file format consists of the database schema, with perhaps a few extra words about what each table and column represents. The description of a custom file format, on the other hand, typically runs on for hundreds of pages. A pile-of-files format, while much simpler and easier to describe than a fully custom format, still tends to be much larger and more complex than an SQL schema dump, since the names and format for the individual files must still be described.

    This is not a trivial point. A clear, concise, and easy to understand file format is a crucial part of any application design. Fred Brooks, in his all-time best-selling computer science text, *The Mythical Man-Month* says:

> *Representation is the essence of computer programming. ... Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.*

Rob Pike, in his *Rules of Programming* expresses the same idea this way:

> *Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.*

Linus Torvalds used different words to say much the same thing on the Git mailing list on 2006-06-27:

> *Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*

The point is this: an SQL database schema almost always does a far better job of defining and organizing the tables and data structures and their relationships. And having clear, concise, and well-defined representation almost always results in an application that performs better, has fewer problems, and is easier to develop and maintain.

Conclusion

SQLite is not the perfect application file format for every situation. But in many cases, SQLite is a far better choice than either a custom file format, a pile-of-files, or a wrapped pile-of-files. SQLite is a high-level, stable, reliable, cross-platform, widely-deployed, extensible, performant, accessible, concurrent file format. It deserves your consideration as the standard file format on your next application design.

Well-Known Users of SQLite

SQLite is used by literally millions of applications with literally billions and billions of deployments. SQLite is the most widely deployed database engine in the world today.

A few of the better-known users of SQLite are shown below in alphabetical order. This is not a complete list. SQLite is in the public domain and so most developers use it in their projects without ever telling us.

□	Adobe uses SQLite as the application file format for their Photoshop Lightroom product. SQLite is also a standard part of the Adobe Integrated Runtime (AIR). It is reported that Acrobat Reader also uses SQLite.
□	Airbus confirms that SQLite is being used in the flight software for the A350 XWB family of aircraft.
□	Apple uses SQLite in many (most?) of the native applications running on Mac OS-X desktops and servers and on iOS devices such as iPhones and iPods. SQLite is also used in iTunes, even on non-Apple hardware.
□	Bentley Systems uses SQLite as the application file format for their Microstation CAD/CAM product.
□	Bosch uses SQLite in the multimedia systems install on GM, Nissan, and Suzuki automobiles. (link)
□	The increasingly popular Dropbox file archiving and synchronization service is reported to use SQLite as the primary data store on the client side.
□	Expensify uses SQLite as a server-side database engine for their enterprise-scale expense reporting software.
□	Facebook uses SQLite as the SQL database engine in their osquery product.
□	SQLite is the primary meta-data storage format for the Firefox Web Browser and the Thunderbird Email Reader from Mozilla.
□	Flame is a malware spy program that is reported to make heavy use of SQLite.
□	We believe that General Electric uses SQLite in some product or another because they twice wrote to the SQLite developers requesting the US Export Control Number for SQLite. So presumably GE is using SQLite in something that they are exporting. But nobody (outside of GE) seems to know what that might be.
□	Google uses SQLite in their in the Android cell-phone operating system, and in the Chrome Web Browser.
□	Intuit apparently uses SQLite in QuickBooks and in TurboTax to judge from some error reports from users seen here and here.
□	McAfee uses SQLite in its antivirus programs. Mentioned here and implied here.

□	Microsoft uses SQLite as a core component of Windows 10, and in individual numerous products.
□	The Navigation Data Stanard uses SQLite as its application file format.
□	The popular PHP programming language comes with both SQLite2 and SQLite3 built in.
□	SQLite comes bundled with the Python programming language since Python 2.5.
□	The REALbasic programming environment comes bundled with an enhanced version of SQLite that supports AES encryption.
□	The RedHat Package Manager (RPM) uses SQLite to track its state.
□	There are multiple sightings of SQLite in the Skype client for Mac OS X and Windows.
□	The Tcl/Tk programming language now comes with SQLite built-in. SQLite works particularly well with Tcl since SQLite was originally a Tcl extension that subsequently "escaped" into the wild.
□	A representative of Toshiba wrote to the SQLite developers and requested the US Export Control Number for SQLite. We infer from this that Toshiba is exporting something from the US that uses SQLite, but we do not know what that something is.

SQLite Technical/Design Documentation

How To Corrupt An SQLite Database File

An SQLite database is highly resistant to corruption. If an application crash, or an operating-system crash, or even a power failure occurs in the middle of a transaction, the partially written transaction should be automatically rolled back the next time the database file is accessed. The recovery process is fully automatic and does not require any action on the part of the user or the application.

Though SQLite is resistant to database corruption, it is not immune. This document describes the various ways that an SQLite database might go corrupt.

1.0 File overwrite by a rogue thread or process

SQLite database files are ordinary disk files. That means that any process can open the file and overwrite it with garbage. There is nothing that the SQLite library can do to defend against this.

1.1 Continuing to use a file descriptor after it has been closed

We have seen multiple cases where a file descriptor was open on a file, then that file descriptor was closed and reopened on an SQLite database. Later, some other thread continued to write into the old file descriptor, not realizing that the original file had been closed already. But because the file descriptor had been reopened by SQLite, the information that was intended to go into the original file ended up overwriting parts of the SQLite database, leading to corruption of the database.

One example of this occurred circa 2013-08-30 on the canonical repository for the Fossil DVCS. In that event, file descriptor 2 (standard error) was being erroneously closed (by stunnel, we suspect) prior to sqlite3_open_v2() so that the file descriptor used for the repository database file was 2. Later, an application bug caused an assert() statement to emit an error message by invoking write(2,...). But since file descriptor 2 was now connected to a database file, the error message overwrote part of the database. To guard against this kind of problem, SQLite version 3.8.1 and later refuse to use low-numbered file descriptors for database files. (See SQLITE_MINIMUM_FILE_DESCRIPTOR for additional information.)

Another example of corruption caused by using a closed file descriptor was reported by facebook engineers in a blog post on 2014-08-12.

1.2 Backup or restore while a transaction is active

Systems that run automatic backups in the background might try to make a backup copy of an SQLite database file while it is in the middle of a transaction. The backup copy then might contain some old and some new content, and thus be corrupt.

The best approach to make reliable backup copies of an SQLite database is to make use of the backup API that is part of the SQLite library. Failing that, it is safe to make a copy of an SQLite database file as long as there are no transactions in progress by any process. If the previous transaction failed, then it is important that any rollback journal (the `*-journal` file) or write-ahead log (the `*-wal` file) be copied together with the database file itself.

1.3 Deleting a hot journal

SQLite normally stores all content in a single disk file. However, while performing a transaction, information necessary to roll back that transaction following a crash or power failure is stored in auxiliary journal files. These journal files have the same name as the original database file with the addition of `-journal` or `-wal` suffix.

SQLite must see the journal files in order to recover from a crash or power failure. If the journal files are moved, deleted, or renamed after a crash or power failure, then automatic recovery will not work and the database may go corrupt.

Another manifestation of this problem is database corruption caused by inconsistent use of 8+3 filenames.

2.0 File locking problems

SQLite uses file locks on the database file, and on the write-ahead log or WAL file, to coordinate access between concurrent processes. Without coordination, two threads or processes might try to make incompatible changes to a database file at the same time, resulting in database corruption.

2.1 Filesystems with broken or missing lock implementations

SQLite depends on the underlying filesystem to do locking as the documentation says it will. But some filesystems contain bugs in their locking logic such that the locks do not always behave as advertised. This is especially true of network filesystems and NFS in particular. If

SQLite is used on a filesystem where the locking primitives contain bugs, and if two or more threads or processes try to access the same database at the same time, then database corruption might result.

2.2 Posix advisory locks canceled by a separate thread doing close()

The default locking mechanism used by SQLite on unix platforms is POSIX advisory locking. Unfortunately, POSIX advisory locking has design quirks that make it prone to misuse and failure. In particular, any thread in the same process with a file descriptor that is holding a POSIX advisory lock can override that lock using a different file descriptor. One particularly pernicious problem is that the `close()` system call will cancel all POSIX advisory locks on the same file for all threads and all file descriptors in the process.

So, for example, suppose a multi-thread process has two or more threads with separate SQLite database connections to the same database file. Then a third thread comes along and wants to read something out of that same database file on its own, without using the SQLite library. The third thread does an `open()` , a `read()` and then a `close()` . One would think this would be harmless. But the `close()` system call caused the locks held on the database by all the other threads to be dropped. Those other threads have no way of knowing that their locks have just been trashed (POSIX does not provide any mechanism to determine this) and so they keep on running under the assumption that their locks are still valid. This can lead to two or more threads or processes trying to write to the database at the same time, resulting in database corruption.

Note that it is perfectly safe for two or more threads to access the same SQLite database file using the SQLite library. The unix drivers for SQLite know about the POSIX advisory locking quirks and work around them. This problem only arises when a thread tries to bypass the SQLite library and read the database file directly.

2.2.1 Multiple copies of SQLite linked into the same application

As pointed out in the previous paragraph, SQLite takes steps to work around the quirks of POSIX advisory locking. Part of that work-around involves keeping a global list (mutex protected) of open SQLite database files. But, if multiple copies of SQLite are linked into the same application, then there will be multiple instances of this global list. Database connections opened using one copy of the SQLite library will be unaware of database connections opened using the other copy, and will be unable to work around the POSIX advisory locking quirks. A `close()` operation on one connection might unknowingly clear the locks on a different database connection, leading to database corruption.

The scenario above sounds far-fetched. But the SQLite developers are aware of at least one commercial product that was released with exactly this bug. The vendor came to the SQLite developers seeking help in tracking down some infrequent database corruption issues they were seeing on Linux and Mac. The problem was eventually traced to the fact that the application was linking against two separate copies of SQLite. The solution was to change the application build procedures to link against just one copy of SQLite instead of two.

2.3 Two processes using different locking protocols

The default locking mechanism used by SQLite on unix platforms is POSIX advisory locking, but there are other options. By selecting an alternative sqlite3_vfs using the sqlite3_open_v2() interface, an application can make use of other locking protocols that might be more appropriate to certain filesystems. For example, dot-file locking might be select for use in an application that has to run on an NFS filesystem that does not support POSIX advisory locking.

It is important that all connections to the same database file use the same locking protocol. If one application is using POSIX advisory locks and another application is using dot-file locking, then the two applications will not see each other's locks and will not be able to coordinate database access, possibly leading to database corruption.

2.4 Unlinking or renaming a database file while in use

If two processes have open connections to the same database file and one process closes its connection, unlinks the file, then creates a new database file in its place with the same name and reopens the new file, then the two processes will be talking to different database files with the same name. (Note that this is only possible on Posix and Posix-like systems that permit a file to be unlinked while it is still open for reading and writing. Windows does not allow this to occur.) Since rollback journals and WAL files are based on the name of the database file, the two different database files will share the same rollback journal or WAL file. A rollback or recovery for one of the databases might use content from the other database, resulting in corruption.

A similar problem occurs if a database file is renamed while it is opened and a new file is created with the old name.

Beginning with SQLite version 3.7.17, the unix OS interface will send SQLITE_WARNING messages to the error log if a database file is unlinked while it is still in use.

2.5 Multiple links to the same file

If a single database file has multiple links (either hard or soft links) then that is just another way of saying that the file has multiple names. If two or more processes open the database using different names, then they will use different rollback journals and WAL files. That means that if one process crashes, the other process will be unable to recover the transaction in progress because it will be looking in the wrong place for the appropriate journal.

Beginning with SQLite version 3.7.17, the unix OS interface will send SQLITE_WARNING messages to the error log if a database file has multiple hard links. As of this writing, SQLite still does not yet detect or warn about the use of database files through soft links.

3.0 Failure to sync

In order to guarantee that database files are always consistent, SQLite will occasionally ask the operating system to flush all pending writes to persistent storage then wait for that flush to complete. This is accomplished using the `fsync()` system call under unix and `FlushFileBuffers()` under Windows. We call this flush of pending writes a "sync".

Actually, if one is only concerned with atomic and consistent writes and is willing to forego durable writes, the sync operation does not need to wait until the content is completely stored on persistent media. Instead, the sync operation can be thought of as an I/O barrier. As long as all writes that occur before the sync are completed before any write that happens after the sync, no database corruption will occur. If sync is operating as an I/O barrier and not as a true sync, then a power failure or system crash might cause one or more previously committed transactions to roll back (in violation of the "durable" property of "ACID") but the database will at least continue to be consistent, and that is what most people care about.

3.1 Disk drives that do not honor sync requests

Unfortunately, most consumer-grade mass storage devices lie about syncing. Disk drives will report that content is safely on persistent media as soon as it reaches the track buffer and before actually being written to oxide. This makes the disk drives seem to operate faster (which is vitally important to the manufacturer so that they can show good benchmark numbers in trade magazines). And in fairness, the lie normally causes no harm, as long as there is no power loss or hard reset prior to the track buffer actually being written to oxide. But if a power loss or hard reset does occur, and if that results in content that was written after a sync reaching oxide while content written before the sync is still in a track buffer, then database corruption can occur.

USB flash memory sticks seem to be especially pernicious liars regarding sync requests. One can easily see this by committing a large transaction to an SQLite database on a USB memory stick. The COMMIT command will return relatively quickly, indicating that the memory stick has told the operating system and the operating system has told SQLite that all content is safely in persistent storage, and yet the LED on the end of the memory stick will continue flashing for several more seconds. Pulling out the memory stick while the LED is still flashing will frequently result in database corruption.

Note that SQLite must believe whatever the operating system and hardware tell it about the status of sync requests. There is no way for SQLite to detect that either is lying and that writes might be occurring out-of-order. However, SQLite in WAL mode is far more forgiving of out-of-order writes than in the default rollback journal modes. In WAL mode, the only time that a failed sync operation can cause database corruption is during a checkpoint operation. A sync failure during a COMMIT might result in loss of durability but not in a corrupt database file. Hence, one line of defense against database corruption due to failed sync operations is to use SQLite in WAL mode and to checkpoint as infrequently as possible.

3.2 Disabling sync using PRAGMAs

The sync operations that SQLite performs to help ensure integrity can be disabled at run-time using the synchronous pragma. By setting PRAGMA synchronous=OFF, all sync operations are omitted. This makes SQLite seem to run faster, but it also allows the operating system to freely reorder writes, which could result in database corruption if a power failure or hard reset occurs prior to all content reaching persistent storage.

For maximum reliability and for robustness against database corruption, SQLite should always be run with its default synchronous setting of FULL.

4.0 Disk Drive and Flash Memory Failures

An SQLite database can become corrupt if the file content changes due to a disk drive or flash memory failure. It is very rare, but disks will occasionally flip a bit in the middle of a sector.

4.1 Non-powersafe flash memory controllers

We are told that in some flash memory controllers the wear-leveling logic can cause random filesystem damage if power is interrupted during a write. This can manifest, for example, as random changes in the middle of a file that was not even open at the time of the power loss.

So, for example, a device would be writing content into an MP3 file in flash memory when a power loss occurs, and that could result in an SQLite database being corrupted even though the database as not even in use at the time of the power loss.

4.2 Fake capacity USB sticks

There are many fraudulent USB sticks in circulation that report to have a high capacity (ex: 8GB) but are really only capable of storing a much smaller amount (ex: 1GB). Attempts to write on these devices will often result in unrelated files being overwritten. Any use of a fraudulent flash memory device can easily lead to database corruption, therefore. Internet searches such as "fake capacity usb" will turn up lots of disturbing information about this problem.

5.0 Memory corruption

SQLite is a C-library that runs in the same address space as the application that it serves. That means that stray pointers, buffer overruns, heap corruption, or other malfunctions in the application can corrupt internal SQLite data structure and ultimately result in a corrupt database file. Normally these kinds of problems manifest themselves as segfaults prior to any database corruption occurring, but there have been instances where application code errors have caused SQLite to malfunction subtly so as to corrupt the database file rather than panicking.

The memory corruption problem becomes more acute when using memory-mapped I/O. When all or part of the database file is mapped into the application's address space, then a stray pointer the overwrites any part of that mapped space will immediately corrupt the database file, without requiring the application to do a subsequent write() system call.

6.0 Other operating system problems

Sometimes operating systems will exhibit non-standard behavior which can lead to problems. Sometimes this non-standard behavior is deliberate, and sometimes it is a mistake in the implementation. But in any event, if the operating performs differently from they way SQLite expects it to perform, the possibility of database corruption exists.

6.1 Linux Threads

Some older versions of Linux used the LinuxThreads library for thread support. LinuxThreads is similar to Pthreads, but is subtly different with respect to handling of POSIX advisory locks. SQLite versions 2.2.3 through 3.6.23 recognized that LinuxThreads where being used at runtime and took appropriate action to work around the non-standard behavior of LinuxThreads. But most modern Linux implementations make use of the newer, and correct, NPTL implementation of Pthreads. Beginning with SQLite version 3.7.0, the use of NPTL is assumed. No checks are made. Hence, recent versions of SQLite will subtly malfunction and may corrupt database files if used in multi-threaded application that run on older linux systems that make use of LinuxThreads.

6.2 Failures of mmap() on QNX

There exists some subtle problem with mmap() on QNX such that making a second mmap() call against the a single file descriptor can cause the memory obtained from the first mmap() call to be zeroed. SQLite on unix uses mmap() to create a shared memory region for transaction coordination in WAL mode, and it will call mmap() multiple times for large transactions. The QNX mmap() has been demonstrated to corrupt database file under that scenario. QNX engineers are aware of this problem and are working on a solution; the problem may have already been fixed by the time you read this.

When running on QNX, it is recommended that memory-mapped I/O never be used. Furthermore, to use WAL mode, it is recommended that applications employ the exclusive locking mode in order to use WAL without shared memory.

6.3 Filesystem Corruption

Since SQLite databases are ordinary disk files, any malfunction in the filesystem can corrupt the database. Filesystems in modern operating systems are very reliable, but errors do still occur. For example, on 2013-10-01 the SQLite database that holds the Wiki for Tcl/Tk went corrupt a few days after the host computer was moved to a dodgy build of the (linux) kernel that had issues in the filesystem layer. In that event, the filesystem eventually became so badly corrupted that the machine was unusable, but the earliest symptom of trouble was the corrupted SQLite database.

7.0 Bugs in SQLite

SQLite is very carefully tested to help ensure that it is as bug-free as possible. Among the many tests that are carried out for every SQLite version are tests that simulate power failures, I/O errors, and out-of-memory (OOM) errors and verify that no database corrupt occurs during any of these events. SQLite is also field-proven with approximately two billion active deployments with no serious problems.

Nevertheless, no software is 100% perfect. There have been a few historical bugs in SQLite (now fixed) that could cause database corruption. And there may be yet a few more that remain undiscovered. Because of the extensive testing and widespread use of SQLite, bugs that result in database corruption tend to be very obscure. The likelihood of an application encountering an SQLite bug is small. To illustrate this, an account is given below of all database-corruption bugs found in SQLite during the four-year period from 2009-04-01 to 2013-04-15. This account should give the reader an intuitive sense of the kinds of bugs in SQLite that manage to slip through testing procedures and make it into a release.

7.1 False corruption reports due to database shrinkage

If a database is written by SQLite version 3.7.0 or later and then written again by SQLite version 3.6.23 or earlier in such a way as to make the size of the database file decrease, then the next time that SQLite version 3.7.0 access the database file, it might report that the database file is corrupt. The database file is not really corrupt, however. Version 3.7.0 was simply being overly zealous in its corruption detection.

The problem was fixed on 2011-02-20. The fix first appears in SQLite version 3.7.6.

7.2 Corruption following switches between rollback and WAL modes

Repeatedly switching an SQLite database in and out of WAL mode and running the VACUUM command in between switches, in one process or thread, can cause another process or thread that has the database file open to miss the fact that the database has changed. That second process or thread might then try to modify the database using a stale cache and cause database corruption.

This problem was discovered during internal testing and has never been observed in the wild. The problem was fixed on 2011-01-27 and in version 3.7.5.

7.3 I/O while obtaining a lock leads to corruption

If the operating system returns an I/O error while attempting to obtain a certain lock on shared memory in WAL mode then SQLite might fail to reset its cache, which could lead to database corruption if subsequent writes are attempted.

Note that this problem only occurs if the attempt to acquire the lock resulted in an I/O error. If the lock is simply not granted (because some other thread or process is already holding a conflicting lock) then no corruption will ever occur. We are not aware of any operating systems that will fail with an I/O error while attempting to get a file lock on shared memory.

So this is a theoretical problem rather than a real problem. Needless to say, this problem has never been observed in the wild. The problem was discovered while doing stress testing of SQLite in a test harness that simulates I/O errors.

This problem was fixed on 2010-09-20 for SQLite version 3.7.3.

7.4 Database pages leak from the free page list

When content is deleted from an SQLite database, pages that are no longer used are added to a free list and are reused to hold content added but subsequent inserts. A bug in SQLite that was present in version 3.6.16 through 3.7.2 might cause pages to go missing out of the free list when incremental_vacuum was used. This would not cause data loss. But it would result in the database file being larger than necessary. And it would cause the integrity_check pragma to report pages missing from the free list.

This problem was fixed on 2010-08-23 for SQLite version 3.7.2.

7.5 Corruption following alternating writes from 3.6 and 3.7.

SQLite version 3.7.0 introduced a number of new enhancements to the SQLite database file format (such as but not limited to WAL). The 3.7.0 release was a shake-out release for these new features. We expected to find problems and were not disappointed.

If a database were originally created using SQLite version 3.7.0, then written by SQLite version 3.6.23.1 such that the size of the database file increased, then written again by SQLite version 3.7.0, the database file could go corrupt.

This problem was fixed on 2010-08-04 for SQLite version 3.7.1.

7.6 Race condition in recovery on windows system.

SQLite version 3.7.16.2 fixes a subtle race condition in the locking logic on Windows systems. When a database file is in need of recovery because the previous process writing to it crashed in the middle of a transaction and two or more processes try to open the that database at the same time, then the race condition might cause one of those processes to get a false indication that the recovery has already completed, allowing that process to continue using the database file without running recovery first. If that process writes to the file, then the file might go corrupt. This race condition had apparently existing in all prior versions of SQLite for Windows going back to 2004. But the race was very tight. Practically speaking, you need a fast multi-core machine in which you launch two processes to run recovery at the same moment on two separate cores. This defect was on Windows systems only and did not affect the posix OS interface.

SQLite's Use Of Temporary Disk Files

1.0 Introduction

One of the distinctive features of SQLite is that a database consists of a single disk file. This simplifies the use of SQLite since moving or backing up a database is a simple as copying a single file. It also makes SQLite appropriate for use as an application file format. But while a complete database is held in a single disk file, SQLite does make use of many temporary files during the course of processing a database.

This article describes the various temporary files that SQLite creates and uses. It describes when the files are created, when they are deleted, what they are used for, why they are important, and how to avoid them on systems where creating temporary files is expensive.

The manner in which SQLite uses temporary files is not considered part of the contract that SQLite makes with applications. The information in this document is a correct description of how SQLite operates at the time that this document was written or last updated. But there is no guarantee that future versions of SQLite will use temporary files in the same way. New kinds of temporary files might be employed and some of the current temporary file uses might be discontinued in future releases of SQLite.

2.0 Nine Kinds Of Temporary Files

SQLite currently uses nine distinct types of temporary files:

1. Rollback journals
2. Master journals
3. Write-ahead Log (WAL) files
4. Shared-memory files
5. Statement journals
6. TEMP databases
7. Materializations of views and subqueries
8. Transient indices
9. Transient databases used by VACUUM

Additional information about each of these temporary file types is in the sequel.

2.1 Rollback Journals

A rollback journal is a temporary file used to implement atomic commit and rollback capabilities in SQLite. (For a detailed discussion of how this works, see the separate document titled Atomic Commit In SQLite.) The rollback journal is always located in the same directory as the database file and has the same name as the database file except with the 8 characters "**-journal**" appended. The rollback journal is usually created when a transaction is first started and is usually deleted when a transaction commits or rolls back. The rollback journal file is essential for implementing the atomic commit and rollback capabilities of SQLite. Without a rollback journal, SQLite would be unable to rollback an incomplete transaction, and if a crash or power loss occurred in the middle of a transaction the entire database would likely go corrupt without a rollback journal.

The rollback journal is *usually* created and destroyed at the start and end of a transaction, respectively. But there are exceptions to this rule.

If a crash or power loss occurs in the middle of a transaction, then the rollback journal file is left on disk. The next time another application attempts to open the database file, it notices the presence of the abandoned rollback journal (we call it a "hot journal" in this circumstance) and uses the information in the journal to restore the database to its state prior to the start of the incomplete transaction. This is how SQLite implements atomic commit.

If an application puts SQLite in exclusive locking mode using the pragma:

```
PRAGMA locking_mode=EXCLUSIVE;
```

SQLite creates a new rollback journal at the start of the first transaction within an exclusive locking mode session. But at the conclusion of the transaction, it does not delete the rollback journal. The rollback journal might be truncated, or its header might be zeroed (depending on what version of SQLite you are using) but the rollback journal is not deleted. The rollback journal is not deleted until exclusive access mode is exited.

Rollback journal creation and deletion is also changed by the journal_mode pragma. The default journaling mode is DELETE, which is the default behavior of deleting the rollback journal file at the end of each transaction, as described above. The PERSIST journal mode foregoes the deletion of the journal file and instead overwrites the rollback journal header with zeros, which prevents other processes from rolling back the journal and thus has the same effect as deleting the journal file, though without the expense of actually removing the file from disk. In other words, journal mode PERSIST exhibits the same behavior as is seen in EXCLUSIVE locking mode. The OFF journal mode causes SQLite to omit the rollback journal, completely. In other words, no rollback journal is ever written if journal mode is set to OFF. The OFF journal mode disables the atomic commit and rollback capabilities of SQLite. The ROLLBACK command is not available when OFF journal mode is set. And if a crash or power loss occurs in the middle of a transaction that uses the OFF journal mode, no recovery is possible and the database file will likely go corrupt. The MEMORY journal mode causes the rollback journal to be stored in memory rather than on disk. The ROLLBACK command still works when the journal mode is MEMORY, but because no file exists on disks for recovery, a crash or power loss in the middle of a transaction that uses the MEMORY journal mode will likely result in a corrupt database.

2.2 Write-Ahead Log (WAL) Files

A write-ahead log or WAL file is used in place of a rollback journal when SQLite is operating in WAL mode. As with the rollback journal, the purpose of the WAL file is to implement atomic commit and rollback. The WAL file is always located in the same directory as the database file and has the same name as the database file except with the 4 characters "**-wal**" appended. The WAL file is created when the first connection to the database is opened and is normally removed when the last connection to the database closes. However, if the last connection does not shutdown cleanly, the WAL file will remain in the filesystem and will be automatically cleaned up the next time the database is opened.

2.3 Shared-Memory Files

When operating in WAL mode, all SQLite database connections associated with the same database file need to share some memory that is used as an index for the WAL file. In most implementations, this shared memory is implemented by calling mmap() on a file created for this sole purpose: the shared-memory file. The shared-memory file, if it exists, is located in the same directory as the database file and has the same name as the database file except with the 4 characters "**-shm**" appended. Shared memory files only exist while running in WAL mode.

The shared-memory file contains no persistent content. The only purpose of the shared-memory file is to provide a block of shared memory for use by multiple processes all accessing the same database in WAL mode. If the VFS is able to provide an alternative method for accessing shared memory, then that alternative method might be used rather than the shared-memory file. For example, if PRAGMA locking_mode is set to EXCLUSIVE (meaning that only one process is able to access the database file) then the shared memory will be allocated from heap rather than out of the shared-memory file, and the shared-memory file will never be created.

The shared-memory file has the same lifetime as its associated WAL file. The shared-memory file is created when the WAL file is created and is deleted when the WAL file is deleted. During WAL file recovery, the shared memory file is recreated from scratch based on the contents of the WAL file being recovered.

2.4 Master Journal Files

The master journal file is used as part of the atomic commit process when a single transaction makes changes to multiple databases that have been added to a single database connection using the ATTACH statement. The master journal file is always located in the same directory as the main database file (the main database file is the database that is identified in the original sqlite3_open(), sqlite3_open16(), or sqlite3_open_v2() call that created the database connection) with a randomized suffix. The master journal file contains the names of all of the various attached auxiliary databases that were changed during the transaction. The multi-database transaction commits when the master journal file is deleted. See the documentation titled Atomic Commit In SQLite for additional detail.

Without the master journal, the transaction commit on a multi-database transaction would be atomic for each database individually, but it would not be atomic across all databases. In other words, if the commit were interrupted in the middle by a crash or power loss, then the changes to one of the databases might complete while the changes to another database might roll back. The master journal causes all changes in all databases to either rollback or commit together.

The master journal file is only created for COMMIT operations that involve multiple database files where at least two of the databases meet all of the following requirements:

1. The database is modified by the transaction
2. The PRAGMA synchronous setting is not OFF
3. The PRAGMA journal_mode is not OFF, MEMORY, or WAL

This means that SQLite transactions are not atomic across multiple database files on a power-loss when the database files have synchronous turned off or when they are using journal modes of OFF, MEMORY, or WAL. For synchronous OFF and for journal_modes OFF and MEMORY, database will usually corrupt if a transaction commit is interrupted by a power loss. For WAL mode, individual database files are updated atomically across a power-loss, but in the case of a multi-file transactions, some files might rollback while others roll forward after power is restored.

2.5 Statement Journal Files

A statement journal file is used to rollback partial results of a single statement within a larger transaction. For example, suppose an UPDATE statement will attempt to modify 100 rows in the database. But after modifying the first 50 rows, the UPDATE hits a constraint violation which should block the entire statement. The statement journal is used to undo the first 50 row changes so that the database is restored to the state it was in at the start of the statement.

A statement journal is only created for an UPDATE or INSERT statement that might change multiple rows of a database and which might hit a constraint or a RAISE exception within a trigger and thus need to undo partial results. If the UPDATE or INSERT is not contained within BEGIN...COMMIT and if there are no other active statements on the same database connection then no statement journal is created since the ordinary rollback journal can be used instead. The statement journal is also omitted if an alternative conflict resolution algorithm is used. For example:

```
UPDATE OR FAIL ...
UPDATE OR IGNORE ...
UPDATE OR REPLACE ...
UPDATE OR ROLLBACK ...
INSERT OR FAIL ...
INSERT OR IGNORE ...
INSERT OR REPLACE ...
INSERT OR ROLLBACK ...
REPLACE INTO ....
```

The statement journal is given a randomized name, not necessarily in the same directory as the main database, and is automatically deleted at the conclusion of the transaction. The size of the statement journal is proportional to the size of the change implemented by the UPDATE or INSERT statement that caused the statement journal to be created.

2.6 TEMP Databases

Tables created using the "CREATE TEMP TABLE" syntax are only visible to the database connection in which the "CREATE TEMP TABLE" statement is originally evaluated. These TEMP tables, together with any associated indices, triggers, and views, are collectively stored in a separate temporary database file that is created as soon as the first "CREATE TEMP TABLE" statement is seen. This separate temporary database file also has an associated rollback journal. The temporary database file used to store TEMP tables is deleted automatically when the database connection is closed using sqlite3_close().

The TEMP database file is very similar to auxiliary database files added using the ATTACH statement, though with a few special properties. The TEMP database is always automatically deleted when the database connection is closed. The TEMP database always uses the synchronous=OFF and journal_mode=PERSIST PRAGMA settings. And, the TEMP database cannot be used with DETACH nor can another process ATTACH the TEMP database.

The temporary files associated with the TEMP database and its rollback journal are only created if the application makes use of the "CREATE TEMP TABLE" statement.

2.7 Materializations Of Views And Subqueries

Queries that contain subqueries must sometime evaluate the subqueries separately and store the results in a temporary table, then use the content of the temporary table to evaluate the outer query. We call this "materializing" the subquery. The query optimizer in SQLite attempts to avoid materializing, but sometimes it is not easily avoidable. The temporary tables created by materialization are each stored in their own separate temporary file, which is automatically deleted at the conclusion of the query. The size of these temporary tables depends on the amount of data in the materialization of the subquery, of course.

A subquery on the right-hand side of IN operator must often be materialized. For example:

```
SELECT * FROM ex1 WHERE ex1.a IN (SELECT b FROM ex2);
```

In the query above, the subquery "SELECT b FROM ex2" is evaluated and its results are stored in a temporary table (actually a temporary index) that allows one to determine whether or not a value ex2.b exists using a simple binary search. Once this table is constructed, the outer query is run and for each prospective result row a check is made to see if ex1.a is contained within the temporary table. The row is output only if the check is true.

To avoid creating the temporary table, the query might be rewritten as follows:

```
SELECT * FROM ex1 WHERE EXISTS(SELECT 1 FROM ex2 WHERE ex2.b=ex1.a);
```

Recent versions of SQLite (version 3.5.4 and later) will do this rewrite automatically if an index exists on the column ex2.b.

If the right-hand side of an IN operator can be list of values as in the following:

```
SELECT * FROM ex1 WHERE a IN (1,2,3);
```

List values on the right-hand side of IN are treated as a subquery that must be materialized. In other words, the previous statement acts as if it were:

```
SELECT * FROM ex1 WHERE a IN (SELECT 1 UNION ALL
                              SELECT 2 UNION ALL
                              SELECT 3);
```

A temporary index is always used to hold the values of the right-hand side of an IN operator when that right-hand side is a list of values.

Subqueries might also need to be materialized when they appear in the FROM clause of a SELECT statement. For example:

```
SELECT * FROM ex1 JOIN (SELECT b FROM ex2) AS t ON t.b=ex1.a;
```

Depending on the query, SQLite might need to materialize the "(SELECT b FROM ex2)" subquery into a temporary table, then perform the join between ex1 and the temporary table. The query optimizer tries to avoid this by "flattening" the query. In the previous example the query can be flattened, and SQLite will automatically transform the query into

```
SELECT ex1.*, ex2.b FROM ex1 JOIN ex2 ON ex2.b=ex1.a;
```

More complex queries may or may not be able to employ query flattening to avoid the temporary table. Whether or not the query can be flattened depends on such factors as whether or not the subquery or outer query contain aggregate functions, ORDER BY or GROUP BY clauses, LIMIT clauses, and so forth. The rules for when a query can and cannot be flattened are very complex and are beyond the scope of this document.

2.8 Transient Indices

SQLite may make use of transient indices to implement SQL language features such as:

- An ORDER BY or GROUP BY clause
- The DISTINCT keyword in an aggregate query
- Compound SELECT statements joined by UNION, EXCEPT, or INTERSECT

Each transient index is stored in its own temporary file. The temporary file for a transient index is automatically deleted at the end of the statement that uses it.

SQLite strives to implement ORDER BY clauses using a preexisting index. If an appropriate index already exists, SQLite will walk the index, rather than the underlying table, to extract the requested information, and thus cause the rows to come out in the desired order. But if SQLite cannot find an appropriate index it will evaluate the query and store each row in a transient index whose data is the row data and whose key is the ORDER BY terms. After the query is evaluated, SQLite goes back and walks the transient index from beginning to end in order to output the rows in the desired order.

SQLite implements GROUP BY by ordering the output rows in the order suggested by the GROUP BY terms. Each output row is compared to the previous to see if it starts a new "group". The ordering by GROUP BY terms is done in exactly the same way as the ordering by ORDER BY terms. A preexisting index is used if possible, but if no suitable index is available, a transient index is created.

The DISTINCT keyword on an aggregate query is implemented by creating a transient index in a temporary file and storing each result row in that index. As new result rows are computed a check is made to see if they already exist in the transient index and if they do the new result row is discarded.

The UNION operator for compound queries is implemented by creating a transient index in a temporary file and storing the results of the left and right subquery in the transient index, discarding duplicates. After both subqueries have been evaluated, the transient index is walked from beginning to end to generate the final output.

The EXCEPT operator for compound queries is implemented by creating a transient index in a temporary file, storing the results of the left subquery in this transient index, then removing the result from right subquery from the transient index, and finally walking the index from beginning to end to obtain the final output.

The INTERSECT operator for compound queries is implemented by creating two separate transient indices, each in a separate temporary file. The left and right subqueries are evaluated each into a separate transient index. Then the two indices are walked together and entries that appear in both indices are output.

Note that the UNION ALL operator for compound queries does not use transient indices by itself (though of course the right and left subqueries of the UNION ALL might use transient indices depending on how they are composed.)

2.9 Transient Database Used By VACUUM

The VACUUM command works by creating a temporary file and then rebuilding the entire database into that temporary file. Then the content of the temporary file is copied back into the original database file and the temporary file is deleted.

The temporary file created by the VACUUM command exists only for the duration of the command itself. The size of the temporary file will be no larger than the original database.

3.0 The SQLITE_TEMP_STORE Compile-Time Parameter and Pragma

The temporary files associated with transaction control, namely the rollback journal, master journal, write-ahead log (WAL) files, and shared-memory files, are always written to disk. But the other kinds of temporary files might be stored in memory only and never written to disk. Whether or not temporary files other than the rollback, master, and statement journals are written to disk or stored only in memory depends on the SQLITE_TEMP_STORE compile-time parameter, the temp_store pragma, and on the size of the temporary file.

The SQLITE_TEMP_STORE compile-time parameter is a #define whose value is an integer between 0 and 3, inclusive. The meaning of the SQLITE_TEMP_STORE compile-time parameter is as follows:

1. Temporary files are always stored on disk regardless of the setting of the temp_store pragma.
2. Temporary files are stored on disk by default but this can be overridden by the temp_store pragma.
3. Temporary files are stored in memory by default but this can be overridden by the temp_store pragma.
4. Temporary files are always stored in memory regardless of the setting of the temp_store pragma.

The default value of the SQLITE_TEMP_STORE compile-time parameter is 1, which means to store temporary files on disk but provide the option of overriding the behavior using the temp_store pragma.

The temp_store pragma has an integer value which also influences the decision of where to store temporary files. The values of the temp_store pragma have the following meanings:

1. Use either disk or memory storage for temporary files as determined by the SQLITE_TEMP_STORE compile-time parameter.
2. If the SQLITE_TEMP_STORE compile-time parameter specifies memory storage for temporary files, then override that decision and use disk storage instead. Otherwise follow the recommendation of the SQLITE_TEMP_STORE compile-time parameter.
3. If the SQLITE_TEMP_STORE compile-time parameter specifies disk storage for temporary files, then override that decision and use memory storage instead. Otherwise follow the recommendation of the SQLITE_TEMP_STORE compile-time parameter.

The default setting for the temp_store pragma is 0, which means to following the recommendation of SQLITE_TEMP_STORE compile-time parameter.

To reiterate, the SQLITE_TEMP_STORE compile-time parameter and the temp_store pragma only influence the temporary files other than the rollback journal and the master journal. The rollback journal and the master journal are always written to disk regardless of the settings of the SQLITE_TEMP_STORE compile-time parameter and the temp_store pragma.

4.0 Other Temporary File Optimizations

SQLite uses a page cache of recently read and written database pages. This page cache is used not just for the main database file but also for transient indices and tables stored in temporary files. If SQLite needs to use a temporary index or table and the

SQLITE_TEMP_STORE compile-time parameter and the temp_store pragma are set to store temporary tables and index on disk, the information is still initially stored in memory in the page cache. The temporary file is not opened and the information is not truly written to disk until the page cache is full.

This means that for many common cases where the temporary tables and indices are small (small enough to fit into the page cache) no temporary files are created and no disk I/O occurs. Only when the temporary data becomes too large to fit in RAM does the information spill to disk.

Each temporary table and index is given its own page cache which can store a maximum number of database pages determined by the SQLITE_DEFAULT_TEMP_CACHE_SIZE compile-time parameter. (The default value is 500 pages.) The maximum number of database pages in the page cache is the same for every temporary table and index. The value cannot be changed at run-time or on a per-table or per-index basis. Each temporary file gets its own private page cache with its own SQLITE_DEFAULT_TEMP_CACHE_SIZE page limit.

In-Memory Databases

An SQLite database is normally stored in a single ordinary disk file. However, in certain circumstances, the database might be stored in memory.

The most common way to force an SQLite database to exist purely in memory is to open the database using the special filename "**:memory:**". In other words, instead of passing the name of a real disk file into one of the sqlite3_open(), sqlite3_open16(), or sqlite3_open_v2() functions, pass in the string ":memory:". For example:

```
rc = sqlite3_open(":memory:", &db);
```

When this is done, no disk file is opened. Instead, a new database is created purely in memory. The database ceases to exist as soon as the database connection is closed. Every :memory: database is distinct from every other. So, opening two database connections each with the filename ":memory:" will create two independent in-memory databases.

The special filename ":memory:" can be used anywhere that a database filename is permitted. For example, it can be used as the *filename* in an ATTACH command:

```
ATTACH DATABASE ':memory:' AS aux1;
```

Note that in order for the special ":memory:" name to apply and to create a pure in-memory database, there must be no additional text in the filename. Thus, a disk-based database can be created in a file by prepending a pathname, like this: "./:memory:".

The special ":memory:" filename also works when using URI filenames. For example:

```
rc = sqlite3_open("file::memory:", &db);
```

Or,

```
ATTACH DATABASE 'file::memory:' AS aux1;
```

In-memory Databases And Shared Cache

In-memory databases are allowed to use shared cache if they are opened using a URI filename. If the unadorned ":memory:" name is used to specify the in-memory database, then that database always has a private cache and is this only visible to the database connection that originally opened it. However, the same in-memory database can be opened by two or more database connections as follows:

```
rc = sqlite3_open("file::memory:?cache=shared", &db);
```

Or,

```
ATTACH DATABASE 'file::memory:?cache=shared' AS aux1;
```

This allows separate database connections to share the same in-memory database. Of course, all database connections sharing the in-memory database need to be in the same process. The database is automatically deleted and memory is reclaimed when the last connection to the database closes.

If two or more distinct but shareable in-memory databases are needed in a single process, then the mode=memory query parameter can be used with a URI filename to create a named in-memory database:

```
rc = sqlite3_open("file:memdb1?mode=memory&cache=shared", &db);
```

Or,

```
ATTACH DATABASE 'file:memdb1?mode=memory&cache=shared' AS aux1;
```

When an in-memory database is named in this way, it will only share its cache with another connection that uses exactly the same name.

Temporary Databases

When the name of the database file handed to sqlite3_open() or to ATTACH is an empty string, then a new temporary file is created to hold the database.

```
rc = sqlite3_open("", &db);
```

```
ATTACH DATABASE '' AS aux2;
```

A different temporary file is created each time, so that just like as with the special ":memory:" string, two database connections to temporary databases each have their own private database. Temporary databases are automatically deleted when the connection that created them closes.

Even though a disk file is allocated for each temporary database, in practice the temporary database usually resides in the in-memory pager cache and hence is very little difference between a pure in-memory database created by ":memory:" and a temporary database created by an empty filename. The sole difference is that a ":memory:" database must remain in memory at all times whereas parts of a temporary database might be flushed to disk if database becomes large or if SQLite comes under memory pressure.

The previous paragraphs describe the behavior of temporary databases under the default SQLite configuration. An application can use the temp_store pragma and the SQLITE_TEMP_STORE compile-time parameter to force temporary databases to behave as pure in-memory databases, if desired.

Atomic Commit In SQLite

1.0 Introduction

An important feature of transactional databases like SQLite is "atomic commit". Atomic commit means that either all database changes within a single transaction occur or none of them occur. With atomic commit, it is as if many different writes to different sections of the database file occur instantaneously and simultaneously. Real hardware serializes writes to mass storage, and writing a single sector takes a finite amount of time. So it is impossible to truly write many different sectors of a database file simultaneously and/or instantaneously. But the atomic commit logic within SQLite makes it appear as if the changes for a transaction are all written instantaneously and simultaneously.

SQLite has the important property that transactions appear to be atomic even if the transaction is interrupted by an operating system crash or power failure.

This article describes the techniques used by SQLite to create the illusion of atomic commit.

The information in this article applies only when SQLite is operating in "rollback mode", or in other words when SQLite is not using a write-ahead log. SQLite still supports atomic commit when write-ahead logging is enabled, but it accomplishes atomic commit by a different mechanism from the one described in this article. See the write-ahead log documentation for additional information on how SQLite supports atomic commit in that context.

2.0 Hardware Assumptions

Throughout this article, we will call the mass storage device "disk" even though the mass storage device might really be flash memory.

We assume that disk is written in chunks which we call a "sector". It is not possible to modify any part of the disk smaller than a sector. To change a part of the disk smaller than a sector, you have to read in the full sector that contains the part you want to change, make the change, then write back out the complete sector.

On a traditional spinning disk, a sector is the minimum unit of transfer in both directions, both reading and writing. On flash memory, however, the minimum size of a read is typically much smaller than a minimum write. SQLite is only concerned with the minimum write amount and so for the purposes of this article, when we say "sector" we mean the minimum amount of data that can be written to mass storage in a single go.

Prior to SQLite version 3.3.14, a sector size of 512 bytes was assumed in all cases. There was a compile-time option to change this but the code had never been tested with a larger value. The 512 byte sector assumption seemed reasonable since until very recently all disk drives used a 512 byte sector internally. However, there has recently been a push to increase the sector size of disks to 4096 bytes. Also the sector size for flash memory is usually larger than 512 bytes. For these reasons, versions of SQLite beginning with 3.3.14 have a method in the OS interface layer that interrogates the underlying filesystem to find the true sector size. As currently implemented (version 3.5.0) this method still returns a hard-coded value of 512 bytes, since there is no standard way of discovering the true sector size on either Unix or Windows. But the method is available for embedded device manufactures to tweak according to their own needs. And we have left open the possibility of filling in a more meaningful implementation on Unix and Windows in the future.

SQLite has traditionally assumed that a sector write is not atomic. However, SQLite does always assume that a sector write is linear. By "linear" we mean that SQLite assumes that when writing a sector, the hardware begins at one end of the data and writes byte by byte until it gets to the other end. The write might go from beginning to end or from end to beginning. If a power failure occurs in the middle of a sector write it might be that part of the sector was modified and another part was left unchanged. The key assumption by SQLite is that if any part of the sector gets changed, then either the first or the last bytes will be changed. So the hardware will never start writing a sector in the middle and work towards the ends. We do not know if this assumption is always true but it seems reasonable.

The previous paragraph states that SQLite does not assume that sector writes are atomic. This is true by default. But as of SQLite version 3.5.0, there is a new interface called the Virtual File System (VFS) interface. The VFS is the only means by which SQLite communicates to the underlying filesystem. The code comes with default VFS implementations for Unix and Windows and there is a mechanism for creating new custom VFS implementations at runtime. In this new VFS interface there is a method called xDeviceCharacteristics. This method interrogates the underlying filesystem to discover various properties and behaviors that the filesystem may or may not exhibit. The xDeviceCharacteristics method might indicate that sector writes are atomic, and if it does so indicate, SQLite will try to take advantage of that fact. But the default xDeviceCharacteristics method for both Unix and Windows does not indicate atomic sector writes and so these optimizations are normally omitted.

SQLite assumes that the operating system will buffer writes and that a write request will return before data has actually been stored in the mass storage device. SQLite further assumes that write operations will be reordered by the operating system. For this reason, SQLite does a "flush" or "fsync" operation at key points. SQLite assumes that the flush or fsync will not return until all pending write operations for the file that is being flushed have completed. We are told that the flush and fsync primitives are broken on some versions of

Windows and Linux. This is unfortunate. It opens SQLite up to the possibility of database corruption following a power loss in the middle of a commit. However, there is nothing that SQLite can do to test for or remedy the situation. SQLite assumes that the operating system that it is running on works as advertised. If that is not quite the case, well then hopefully you will not lose power too often.

SQLite assumes that when a file grows in length that the new file space originally contains garbage and then later is filled in with the data actually written. In other words, SQLite assumes that the file size is updated before the file content. This is a pessimistic assumption and SQLite has to do some extra work to make sure that it does not cause database corruption if power is lost between the time when the file size is increased and when the new content is written. The xDeviceCharacteristics method of the VFS might indicate that the filesystem will always write the data before updating the file size. (This is the SQLITE_IOCAP_SAFE_APPEND property for those readers who are looking at the code.) When the xDeviceCharacteristics method indicates that files content is written before the file size is increased, SQLite can forego some of its pedantic database protection steps and thereby decrease the amount of disk I/O needed to perform a commit. The current implementation, however, makes no such assumptions for the default VFSes for Windows and Unix.

SQLite assumes that a file deletion is atomic from the point of view of a user process. By this we mean that if SQLite requests that a file be deleted and the power is lost during the delete operation, once power is restored either the file will exist completely with all if its original content unaltered, or else the file will not be seen in the filesystem at all. If after power is restored the file is only partially deleted, if some of its data has been altered or erased, or the file has been truncated but not completely removed, then database corruption will likely result.

SQLite assumes that the detection and/or correction of bit errors caused by cosmic rays, thermal noise, quantum fluctuations, device driver bugs, or other mechanisms, is the responsibility of the underlying hardware and operating system. SQLite does not add any redundancy to the database file for the purpose of detecting corruption or I/O errors. SQLite assumes that the data it reads is exactly the same data that it previously wrote.

By default, SQLite assumes that an operating system call to write a range of bytes will not damage or alter any bytes outside of that range even if a power loss or OS crash occurs during that write. We call this the "powersafe overwrite" property. Prior to version 3.7.9, SQLite did not assume powersafe overwrite. But with the standard sector size increasing from 512 to 4096 bytes on most disk drives, it has become necessary to assume powersafe overwrite in order to maintain historical performance levels and so powersafe overwrite is assumed by default in recent versions of SQLite. The assumption of powersafe overwrite property can be disabled at compile-time or a run-time if desired. See the powersafe overwrite documentation for further details.

3.0 Single File Commit

We begin with an overview of the steps SQLite takes in order to perform an atomic commit of a transaction against a single database file. The details of file formats used to guard against damage from power failures and techniques for performing an atomic commit across multiple databases are discussed in later sections.

3.1 Initial State

The state of the computer when a database connection is first opened is shown conceptually by the diagram at the right. The area of the diagram on the extreme right (labeled "Disk") represents information stored on the mass storage device. Each rectangle is a sector. The blue color represents that the sectors contain original data. The middle area is the operating systems disk cache. At the onset of our example, the cache is cold and this is represented by leaving the rectangles of the disk cache empty. The left area of the diagram shows the content of memory for the process that is using SQLite. The database connection has just been opened and no information has been read yet, so the user space is empty.

3.2 Acquiring A Read Lock

Before SQLite can write to a database, it must first read the database to see what is there already. Even if it is just appending new data, SQLite still has to read in the database schema from the **sqlite_master** table so that it can know how to parse the INSERT statements and discover where in the database file the new information should be stored.

The first step toward reading from the database file is obtaining a shared lock on the database file. A "shared" lock allows two or more database connections to read from the database file at the same time. But a shared lock prevents another database connection from writing to the database file while we are reading it. This is necessary because if another database connection were writing to the database file at the same time we are reading from the database file, we might read some data before the change and other data after the change. This would make it appear as if the change made by the other process is not atomic.

Notice that the shared lock is on the operating system disk cache, not on the disk itself. File locks really are just flags within the operating system kernel, usually. (The details depend on the specific OS layer interface.) Hence, the lock will instantly vanish if the operating system crashes or if there is a power loss. It is usually also the case that the lock will vanish if the process that created the lock exits.

3.3 Reading Information Out Of The Database

After the shared lock is acquired, we can begin reading information from the database file. In this scenario, we are assuming a cold cache, so information must first be read from mass storage into the operating system cache then transferred from operating system cache into user space. On subsequent reads, some or all of the information might already be found in the operating system cache and so only the transfer to user space would be required.

Usually only a subset of the pages in the database file are read. In this example we are showing three pages out of eight being read. In a typical application, a database will have thousands of pages and a query will normally only touch a small percentage of those pages.

3.4 Obtaining A Reserved Lock

Before making changes to the database, SQLite first obtains a "reserved" lock on the database file. A reserved lock is similar to a shared lock in that both a reserved lock and shared lock allow other processes to read from the database file. A single reserve lock can coexist with multiple shared locks from other processes. However, there can only be a single reserved lock on the database file. Hence only a single process can be attempting to write to the database at one time.

The idea behind a reserved lock is that it signals that a process intends to modify the database file in the near future but has not yet started to make the modifications. And because the modifications have not yet started, other processes can continue to read from the database. However, no other process should also begin trying to write to the database.

3.5 Creating A Rollback Journal File

Prior to making any changes to the database file, SQLite first creates a separate rollback journal file and writes into the rollback journal the original content of the database pages that are to be altered. The idea behind the rollback journal is that it contains all information needed to restore the database back to its original state.

The rollback journal contains a small header (shown in green in the diagram) that records the original size of the database file. So if a change causes the database file to grow, we will still know the original size of the database. The page number is stored together with each database page that is written into the rollback journal.

When a new file is created, most desktop operating systems (Windows, Linux, Mac OS X) will not actually write anything to disk. The new file is created in the operating systems disk cache only. The file is not created on mass storage until sometime later, when the operating system has a spare moment. This creates the impression to users that I/O is happening much faster than is possible when doing real disk I/O. We illustrate this idea in the diagram to the right by showing that the new rollback journal appears in the operating system disk cache only and not on the disk itself.

3.6 Changing Database Pages In User Space

After the original page content has been saved in the rollback journal, the pages can be modified in user memory. Each database connection has its own private copy of user space, so the changes that are made in user space are only visible to the database connection that is making the changes. Other database connections still see the information in operating system disk cache buffers which have not yet been changed. And so even though one process is busy modifying the database, other processes can continue to read their own copies of the original database content.

3.7 Flushing The Rollback Journal File To Mass Storage

The next step is to flush the content of the rollback journal file to nonvolatile storage. As we will see later, this is a critical step in insuring that the database can survive an unexpected power loss. This step also takes a lot of time, since writing to nonvolatile storage is normally a slow operation.

This step is usually more complicated than simply flushing the rollback journal to the disk. On most platforms two separate flush (or fsync()) operations are required. The first flush writes out the base rollback journal content. Then the header of the rollback journal is modified to show the number of pages in the rollback journal. Then the header is flushed to disk. The details on why we do this header modification and extra flush are provided in a later section of this paper.

3.8 Obtaining An Exclusive Lock

Prior to making changes to the database file itself, we must obtain an exclusive lock on the database file. Obtaining an exclusive lock is really a two-step process. First SQLite obtains a "pending" lock. Then it escalates the pending lock to an exclusive lock.

A pending lock allows other processes that already have a shared lock to continue reading the database file. But it prevents new shared locks from being established. The idea behind a pending lock is to prevent writer starvation caused by a large pool of readers. There might be dozens, even hundreds, of other processes trying to read the database file. Each process acquires a shared lock before it starts reading, reads what it needs, then releases the shared lock. If, however, there are many different processes all reading from the same database, it might happen that a new process always acquires its shared lock before the previous process releases its shared lock. And so there is never an instant when there are no shared locks on the database file and hence there is never an opportunity for the writer to seize the exclusive lock. A pending lock is designed to prevent that cycle by allowing existing shared locks to proceed but blocking new shared locks from being established. Eventually all shared locks will clear and the pending lock will then be able to escalate into an exclusive lock.

3.9 Writing Changes To The Database File

Once an exclusive lock is held, we know that no other processes are reading from the database file and it is safe to write changes into the database file. Usually those changes only go as far as the operating systems disk cache and do not make it all the way to mass storage.

3.10 Flushing Changes To Mass Storage

Another flush must occur to make sure that all the database changes are written into nonvolatile storage. This is a critical step to ensure that the database will survive a power loss without damage. However, because of the inherent slowness of writing to disk or flash memory, this step together with the rollback journal file flush in section 3.7 above takes up most of the time required to complete a transaction commit in SQLite.

3.11 Deleting The Rollback Journal

After the database changes are all safely on the mass storage device, the rollback journal file is deleted. This is the instant where the transaction commits. If a power failure or system crash occurs prior to this point, then recovery processes to be described later make it appear as if no changes were ever made to the database file. If a power failure or system crash occurs after the rollback journal is deleted, then it appears as if all changes have been written to disk. Thus, SQLite gives the appearance of having made no changes to the database file or having made the complete set of changes to the database file depending on whether or not the rollback journal file exists.

Deleting a file is not really an atomic operation, but it appears to be from the point of view of a user process. A process is always able to ask the operating system "does this file exist?" and the process will get back a yes or no answer. After a power failure that occurs during a transaction commit, SQLite will ask the operating system whether or not the rollback journal file exists. If the answer is "yes" then the transaction is incomplete and is rolled back. If the answer is "no" then it means the transaction did commit.

The existence of a transaction depends on whether or not the rollback journal file exists and the deletion of a file appears to be an atomic operation from the point of view of a user-space process. Therefore, a transaction appears to be an atomic operation.

The act of deleting a file is expensive on many systems. As an optimization, SQLite can be configured to truncate the journal file to zero bytes in length or overwrite the journal file header with zeros. In either case, the resulting journal file is no longer capable of rolling back and so the transaction still commits. Truncating a file to zero length, like deleting a file, is assumed to be an atomic operation from the point of view of a user process. Overwriting the header of the journal with zeros is not atomic, but if any part of the header is malformed the journal will not roll back. Hence, one can say that the commit occurs as soon as the header is sufficiently changed to make it invalid. Typically this happens as soon as the first byte of the header is zeroed.

3.12 Releasing The Lock

The last step in the commit process is to release the exclusive lock so that other processes can once again start accessing the database file.

In the diagram at the right, we show that the information that was held in user space is cleared when the lock is released. This used to be literally true for older versions of SQLite. But more recent versions of SQLite keep the user space information in memory in case it might be needed again at the start of the next transaction. It is cheaper to reuse information that is already in local memory than to transfer the information back from the operating system disk cache or to read it off of the disk drive again. Prior to reusing the information in user space, we must first reacquire the shared lock and then we have to check to make sure that no other process modified the database file while we were not holding a lock. There is a counter in the first page of the database that is incremented every time the database file is modified. We can find out if another process has modified the database by checking that counter. If the database was modified, then the user space cache must be cleared and reread. But it is commonly the case that no changes have been made and the user space cache can be reused for a significant performance savings.

4.0 Rollback

An atomic commit is supposed to happen instantaneously. But the processing described above clearly takes a finite amount of time. Suppose the power to the computer were cut part way through the commit operation described above. In order to maintain the illusion that the changes were instantaneous, we have to "rollback" any partial changes and restore the database to the state it was in prior to the beginning of the transaction.

4.1 When Something Goes Wrong...

Suppose the power loss occurred during step 3.10 above, while the database changes were being written to disk. After power is restored, the situation might be something like what is shown to the right. We were trying to change three pages of the database file but only one page was successfully written. Another page was partially written and a third page was not written at all.

The rollback journal is complete and intact on disk when the power is restored. This is a key point. The reason for the flush operation in step 3.7 is to make absolutely sure that all of the rollback journal is safely on nonvolatile storage prior to making any changes to the database file itself.

4.2 Hot Rollback Journals

The first time that any SQLite process attempts to access the database file, it obtains a shared lock as described in section 3.2 above. But then it notices that there is a rollback journal file present. SQLite then checks to see if the rollback journal is a "hot journal". A hot journal is a rollback journal that needs to be played back in order to restore the database to a sane state. A hot journal only exists when an earlier process was in the middle of committing a transaction when it crashed or lost power.

A rollback journal is a "hot" journal if all of the following are true:

- The rollback journal exists.
- The rollback journal is not an empty file.
- There is no reserved lock on the main database file.
- The header of the rollback journal is well-formed and in particular has not been zeroed out.
- The rollback journal does not contain the name of a master journal file (see section 5.5 below) or if does contain the name of a master journal, then that master journal file exists.

The presence of a hot journal is our indication that a previous process was trying to commit a transaction but it aborted for some reason prior to the completion of the commit. A hot journal means that the database file is in an inconsistent state and needs to be repaired (by rollback) prior to being used.

4.3 Obtaining An Exclusive Lock On The Database

The first step toward dealing with a hot journal is to obtain an exclusive lock on the database file. This prevents two or more processes from trying to rollback the same hot journal at the same time.

4.4 Rolling Back Incomplete Changes

Once a process obtains an exclusive lock, it is permitted to write to the database file. It then proceeds to read the original content of pages out of the rollback journal and write that content back to where it came from in the database file. Recall that the header of the rollback journal records the original size of the database file prior to the start of the aborted transaction. SQLite uses this information to truncate the database file back to its original size in cases where the incomplete transaction caused the database to grow. At the end of this step, the database should be the same size and contain the same information as it did before the start of the aborted transaction.

4.5 Deleting The Hot Journal

After all information in the rollback journal has been played back into the database file (and flushed to disk in case we encounter yet another power failure), the hot rollback journal can be deleted.

As in section 3.11, the journal file might be truncated to zero length or its header might be overwritten with zeros as an optimization on systems where deleting a file is expensive. Either way, the journal is no longer hot after this step.

4.6 Continue As If The Uncompleted Writes Had Never Happened

The final recovery step is to reduce the exclusive lock back to a shared lock. Once this happens, the database is back in the state that it would have been if the aborted transaction had never started. Since all of this recovery activity happens completely automatically and transparently, it appears to the program using SQLite as if the aborted transaction had never begun.

5.0 Multi-file Commit

SQLite allows a single database connection to talk to two or more database files simultaneously through the use of the ATTACH DATABASE command. When multiple database files are modified within a single transaction, all files are updated atomically. In other words, either all of the database files are updated or else none of them are. Achieving an atomic commit across multiple database files is more complex that doing so for a single file. This section describes how SQLite works that bit of magic.

5.1 Separate Rollback Journals For Each Database

When multiple database files are involved in a transaction, each database has its own rollback journal and each database is locked separately. The diagram at the right shows a scenario where three different database files have been modified within one transaction. The situation at this step is analogous to the single-file transaction scenario at step 3.6. Each database file has a reserved lock. For each database, the original content of pages that are being changed have been written into the rollback journal for that database, but the content of the journals have not yet been flushed to disk. No changes have been made to the database file itself yet, though presumably there are changes being held in user memory.

For brevity, the diagrams in this section are simplified from those that came before. Blue color still signifies original content and pink still signifies new content. But the individual pages in the rollback journal and the database file are not shown and we are not making the distinction between information in the operating system cache and information that is on disk. All of these factors still apply in a multi-file commit scenario. They just take up a lot of space in the diagrams and they do not add any new information, so they are omitted here.

5.2 The Master Journal File

The next step in a multi-file commit is the creation of a "master journal" file. The name of the master journal file is the same name as the original database filename (the database that was opened using the sqlite3_open() interface, not one of the ATTACHed auxiliary databases) with the text "**-mj***HHHHHHHH*" appended where *HHHHHHHH* is a random 32-bit hexadecimal number. The random *HHHHHHHH* suffix changes for every new master journal.

(Nota bene: The formula for computing the master journal filename given in the previous paragraph corresponds to the implementation as of SQLite version 3.5.0. But this formula is not part of the SQLite specification and is subject to change in future releases.)

Unlike the rollback journals, the master journal does not contain any original database page content. Instead, the master journal contains the full pathnames for rollback journals for every database that is participating in the transaction.

After the master journal is constructed, its content is flushed to disk before any further actions are taken. On Unix, the directory that contains the master journal is also synced in order to make sure the master journal file will appear in the directory following a power failure.

The purpose of the master journal is to ensure that multi-file transactions are atomic across a power-loss. But if the database files have other settings that compromise integrity across a power-loss event (such as PRAGMA synchronous=OFF or PRAGMA journal_mode=MEMORY) then the creation of the master journal is omitted, as an optimization.

5.3 Updating Rollback Journal Headers

The next step is to record the full pathname of the master journal file in the header of every rollback journal. Space to hold the master journal filename was reserved at the beginning of each rollback journal as the rollback journals were created.

The content of each rollback journal is flushed to disk both before and after the master journal filename is written into the rollback journal header. It is important to do both of these flushes. Fortunately, the second flush is usually inexpensive since typically only a single page of the journal file (the first page) has changed.

This step is analogous to step 3.7 in the single-file commit scenario described above.

5.4 Updating The Database Files

Once all rollback journal files have been flushed to disk, it is safe to begin updating database files. We have to obtain an exclusive lock on all database files before writing the changes. After all the changes are written, it is important to flush the changes to disk so that they will be preserved in the event of a power failure or operating system crash.

This step corresponds to steps 3.8, 3.9, and 3.10 in the single-file commit scenario described previously.

5.5 Delete The Master Journal File

The next step is to delete the master journal file. This is the point where the multi-file transaction commits. This step corresponds to step 3.11 in the single-file commit scenario where the rollback journal is deleted.

If a power failure or operating system crash occurs at this point, the transaction will not rollback when the system reboots even though there are rollback journals present. The difference is the master journal pathname in the header of the rollback journal. Upon restart, SQLite only considers a journal to be hot and will only playback the journal if there is no master journal filename in the header (which is the case for a single-file commit) or if the master journal file still exists on disk.

5.6 Clean Up The Rollback Journals

The final step in a multi-file commit is to delete the individual rollback journals and drop the exclusive locks on the database files so that other processes can see the changes. This corresponds to step 3.12 in the single-file commit sequence.

The transaction has already committed at this point so timing is not critical in the deletion of the rollback journals. The current implementation deletes a single rollback journal then unlocks the corresponding database file before proceeding to the next rollback journal. But in the future we might change this so that all rollback journals are deleted before any database files are unlocked. As long as the rollback journal is deleted before its corresponding database file is unlocked it does not matter in what order the rollback journals are deleted or the database files are unlocked.

6.0 Additional Details Of The Commit Process

Section 3.0 above provides an overview of how atomic commit works in SQLite. But it glosses over a number of important details. The following subsections will attempt to fill in the gaps.

6.1 Always Journal Complete Sectors

When the original content of a database page is written into the rollback journal (as shown in section 3.5), SQLite always writes a complete sectors worth of data, even if the page size of the database is smaller than the sector size. Historically, the sector size in SQLite has been hard coded to 512 bytes and since the minimum page size is also 512 bytes, this has never been an issue. But beginning with SQLite version 3.3.14, it is possible for SQLite to use mass storage devices with a sector size larger than 512 bytes. So, beginning with version 3.3.14, whenever any page within a sector is written into the journal file, all pages in that same sector are stored with it.

It is important to store all pages of a sector in the rollback journal in order to prevent database corruption following a power loss while writing the sector. Suppose that pages 1, 2, 3, and 4 are all stored in sector 1 and that page 2 is modified. In order to write the changes to page 2, the underlying hardware must also rewrite the content of pages 1, 3, and 4 since the hardware must write the complete sector. If this write operation is interrupted by a power outage, one or more of the pages 1, 3, or 4 might be left with incorrect data. Hence, to avoid lasting corruption to the database, the original content of all of those pages must be contained in the rollback journal.

6.2 Dealing With Garbage Written Into Journal Files

When data is appended to the end of the rollback journal, SQLite normally makes the pessimistic assumption that the file is first extended with invalid "garbage" data and that afterwards the correct data replaces the garbage. In other words, SQLite assumes that the file size is increased first and then afterwards the content is written into the file. If a power failure occurs after the file size has been increased but before the file content has been written, the rollback journal can be left containing garbage data. If after power is restored, another SQLite process sees the rollback journal containing the garbage data and tries to roll it back into the original database file, it might copy some of the garbage into the database file and thus corrupt the database file.

SQLite uses two defenses against this problem. In the first place, SQLite records the number of pages in the rollback journal in the header of the rollback journal. This number is initially zero. So during an attempt to rollback an incomplete (and possibly corrupt) rollback journal, the process doing the rollback will see that the journal contains zero pages and will thus make no changes to the database. Prior to a commit, the rollback journal is flushed to disk to ensure that all content has been synced to disk and there is no "garbage" left in the file, and only then is the page count in the header changed from zero to true number of pages in the rollback journal. The rollback journal header is always kept in a separate sector from any page data so that it can be overwritten and flushed without risking damage to a data page if a power outage occurs. Notice that the rollback journal is flushed to disk twice: once to write the page content and a second time to write the page count in the header.

The previous paragraph describes what happens when the synchronous pragma setting is "full".

> PRAGMA synchronous=FULL;

The default synchronous setting is full so the above is what usually happens. However, if the synchronous setting is lowered to "normal", SQLite only flushes the rollback journal once, after the page count has been written. This carries a risk of corruption because it might happen that the modified (non-zero) page count reaches the disk surface before all of the data does. The data will have been written first, but SQLite assumes that the underlying filesystem can reorder write requests and that the page count can be burned into oxide first even though its write request occurred last. So as a second line of defense, SQLite also uses a 32-bit checksum on every page of data in the rollback journal. This checksum is evaluated for each page during rollback while rolling back a journal as described in section 4.4. If an incorrect checksum is seen, the rollback is abandoned. Note that the checksum does not guarantee that the page data is correct since there is a small but finite probability that the checksum might be right even if the data is corrupt. But the checksum does at least make such an error unlikely.

Note that the checksums in the rollback journal are not necessary if the synchronous setting is FULL. We only depend on the checksums when synchronous is lowered to NORMAL. Nevertheless, the checksums never hurt and so they are included in the rollback journal regardless of the synchronous setting.

6.3 Cache Spill Prior To Commit

The commit process shown in section 3.0 assumes that all database changes fit in memory until it is time to commit. This is the common case. But sometimes a larger change will overflow the user-space cache prior to transaction commit. In those cases, the cache must spill to the database before the transaction is complete.

At the beginning of a cache spill, the status of the database connection is as shown in step 3.6. Original page content has been saved in the rollback journal and modifications of the pages exist in user memory. To spill the cache, SQLite executes steps 3.7 through 3.9. In other words, the rollback journal is flushed to disk, an exclusive lock is acquired, and changes are written into the database. But the remaining steps are deferred until the transaction really commits. A new journal header is appended to the end of the rollback journal (in its own sector) and the exclusive database lock is retained, but otherwise processing returns to step 3.6. When the transaction commits, or if another cache spill occurs, steps 3.7 and 3.9 are repeated. (Step 3.8 is omitted on second and subsequent passes since an exclusive database lock is already held due to the first pass.)

A cache spill causes the lock on the database file to escalate from reserved to exclusive. This reduces concurrency. A cache spill also causes extra disk flush or fsync operations to occur and these operations are slow, hence a cache spill can seriously reduce performance. For these reasons a cache spill is avoided whenever possible.

7.0 Optimizations

Profiling indicates that for most systems and in most circumstances SQLite spends most of its time doing disk I/O. It follows then that anything we can do to reduce the amount of disk I/O will likely have a large positive impact on the performance of SQLite. This section describes some of the techniques used by SQLite to try to reduce the amount of disk I/O to a minimum while still preserving atomic commit.

7.1 Cache Retained Between Transactions

Step 3.12 of the commit process shows that once the shared lock has been released, all user-space cache images of database content must be discarded. This is done because without a shared lock, other processes are free to modify the database file content and so any user-space image of that content might become obsolete. Consequently, each new transaction would begin by rereading data which had previously been read. This is not as bad as it sounds at first since the data being read is still likely in the operating systems file cache. So the "read" is really just a copy of data from kernel space into user space. But even so, it still takes time.

Beginning with SQLite version 3.3.14 a mechanism has been added to try to reduce the needless rereading of data. In newer versions of SQLite, the data in the user-space pager cache is retained when the lock on the database file is released. Later, after the shared lock is acquired at the beginning of the next transaction, SQLite checks to see if any other process has modified the database file. If the database has been changed in any way since the lock was last released, the user-space cache is erased at that point. But commonly the database file is unchanged and the user-space cache can be retained, and some unnecessary read operations can be avoided.

In order to determine whether or not the database file has changed, SQLite uses a counter in the database header (in bytes 24 through 27) which is incremented during every change operation. SQLite saves a copy of this counter prior to releasing its database lock. Then after acquiring the next database lock it compares the saved counter value against the current counter value and erases the cache if the values are different, or reuses the cache if they are the same.

7.2 Exclusive Access Mode

SQLite version 3.3.14 adds the concept of "Exclusive Access Mode". In exclusive access mode, SQLite retains the exclusive database lock at the conclusion of each transaction. This prevents other processes from accessing the database, but in many deployments only a single process is using a database so this is not a serious problem. The advantage of exclusive access mode is that disk I/O can be reduced in three ways:

1. It is not necessary to increment the change counter in the database header for transactions after the first transaction. This will often save a write of page one to both the rollback journal and the main database file.

2. No other processes can change the database so there is never a need to check the change counter and clear the user-space cache at the beginning of a transaction.

3. Each transaction can be committed by overwriting the rollback journal header with zeros rather than deleting the journal file. This avoids having to modify the directory entry for the journal file and it avoids having to deallocate disk sectors associated with the journal. Furthermore, the next transaction will overwrite existing journal file content rather than append new content and on most systems overwriting is much faster than appending.

The third optimization, zeroing the journal file header rather than deleting the rollback journal file, does not depend on holding an exclusive lock at all times. This optimization can be set independently of exclusive lock mode using the journal_mode pragma as described in section 7.6 below.

7.3 Do Not Journal Freelist Pages

When information is deleted from an SQLite database, the pages used to hold the deleted information are added to a "freelist". Subsequent inserts will draw pages off of this freelist rather than expanding the database file.

Some freelist pages contain critical data; specifically the locations of other freelist pages. But most freelist pages contain nothing useful. These latter freelist pages are called "leaf" pages. We are free to modify the content of a leaf freelist page in the database without changing the meaning of the database in any way.

Because the content of leaf freelist pages is unimportant, SQLite avoids storing leaf freelist page content in the rollback journal in step 3.5 of the commit process. If a leaf freelist page is changed and that change does not get rolled back during a transaction recovery, the database is not harmed by the omission. Similarly, the content of a new freelist page is never written back into the database at step 3.9 nor read from the database at step 3.3. These optimizations can greatly reduce the amount of I/O that occurs when making changes to a database file that contains free space.

7.4 Single Page Updates And Atomic Sector Writes

Beginning in SQLite version 3.5.0, the new Virtual File System (VFS) interface contains a method named xDeviceCharacteristics which reports on special properties that the underlying mass storage device might have. Among the special properties that xDeviceCharacteristics might report is the ability of to do an atomic sector write.

Recall that by default SQLite assumes that sector writes are linear but not atomic. A linear write starts at one end of the sector and changes information byte by byte until it gets to the other end of the sector. If a power loss occurs in the middle of a linear write then part of the sector might be modified while the other end is unchanged. In an atomic sector write, either the entire sector is overwritten or else nothing in the sector is changed.

We believe that most modern disk drives implement atomic sector writes. When power is lost, the drive uses energy stored in capacitors and/or the angular momentum of the disk platter to provide power to complete any operation in progress. Nevertheless, there are so many layers in between the write system call and the on-board disk drive electronics that we take the safe approach in both Unix and w32 VFS implementations and assume that sector writes are not atomic. On the other hand, device manufacturers with more control over their filesystems might want to consider enabling the atomic write property of xDeviceCharacteristics if their hardware really does do atomic writes.

When sector writes are atomic and the page size of a database is the same as a sector size, and when there is a database change that only touches a single database page, then SQLite skips the whole journaling and syncing process and simply writes the modified page directly into the database file. The change counter in the first page of the database file is modified separately since no harm is done if power is lost before the change counter can be updated.

7.5 Filesystems With Safe Append Semantics

Another optimization introduced in SQLite version 3.5.0 makes use of "safe append" behavior of the underlying disk. Recall that SQLite assumes that when data is appended to a file (specifically to the rollback journal) that the size of the file is increased first and that the content is written second. So if power is lost after the file size is increased but before the content is written, the file is left containing invalid "garbage" data. The xDeviceCharacteristics method of the VFS might, however, indicate that the filesystem implements "safe append" semantics. This means that the content is written before the file size is increased so that it is impossible for garbage to be introduced into the rollback journal by a power loss or system crash.

When safe append semantics are indicated for a filesystem, SQLite always stores the special value of -1 for the page count in the header of the rollback journal. The -1 page count value tells any process attempting to rollback the journal that the number of pages in the journal should be computed from the journal size. This -1 value is never changed. So that when a commit occurs, we save a single flush operation and a sector write of the first page of the journal file. Furthermore, when a cache spill occurs we no longer need to append a new journal header to the end of the journal; we can simply continue appending new pages to the end of the existing journal.

7.6 Persistent Rollback Journals

Deleting a file is an expensive operation on many systems. So as an optimization, SQLite can be configured to avoid the delete operation of section 3.11. Instead of deleting the journal file in order to commit a transaction, the file is either truncated to zero bytes in length or its header is overwritten with zeros. Truncating the file to zero length saves having to make modifications to the directory containing the file since the file is not removed from the directory. Overwriting the header has the additional savings of not having to update the length of the file (in the "inode" on many systems) and not having to deal with newly freed disk sectors. Furthermore, at the next transaction the journal will be created by overwriting existing content rather than appending new content onto the end of a file, and overwriting is often much faster than appending.

SQLite can be configured to commit transactions by overwriting the journal header with zeros instead of deleting the journal file by setting the "PERSIST" journaling mode using the journal_mode PRAGMA. For example:

```
PRAGMA journal_mode=PERSIST;
```

The use of persistent journal mode provide a noticeable performance improvement on many systems. Of course, the drawback is that the journal files remain on the disk, using disk space and cluttering directories, long after the transaction commits. The only safe way to delete a persistent journal file is to commit a transaction with journaling mode set to DELETE:

```
PRAGMA journal_mode=DELETE;
BEGIN EXCLUSIVE;
COMMIT;
```

Beware of deleting persistent journal files by any other means since the journal file might be hot, in which case deleting it will corrupt the corresponding database file.

Beginning in SQLite version 3.6.4, the TRUNCATE journal mode is also supported:

```
PRAGMA journal_mode=TRUNCATE;
```

In truncate journal mode, the transaction is committed by truncating the journal file to zero length rather than deleting the journal file (as in DELETE mode) or by zeroing the header (as in PERSIST mode). TRUNCATE mode shares the advantage of PERSIST mode that the directory that contains the journal file and database does not need to be updated. Hence truncating a file is often faster than deleting it. TRUNCATE has the additional advantage that it is not followed by a system call (ex: fsync()) to synchronize the change to disk. It might be safer if it did. But on many modern filesystems, a truncate is an atomic and synchronous operation and so we think that TRUNCATE will usually be safe in the face of power failures. If you are uncertain about whether or not TRUNCATE will be synchronous and atomic on your filesystem and it is important to you that your database survive a power loss or operating system crash that occurs during the truncation operation, then you might consider using a different journaling mode.

On embedded systems with synchronous filesystems, TRUNCATE results in slower behavior than PERSIST. The commit operation is the same speed. But subsequent transactions are slower following a TRUNCATE because it is faster to overwrite existing content than to append to the end of a file. New journal file entries will always be appended following a TRUNCATE but will usually overwrite with PERSIST.

8.0 Testing Atomic Commit Behavior

The developers of SQLite are confident that it is robust in the face of power failures and system crashes because the automatic test procedures do extensive checks on the ability of SQLite to recover from simulated power loss. We call these the "crash tests".

Crash tests in SQLite use a modified VFS that can simulate the kinds of filesystem damage that occur during a power loss or operating system crash. The crash-test VFS can simulate incomplete sector writes, pages filled with garbage data because a write has not completed, and out of order writes, all occurring at varying points during a test scenario. Crash tests execute transactions over and over, varying the time at which a simulated power loss occurs and the properties of the damage inflicted. Each test then reopens the database after the simulated crash and verifies that the transaction either occurred completely or not at all and that the database is in a completely consistent state.

The crash tests in SQLite have discovered a number of very subtle bugs (now fixed) in the recovery mechanism. Some of these bugs were very obscure and unlikely to have been found using only code inspection and analysis techniques. From this experience, the developers of SQLite feel confident that any other database system that does not use a similar crash test system likely contains undetected bugs that will lead to database corruption following a system crash or power failure.

9.0 Things That Can Go Wrong

The atomic commit mechanism in SQLite has proven to be robust, but it can be circumvented by a sufficiently creative adversary or a sufficiently broken operating system implementation. This section describes a few of the ways in which an SQLite database might be corrupted by a power failure or system crash. (See also: How To Corrupt Your Database Files.)

9.1 Broken Locking Implementations

SQLite uses filesystem locks to make sure that only one process and database connection is trying to modify the database at a time. The filesystem locking mechanism is implemented in the VFS layer and is different for every operating system. SQLite depends on this implementation being correct. If something goes wrong and two or more processes are able to write the same database file at the same time, severe damage can result.

We have received reports of implementations of both Windows network filesystems and NFS in which locking was subtly broken. We can not verify these reports, but as locking is difficult to get right on a network filesystem we have no reason to doubt them. You are advised to avoid using SQLite on a network filesystem in the first place, since performance will be slow. But if you must use a network filesystem to store SQLite database files, consider using a secondary locking mechanism to prevent simultaneous writes to the same database even if the native filesystem locking mechanism malfunctions.

The versions of SQLite that come preinstalled on Apple Mac OS X computers contain a version of SQLite that has been extended to use alternative locking strategies that work on all network filesystems that Apple supports. These extensions used by Apple work great as long as all processes are accessing the database file in the same way. Unfortunately, the locking mechanisms do not exclude one another, so if one process is accessing a file using (for example) AFP locking and another process (perhaps on a different machine) is using dot-file locks, the two processes might collide because AFP locks do not exclude dot-file locks or vice versa.

9.2 Incomplete Disk Flushes

SQLite uses the fsync() system call on Unix and the FlushFileBuffers() system call on w32 in order to sync the file system buffers onto disk oxide as shown in step 3.7 and step 3.10. Unfortunately, we have received reports that neither of these interfaces works as advertised on many systems. We hear that FlushFileBuffers() can be completely disabled using registry settings on some Windows versions. Some historical versions of Linux contain versions of fsync() which are no-ops on some filesystems, we are told. Even on systems where FlushFileBuffers() and fsync() are said to be working, often the IDE disk control lies and says that data has reached oxide while it is still held only in the volatile control cache.

On the Mac, you can set this pragma:

> PRAGMA fullfsync=ON;

Setting fullfsync on a Mac will guarantee that data really does get pushed out to the disk platter on a flush. But the implementation of fullfsync involves resetting the disk controller. And so not only is it profoundly slow, it also slows down other unrelated disk I/O. So its use is not recommended.

9.3 Partial File Deletions

SQLite assumes that file deletion is an atomic operation from the point of view of a user process. If power fails in the middle of a file deletion, then after power is restored SQLite expects to see either the entire file with all of its original data intact, or it expects not to find the file at all. Transactions may not be atomic on systems that do not work this way.

9.4 Garbage Written Into Files

SQLite database files are ordinary disk files that can be opened and written by ordinary user processes. A rogue process can open an SQLite database and fill it with corrupt data. Corrupt data might also be introduced into an SQLite database by bugs in the operating system or disk controller; especially bugs triggered by a power failure. There is nothing SQLite can do to defend against these kinds of problems.

9.5 Deleting Or Renaming A Hot Journal

If a crash or power loss does occur and a hot journal is left on the disk, it is essential that the original database file and the hot journal remain on disk with their original names until the database file is opened by another SQLite process and rolled back. During recovery at step 4.2 SQLite locates the hot journal by looking for a file in the same directory as the database being opened and whose name is derived from the name of the file being opened. If either the original database file or the hot journal have been moved or renamed, then the hot journal will not be seen and the database will not be rolled back.

We suspect that a common failure mode for SQLite recovery happens like this: A power failure occurs. After power is restored, a well-meaning user or system administrator begins looking around on the disk for damage. They see their database file named "important.data". This file is perhaps familiar to them. But after the crash, there is also a hot journal named "important.data-journal". The user then deletes the hot journal, thinking that they are helping to cleanup the system. We know of no way to prevent this other than user education.

If there are multiple (hard or symbolic) links to a database file, the journal will be created using the name of the link through which the file was opened. If a crash occurs and the database is opened again using a different link, the hot journal will not be located and no rollback will occur.

Sometimes a power failure will cause a filesystem to be corrupted such that recently changed filenames are forgotten and the file is moved into a "/lost+found" directory. When that happens, the hot journal will not be found and recovery will not occur. SQLite tries to prevent this by opening and syncing the directory containing the rollback journal at the same time it syncs the journal file itself. However, the movement of files into /lost+found can be caused by unrelated processes creating unrelated files in the same directory as the main database file. And since this is out from under the control of SQLite, there is nothing that SQLite can do to prevent it. If you are running on a system that is vulnerable to this kind of filesystem namespace corruption (most modern journalling filesystems are immune, we believe) then you might want to consider putting each SQLite database file in its own private subdirectory.

10.0 Future Directions And Conclusion

Every now and then someone discovers a new failure mode for the atomic commit mechanism in SQLite and the developers have to put in a patch. This is happening less and less and the failure modes are becoming more and more obscure. But it would still be foolish to suppose that the atomic commit logic of SQLite is entirely bug-free. The developers are committed to fixing these bugs as quickly as they might be found.

The developers are also on the lookout for new ways to optimize the commit mechanism. The current VFS implementations for Unix (Linux and Mac OS X) and Windows make pessimistic assumptions about the behavior of those systems. After consultation with experts on how these systems work, we might be able to relax some of the assumptions on these systems and allow them to run faster. In particular, we suspect that most modern filesystems exhibit the safe append property and that many of them might support atomic sector writes. But until this is known for certain, SQLite will take the conservative approach and assume the worst.

Dynamic Memory Allocation In SQLite

SQLite uses dynamic memory allocation to obtain memory for storing various objects (ex: database connections and prepared statements) and to build a memory cache of the database file and to hold the results of queries. Much effort has gone into making the dynamic memory allocation subsystem of SQLite reliable, predictable, robust, secure, and efficient.

This document provides an overview of dynamic memory allocation within SQLite. The target audience is software engineers who are tuning their use of SQLite for peak performance in demanding environments. Nothing in this document is required knowledge for using SQLite. The default settings and configuration for SQLite will work well in most applications. However, the information contained in this document may be useful to engineers who are tuning SQLite to comply with special requirements or to run under unusual circumstances.

1.0 Features

The SQLite core and its memory allocation subsystem provides the following capabilities:

- **Robust against allocation failures.** If a memory allocation ever fails (that is to say, if malloc() or realloc() ever return NULL) then SQLite will recover gracefully. SQLite will first attempt to free memory from unpinned cache pages then retry the allocation request. Failing that, SQLite will either stop what it is doing and return the SQLITE_NOMEM error code back up to the application or it will make do without the requested memory.
- **No memory leaks.** The application is responsible for destroying any objects it allocates. (For example, the application must use sqlite3_finalize() on every prepared statement and sqlite3_close() on every database connection.) But as long as the application cooperates, SQLite will never leak memory. This is true even in the face of memory allocation failures or other system errors.
- **Memory usage limits.** The sqlite3_soft_heap_limit64() mechanism allows the application to set a memory usage limit that SQLite strives to stay below. SQLite will attempt to reuse memory from its caches rather than allocating new memory as it approaches the soft limit.

- **Zero-malloc option.** The application can optionally provide SQLite with several buffers of bulk memory at startup and SQLite will then use those provided buffers for all of its memory allocation needs and never call system malloc() or free().
- **Application-supplied memory allocators.** The application can provide SQLite with pointers to alternative memory allocators at start-time. The alternative memory allocator will be used in place of system malloc() and free().
- **Proof against breakdown and fragmentation.** SQLite can be configured so that, subject to certain usage constraints detailed below, it is guaranteed to never fail a memory allocation or fragment the heap. This property is important to long-running, high-reliability embedded systems where a memory allocation error could contribute to an overall system failure.
- **Memory usage statistics.** Applications can see how much memory they are using and detect when memory usage is approaching or exceeding design boundaries.
- **Plays well with memory debuggers.** Memory allocation in SQLite is structured so that standard third-party memory debuggers (such as dmalloc or valgrind) can be used to verify correct memory allocation behavior.
- **Minimal calls to the allocator.** The system malloc() and free() implementations are inefficient on many systems. SQLite strives to reduce overall processing time by minimizing its use of malloc() and free().
- **Open access.** Pluggable SQLite extensions or even the application itself can access to the same underlying memory allocation routines used by SQLite through the sqlite3_malloc(), sqlite3_realloc(), and sqlite3_free() interfaces.

2.0 Testing

Most of the code in the SQLite source tree is devoted purely to testing and verification. Reliability is important to SQLite. Among the tasks of the test infrastructure is to ensure that SQLite does not misuse dynamically allocated memory, that SQLite does not leak memory, and that SQLite responds correctly to a dynamic memory allocation failure.

The test infrastructure verifies that SQLite does not misuse dynamically allocated memory by using a specially instrumented memory allocator. The instrumented memory allocator is enabled at compile-time using the SQLITE_MEMDEBUG option. The instrumented memory allocator is much slower than the default memory allocator and so its use is not recommended in production. But when enabled during testing, the instrumented memory allocator performs the following checks:

- **Bounds checking.** The instrumented memory allocator places sentinel values at both ends of each memory allocation to verify that nothing within SQLite writes outside the bounds of the allocation.
- **Use of memory after freeing.** When each block of memory is freed, every byte is overwritten with a nonsense bit pattern. This helps to ensure that no memory is ever used after having been freed.
- **Freeing memory not obtained from malloc.** Each memory allocation from the instrumented memory allocator contains sentinels used to verify that every allocation freed came from prior malloc.
- **Uninitialized memory.** The instrumented memory allocator initializes each memory allocation to a nonsense bit pattern to help ensure that the user makes no assumptions about the content of allocation memory.

Regardless of whether or not the instrumented memory allocator is used, SQLite keeps track of how much memory is currently checked out. There are hundreds of test scripts used for testing SQLite. At the end of each script, all objects are destroyed and a test is made to ensure that all memory has been freed. This is how memory leaks are detected. Notice that memory leak detection is in force at all times, during test builds and during production builds. Whenever one of the developers runs any individual test script, memory leak detection is active. Hence memory leaks that do arise during development are quickly detected and fixed.

The response of SQLite to out-of-memory (OOM) errors is tested using a specialized memory allocator overlay that can simulate memory failures. The overlay is a layer that is inserted in between the memory allocator and the rest of SQLite. The overlay passes most memory allocation requests straight through to the underlying allocator and passes the results back up to the requester. But the overlay can be set to cause the Nth memory allocation to fail. To run an OOM test, the overlay is first set to fail on the first allocation attempt. Then some test script is run and verification that the allocation was correctly caught and handled is made. Then the overlay is set to fail on the second allocation and the test repeats. The failure point continues to advance one allocation at a time until the entire test procedure runs to completion without hitting a memory allocation error. This whole test sequence run twice. On the first pass, the overlay is set to fail only the Nth allocation. On the second pass, the overlay is set to fail the Nth and all subsequent allocations.

Note that the memory leak detection logic continues to work even when the OOM overlay is being used. This verifies that SQLite does not leak memory even when it encounters memory allocation errors. Note also that the OOM overlay can work with any underlying memory allocator, including the instrumented memory allocator that checks for memory allocation misuse. In this way it is verified that OOM errors do not induce other kinds of memory usage errors.

Finally, we observe that the instrumented memory allocator and the memory leak detector both work over the entire SQLite test suite and the TCL test suite provides over 99% statement test coverage and that the TH3 test harness provides 100% branch test coverage with no leak leaks. This is strong evidence that dynamic memory allocation is used correctly everywhere within SQLite.

2.1 Use of reallocarray()

The reallocarray() interface is a recent innovation (circa 2014) from the OpenBSD community that grow out of efforts to prevent the next "heartbleed" bug by avoiding 32-bit integer arithmetic overflow on memory allocation size computations. The reallocarray() function has both unit-size and count parameters. To allocate memory sufficient to hold an array of N elements each X-bytes in size, one calls "reallocarray(0,X,N)". This is preferred over the traditional technique of invoking "malloc(X*N)" as reallocarray() eliminates the risk that the X*N multiplication will overflow and cause malloc() to return a buffer that is a different size from what the application expected.

SQLite does not use reallocarray(). The reason is that reallocarray() is not useful to SQLite. It turns out that SQLite never does memory allocations that are the simple product of two integers. Instead, SQLite does allocations of the form "X+C" or "N*X+C" or "M*N*X+C" or "N*X+M*Y+C", and so forth. The reallocarray() interface is not helpful in avoiding integer overflow in those cases.

Nevertheless, integer overflow in the computation of memory allocation sizes is a concern that SQLite would like to deal with. To prevent problems, all SQLite internal memory allocations occur using thin wrapper functions that take a signed 64-bit integer size parameter. The SQLite source code is audited to ensure that all size computations are carried out using 64-bit signed integers as well. SQLite will refuse to allocate more than about 2GB of memory at one go. (In common use, SQLite seldom ever allocates more than about 8KB of memory at a time so a 2GB allocation limit is not a burden.) So the 64-bit size parameter provides lots of headroom for detecting overflows. The same audit that verifies that all size computations are done as 64-bit signed integers also verifies that it is impossible to overflow a 64-bit integer during the computation.

The code audits used to ensure that memory allocation size computations do not overflow in SQLite are repeated prior to every SQLite release.

3.0 Configuration

The default memory allocation settings in SQLite are appropriate for most applications. However, applications with unusual or particularly strict requirements may want to adjust the configuration to more closely align SQLite to their needs. Both compile-time and start-time configuration options are available.

3.1 Alternative low-level memory allocators

The SQLite source code includes several different memory allocation modules that can be selected at compile-time, or to a limited extent at start-time.

3.1.1 The default memory allocator

By default, SQLite uses the malloc(), realloc(), and free() routines from the standard C library for its memory allocation needs. These routines are surrounded by a thin wrapper that also provides a "memsize()" function that will return the size of an existing allocation. The memsize() function is needed to keep an accurate count of the number of bytes of outstanding memory; memsize() determines how many bytes to remove from the outstanding count when an allocation is freed. The default allocator implements memsize() by always allocating 8 extra bytes on each malloc() request and storing the size of the allocation in that 8-byte header.

The default memory allocator is recommended for most applications. If you do not have a compelling need to use an alternative memory allocator, then use the default.

3.1.2 The debugging memory allocator

If SQLite is compiled with the SQLITE_MEMDEBUG compile-time option, then a different, heavy wrapper is used around system malloc(), realloc(), and free(). The heavy wrapper allocates around 100 bytes of extra space with each allocation. The extra space is used to place sentinel values at both ends of the allocation returned to the SQLite core. When an allocation is freed, these sentinels are checked to make sure the SQLite core did not overrun the buffer in either direction. When the system library is GLIBC, the heavy wrapper also makes use of the GNU backtrace() function to examine the stack and record the ancestor functions of the malloc() call. When running the SQLite test suite, the heavy wrapper also records the name of the current test case. These latter two features are useful for tracking down the source of memory leaks detected by the test suite.

The heavy wrapper that is used when SQLITE_MEMDEBUG is set also makes sure each new allocation is filled with nonsense data prior to returning the allocation to the caller. And as soon as an allocation is free, it is again filled with nonsense data. These two actions help to ensure that the SQLite core does not make assumptions about the state of newly allocated memory and that memory allocations are not used after they have been freed.

The heavy wrapper employed by SQLITE_MEMDEBUG is intended for use only during testing, analysis, and debugging of SQLite. The heavy wrapper has a significant performance and memory overhead and probably should not be used in production.

3.1.3 The Win32 native memory allocator

If SQLite is compiled for Windows with the SQLITE_WIN32_MALLOC compile-time option, then a different, thin wrapper is used around HeapAlloc(), HeapReAlloc(), and HeapFree(). The thin wrapper uses the configured SQLite heap, which will be different from the default process heap if the SQLITE_WIN32_HEAP_CREATE compile-time option is used. In addition, when an allocation is made or freed, HeapValidate() will be called if SQLite is compiled with assert() enabled and the SQLITE_WIN32_MALLOC_VALIDATE compile-time option.

3.1.4 Zero-malloc memory allocator

When SQLite is compiled with the SQLITE_ENABLE_MEMSYS5 option, an alternative memory allocator that does not use malloc() is included in the build. The SQLite developers refer to this alternative memory allocator as "memsys5". Even when it is included in the build, memsys5 is disabled by default. To enable memsys5, the application must invoke the following SQLite interface at start-time:

```
sqlite3_config(SQLITE_CONFIG_HEAP, pBuf, szBuf, mnReq);
```

In the call above, pBuf is a pointer to a large, contiguous chunk of memory space that SQLite will use to satisfy all of its memory allocation needs. pBuf might point to a static array or it might be memory obtained from some other application-specific mechanism. szBuf is an integer that is the number of bytes of memory space pointed to by pBuf. mnReq is another integer that is the minimum size of an allocation. Any call to sqlite3_malloc(N) where N is less than mnReq will be rounded up to mnReq. mnReq must be a power of two. We shall see later that the mnReq parameter is important in reducing the value of **n** and hence the minimum memory size requirement in the Robson proof.

The memsys5 allocator is designed for use on embedded systems, though there is nothing to prevent its use on workstations. The szBuf is typically between a few hundred kilobytes up to a few dozen megabytes, depending on system requirements and memory budget.

The algorithm used by memsys5 can be called "power-of-two, first-fit". The sizes of all memory allocation requests are rounded up to a power of two and the request is satisfied by the first free slot in pBuf that is large enough. Adjacent freed allocations are coalesced using a buddy system. When used appropriately, this algorithm provides mathematical guarantees against fragmentation and breakdown, as described further below.

3.1.5 Experimental memory allocators

The name "memsys5" used for the zero-malloc memory allocator implies that there are several additional memory allocators available, and indeed there are. The default memory allocator is "memsys1". The debugging memory allocator is "memsys2". Those have already been covered.

If SQLite is compiled with SQLITE_ENABLE_MEMSYS3 then another zero-malloc memory allocator, similar to memsys5, is included in the source tree. The memsys3 allocator, like memsys5, must be activated by a call to sqlite3_config(SQLITE_CONFIG_HEAP,...). Memsys3 uses the memory buffer supplied as its source for all memory allocations. The difference between memsys3 and memsys5 is that memsys3 uses a different memory allocation algorithm that seems to work well in practice, but which does not provide mathematical guarantees against memory fragmentation and breakdown. Memsys3 was a predecessor to memsys5. The SQLite developers now believe that memsys5 is superior to memsys3 and that all applications that need a zero-malloc memory allocator should use memsys5 in preference to memsys3. Memsys3 is considered both experimental and deprecated and will likely be removed from the source tree in a future release of SQLite.

Code for memsys4 is still in the SQLite source tree (as of this writing - SQLite version 3.6.1), but it has not been maintained for several release cycles and probably does not work. (Update: memsys4 was removed as of version 3.6.5) Memsys4 was an attempt to use mmap() to obtain memory and then use madvise() to release unused pages back to the operating system so that they could be reused by other processes. The work on memsys4 has been abandoned and the memsys4 module will likely be removed from the source tree in the near future.

Memsys6 uses system malloc() and free() to obtain the memory it needs. Memsys6 serves as an aggregator. Memsys6 only calls system malloc() to obtain large allocations. It then subdivides those large allocations to services multiple smaller memory allocation requests coming from the SQLite core. Memsys6 is intended for use in systems where system malloc() is particularly inefficient. The idea behind memsys6 is to reduce the number of calls to system malloc() by a factor of 10 or more.

Memsys6 is made available by compiling SQLite with the SQLITE_ENABLE_MEMSYS6 compile-time option and then at start-time invoking:

```
sqlite3_config(SQLITE_CONFIG_CHUNKALLOC);
```

Memsys6 was added in SQLite version 3.6.1. It is very experimental. Its future is uncertain and it may be removed in a subsequent release. Update: Memsys6 was removed as of version 3.6.5.

Other experimental memory allocators might be added in future releases of SQLite. One may anticipate that these will be called memsys7, memsys8, and so forth.

3.1.6 Application-defined memory allocators

New memory allocators do not have to be part of the SQLite source tree nor included in the sqlite3.c amalgamation. Individual applications can supply their own memory allocators to SQLite at start-time.

To cause SQLite to use a new memory allocator, the application simply calls:

```
sqlite3_config(SQLITE_CONFIG_MALLOC, pMem);
```

In the call above, pMem is a pointer to an sqlite3_mem_methods object that defines the interface to the application-specific memory allocator. The sqlite3_mem_methods object is really just a structure containing pointers to functions to implement the various memory allocation primitives.

In a multi-threaded application, access to the sqlite3_mem_methods is serialized if and only if SQLITE_CONFIG_MEMSTATUS is enabled. If SQLITE_CONFIG_MEMSTATUS is disabled then the methods in sqlite3_mem_methods must take care of their own serialization needs.

3.1.7 Memory allocator overlays

An application can insert layers or "overlays" in between the SQLite core and the underlying memory allocator. For example, the out-of-memory test logic for SQLite uses an overlay that can simulate memory allocation failures.

An overlay can be created by using the

```
sqlite3_config(SQLITE_CONFIG_GETMALLOC, pOldMem);
```

interface to obtain pointers to the existing memory allocator. The existing allocator is saved by the overlay and is used as a fallback to do real memory allocation. Then the overlay is inserted in place of the existing memory allocator using the sqlite3_config(SQLITE_CONFIG_MALLOC,...) as described above.

3.1.8 No-op memory allocator stub

If SQLite is compiled with the SQLITE_ZERO_MALLOC option, then the default memory allocator is omitted and replaced by a stub memory allocator that never allocates any memory. Any calls to the stub memory allocator will report back that no memory is available.

The no-op memory allocator is not useful by itself. It exists only as a placeholder so that SQLite has a memory allocator to link against on systems that may not have malloc(), free(), or realloc() in their standard library. An application that is compiled with SQLITE_ZERO_MALLOC will need to use sqlite3_config() together with SQLITE_CONFIG_MALLOC or SQLITE_CONFIG_HEAP to specify a new alternative memory allocator before beginning to use SQLite.

3.2 Scratch memory

SQLite occasionally needs a large chunk of "scratch" memory to perform some transient calculation. Scratch memory is used, for example, as temporary storage when rebalancing a B-Tree. These scratch memory allocations are typically about 10 kilobytes in size and are transient - lasting only for the duration of a single, short-lived function call.

In older versions of SQLite, the scratch memory was obtained from the processor stack. That works great on workstations with a large stack. But pulling large buffers from the stack caused problems on embedded systems with a small processor stack (typically 4K or 8K). And so SQLite was modified to allocate scratch memory from the heap.

However, doing occasional large transient allocations from the heap can lead to memory fragmentation in embedded systems. To work around this problem, a separate memory allocation system for scratch memory has been created.

The scratch memory allocator is set up as follows:

```
sqlite3_config(SQLITE_CONFIG_SCRATCH, pBuf, sz, N);
```

The pBuf parameter is a pointer to a contiguous range of bytes that SQLite will use for all scratch memory allocations. The buffer must be at least sz*N bytes in size. The "sz" parameter is the maximum size of each scratch allocation. N is the maximum number of simultaneous scratch allocations. The "sz" parameter should be approximately 6 times the maximum database page size. N should be twice the number of threads running in the system. No single thread will ever request more than two scratch allocation at a time so if there are never more than N threads, then there will always be enough scratch memory available.

If the scratch memory setup does not define enough memory, then SQLite falls back to using the regular memory allocator for its scratch memory allocations. The default setup is sz=0 and N=0 so the use of the regular memory allocator is the default behavior.

3.3 Page cache memory

In most applications, the database page cache subsystem within SQLite uses more dynamically allocated memory than all other parts of SQLite combined. It is not unusual to see the database page cache consumes over 10 times more memory than the rest of SQLite combined.

SQLite can be configured to make page cache memory allocations from a separate and distinct memory pool of fixed-size slots. This can have two advantages:

- Because allocations are all the same size, the memory allocator can operate much faster. The allocator need not bother with coalescing adjacent free slots or searching for a slot of an appropriate size. All unallocated memory slots can be stored on a linked list. Allocating consists of removing the first entry from the list. Deallocating is simply adding an entry to the beginning of the list.
- With a single allocation size, the **n** parameter in the Robson proof is 1, and the total memory space required by the allocator (**N**) is exactly equal to maximum memory used (**M**). No additional memory is required to cover fragmentation overhead, thus reducing memory requirements. This is particularly important for the page cache memory since the page cache constitutes the largest component of the memory needs of SQLite.

The page-cache memory allocator is disabled by default. An application can enable it at start-time as follows:

```
sqlite3_config(SQLITE_CONFIG_PAGECACHE, pBuf, sz, N);
```

The pBuf parameter is a pointer to a contiguous range of bytes that SQLite will use for page-cache memory allocations. The buffer must be at least sz*N bytes in size. The "sz" parameter is the size of each page-cache allocation. N is the maximum number of available allocations.

If SQLite needs a page-cache entry that is larger than "sz" bytes or if it needs more than N entries, it falls back to using the general-purpose memory allocator.

3.4 Lookaside memory allocator

SQLite database connections make many small and short-lived memory allocations. This occurs most commonly when compiling SQL statements using sqlite3_prepare_v2() but also to a lesser extent when running prepared statements using sqlite3_step(). These small memory allocations are used to hold things such as the names of tables and columns, parse tree nodes, individual query results values, and B-Tree cursor objects. There are consequently many calls to malloc() and free() - so many calls that malloc() and free() end up using a significant fraction of the CPU time assigned to SQLite.

SQLite version 3.6.1 introduced the lookaside memory allocator to help reduce the memory allocation load. In the lookaside allocator, each database connection preallocates a single large chunk of memory (typically in the range of 50 to 100 kilobytes) and divides that chunk up into small fixed-size "slots" of around 50 to 200 byte each. This becomes the lookaside memory pool. Thereafter, memory allocations associated with the database connection and that are not too larger are satisfied using one of the lookaside pool slots rather than by calling the general-purpose memory allocator. Larger allocations continue to use the general-purpose memory allocator, as do allocations that occur when the lookaside pool slots are all checked out. But in many cases, the memory allocations are small enough and there are few enough outstanding that the new memory requests can be satisfied from the lookaside pool.

Because lookaside allocations are always the same size, the allocation and deallocation algorithms are very quick. There is no need to coalesce adjacent free slots or search for a slot of a particular size. Each database connection maintains a singly-linked list of unused slots. Allocation requests simply pull the first element of this list. Deallocations simply push the element back onto the front of the list. Furthermore, each database connection is assumed to already be running in a single thread (there are mutexes already in place to enforce this) so no additional mutexing is required to serialize access to the lookaside slot freelist. Consequently, lookaside memory allocations and deallocations are very fast. In speed tests on Linux and Mac OS X workstations, SQLite has shown overall performance improvements as high as 10% and 15%, depending on the workload how lookaside is configured.

The size of the lookaside memory pool has a global default value but can also be configured on a connection-by-connection basis. To change the default size of the lookaside memory pool use the following interface at start-time:

```
sqlite3_config(SQLITE_CONFIG_LOOKASIDE, sz, cnt);
```

The "sz" parameter is the size in bytes of each lookaside slot. The default is 100 bytes. The "cnt" parameter is the total number of lookaside memory slots per database connection. The default value is 500 slots. The total amount of lookaside memory allocated to each database connection is sz*cnt bytes. Hence the lookaside memory pool allocated per database connection is 50 kilobytes by default. (Note: these default values are for SQLite version 3.6.1 and are subject to changes in future releases.)

The lookaside pool can be changed for an individual database connection "db" using this call:

```
sqlite3_db_config(db, SQLITE_DBCONFIG_LOOKASIDE, pBuf, sz, cnt);
```

The "pBuf" parameter is a pointer to memory space that will be used for the lookaside memory pool. If pBuf is NULL, then SQLite will obtain its own space for the memory pool using sqlite3_malloc(). The "sz" and "cnt" parameters are the size of each lookaside slot and the number of slots, respectively. If pBuf is not NULL, then it must point to at least sz*cnt bytes of memory.

The lookaside configuration can only be changed while there are no outstanding lookaside allocations for the database connection. Hence, the configuration should be set immediately after creating the database connection using sqlite3_open() (or equivalent) and before evaluating any SQL statements on the connection.

3.5 Memory status

By default, SQLite keeps statistics on its memory usage. These statistics are useful in helping to determine how much memory an application really needs. The statistics can also be used in high-reliability system to determine if the memory usage is coming close to or exceeding the limits of the Robson proof and hence that the memory allocation subsystem is liable to breakdown.

Most memory statistics are global, and therefore the tracking of statistics must be serialized with a mutex. Statistics are turned on by default, but an option exists to disable them. By disabling memory statistics, SQLite avoids entering and leaving a mutex on each memory allocation and deallocation. That savings can be noticeable on systems where mutex operations are expensive. To disable memory statistics, the following interface is used at start-time:

```
sqlite3_config(SQLITE_CONFIG_MEMSTATUS, onoff);
```

The "onoff" parameter is true to enable the tracking of memory statistics and false to disable statistics tracking.

Assuming statistics are enabled, the following routine can be used to access them:

```
sqlite3_status(verb, &current, &highwater, resetflag);
```

The "verb" argument determines what statistic is accessed. There are various verbs defined. The list is expected to grow as the sqlite3_status() interface matures. The current value the selected parameter is written into integer "current" and the highest historical value is written into integer "highwater". If resetflag is true, then the high-water mark is reset down to the current value after the call returns.

A different interface is used to find statistics associated with a single database connection:

```
sqlite3_db_status(db, verb, &current, &highwater, resetflag);
```

This interface is similar except that it takes a pointer to a database connection as its first argument and returns statistics about that one object rather than about the entire SQLite library. The sqlite3_db_status() interface currently only recognizes a single verb SQLITE_DBSTATUS_LOOKASIDE_USED, though additional verbs may be added in the future.

The per-connection statistics do not use global variables and hence do not require mutexes to update or access. Consequently the per-connection statistics continue to function even if SQLITE_CONFIG_MEMSTATUS is turned off.

3.6 Setting memory usage limits

The sqlite3_soft_heap_limit64() interface can be used to set an upper bound on the total amount of outstanding memory that the general-purpose memory allocator for SQLite will allow to be outstanding at one time. If attempts are made to allocate more memory that specified by the soft heap limit, then SQLite will first attempt to free cache memory before continuing with the allocation request. The soft heap limit mechanism only works if memory statistics are enabled and it works best if the SQLite library is compiled with the SQLITE_ENABLE_MEMORY_MANAGEMENT compile-time option.

The soft heap limit is "soft" in this sense: If SQLite is not able to free up enough auxiliary memory to stay below the limit, it goes ahead and allocations the extra memory and exceeds its limit. This occurs under the theory that it is better to use additional memory than to fail outright.

As of SQLite version 3.6.1, the soft heap limit only applies to the general-purpose memory allocator. The soft heap limit does not know about or interact with the scratch memory allocator, the pagecache memory allocator, or the lookaside memory allocator. This deficiency will likely be addressed in a future release.

4.0 Mathematical Guarantees Against Memory Allocation Failures

The problem of dynamic memory allocation, and specifically the problem of a memory allocator breakdown, has been studied by J. M. Robson and the results published as:

> J. M. Robson. "Bounds for Some Functions Concerning Dynamic Storage Allocation". *Journal of the Association for Computing Machinery*, Volume 21, Number 8, July 1974, pages 491-499.

Let us use the following notation (similar but not identical to Robson's notation):

> | **N** | The amount of raw memory needed by the memory allocation system in order to guarantee that no memory allocation will ever fail. | | **M** | The maximum amount of memory that the application ever has checked out at any point in time. | | **n** | The ratio of the largest memory allocation to the smallest. We assume that every memory allocation size is an integer multiple of the smallest memory allocation size. |

Robson proves the following result:

> **N** = **M***(1 + (log<sub>2</sub> **n**)/2) - **n** + 1

Colloquially, the Robson proof shows that in order to guarantee breakdown-free operation, any memory allocator must use a memory pool of size **N** which exceeds the maximum amount of memory ever used **M** by a multiplier that depends on **n**, the ratio of the largest to the smallest allocation size. In other words, unless all memory allocations are of exactly the same size (**n**=1) then the system needs access to more memory than it will ever use at one time. Furthermore, we see that the amount of surplus memory required grows rapidly as the ratio of largest to smallest allocations increases, and so there is strong incentive to keep all allocations as near to the same size as possible.

Robson's proof is constructive. He provides an algorithm for computing a sequence of allocation and deallocation operations that will lead to an allocation failure due to memory fragmentation if available memory is as much as one byte less than **N**. And, Robson shows that a power-of-two first-fit memory allocator (such as implemented by memsys5) will never fail a memory allocation provided that available memory is **N** or more bytes.

The values **M** and **n** are properties of the application. If an application is constructed in such a way that both **M** and **n** are known, or at least have known upper bounds, and if the application uses the memsys5 memory allocator and is provided with **N** bytes of available memory space using SQLITE_CONFIG_HEAP then Robson proves that no memory allocation request will ever fail within the application. To put this another way, the application developer can select a value for **N** that will guarantee that no call to any SQLite interface will ever return SQLITE_NOMEM. The memory pool will never become so fragmented that a new memory allocation request cannot be satisfied. This is an important property for applications where a software fault could cause injury, physical harm, or loss of irreplaceable data.

4.1 Computing and controlling parameters M and n

The Robson proof applies separately to each of the memory allocators used by SQLite:

- The general-purpose memory allocator (memsys5).
- The scratch memory allocator.

- The pagecache memory allocator.
- The lookaside memory allocator.

For allocators other than memsys5, all memory allocations are of the same size. Hence, **n**=1 and therefore **N**=**M**. In other words, the memory pool need be no larger than the largest amount of memory in use at any given moment.

SQLite guarantees that no thread will ever use more than two scratch memory slots at one time. So if an application allocates twice as many scratch memory slots as there are threads, and assuming the size of each slot is large enough, there is never a chance of overflowing the scratch memory allocator. An upper bound on the size of scratch memory allocations is six times the largest page size. It is easy, therefore, to guarantee breakdown-free operation of the scratch memory allocator.

The usage of pagecache memory is somewhat harder to control in SQLite version 3.6.1, though mechanisms are planned for subsequent releases that will make controlling pagecache memory much easier. Prior to the introduction of these new mechanisms, the only way to control pagecache memory is using the cache_size pragma.

Safety-critical applications will usually want to modify the default lookaside memory configuration so that when the initial lookaside memory buffer is allocated during sqlite3_open() the resulting memory allocation is not so large as to force the **n** parameter to be too large. In order to keep **n** under control, it is best to try to keep the largest memory allocation below 2 or 4 kilobytes. Hence, a reasonable default setup for the lookaside memory allocator might any one of the following:

```
sqlite3_config(SQLITE_CONFIG_LOOKASIDE, 32, 32);  /* 1K */
sqlite3_config(SQLITE_CONFIG_LOOKASIDE, 64, 32);  /* 2K */
sqlite3_config(SQLITE_CONFIG_LOOKASIDE, 32, 64);  /* 2K */
sqlite3_config(SQLITE_CONFIG_LOOKASIDE, 64, 64);  /* 4K */
```

Another approach is to initially disable the lookaside memory allocator:

```
sqlite3_config(SQLITE_CONFIG_LOOKASIDE, 0, 0);
```

Then let the application maintain a separate pool of larger lookaside memory buffers that it can distribute to database connections as they are created. In the common case, the application will only have a single database connection and so the lookaside memory pool can consist of a single large buffer.

```
sqlite3_db_config(db, SQLITE_DBCONFIG_LOOKASIDE, aStatic, 256, 500);
```

The lookaside memory allocator is really intended as performance optimization, not as a method for assuring breakdown-free memory allocation, so it is not unreasonable to completely disable the lookaside memory allocator for safety-critical operations.

The general purpose memory allocator is the most difficult memory pool to manage because it supports allocations of varying sizes. Since **n** is a multiplier on **M** we want to keep **n** as small as possible. This argues for keeping the minimum allocation size for memsys5 as large as possible. In most applications, the lookaside memory allocator is able to handle small allocations. So it is reasonable to set the minimum allocation size for memsys5 to 2, 4 or even 8 times the maximum size of a lookaside allocation. A minimum allocation size of 512 is a reasonable setting.

Further to keeping **n** small, one desires to keep the size of the largest memory allocations under control. Large requests to the general-purpose memory allocator might come from several sources:

1. SQL table rows that contain large strings or BLOBs.
2. Complex SQL queries that compile down to large prepared statements.
3. SQL parser objects used internally by sqlite3_prepare_v2().
4. Storage space for database connection objects.
5. Scratch memory allocations that overflow into the general-purpose memory allocator.
6. Page cache memory allocations that overflow into the general-purpose memory allocator.
7. Lookaside buffer allocations for new database connections.

The last three allocations can be controlled and/or eliminated by configuring the scratch memory allocator, pagecache memory allocator, and lookaside memory allocator appropriately, as described above. The storage space required for database connection objects depends to some extent on the length of the filename of the database file, but rarely exceeds 2KB on 32-bit systems. (More space is required on 64-bit systems due to the increased size of pointers.) Each parser object uses about 1.6KB of memory. Thus, elements 3 through 7 above can easily be controlled to keep the maximum memory allocation size below 2KB.

If the application is designed to manage data in small pieces, then the database should never contain any large strings or BLOBs and hence element 1 above should not be a factor. If the database does contain large strings or BLOBs, they should be read using incremental BLOB I/O and rows that contain the large strings or BLOBs should never be update by any means other than incremental BLOB I/O. Otherwise, the sqlite3_step() routine will need to read the entire row into contiguous memory at some point, and that will involve at least one large memory allocation.

The final source of large memory allocations is the space to hold the prepared statements that result from compiling complex SQL operations. Ongoing work by the SQLite developers is reducing the amount of space required here. But large and complex queries might still require prepared statements that are several kilobytes in size. The only workaround at the moment is for the application to break complex SQL operations up into two or more smaller and simpler operations contained in separate prepared statements.

All things considered, applications should normally be able to hold their maximum memory allocation size below 2K or 4K. This gives a value for log<sub>2</sub>(**n**) of 2 or 3. This will limit **N** to between 2 and 2.5 times **M**.

The maximum amount of general-purpose memory needed by the application is determined by such factors as how many simultaneous open database connection and prepared statement objects the application uses, and on the complexity of the prepared statements. For any given application, these factors are normally fixed and can be determined experimentally using SQLITE_STATUS_MEMORY_USED. A typical application might only use about 40KB of general-purpose memory. This gives a value of **N** of around 100KB.

4.2 Ductile failure

If the memory allocation subsystems within SQLite are configured for breakdown-free operation but the actual memory usage exceeds design limits set by the Robson proof, SQLite will usually continue to operate normally. The scratch memory allocator, the pagecache memory allocator, and the lookaside memory allocator all automatically failover to the memsys5 general-purpose memory allocator. And it is usually the case that the memsys5 memory allocator will continue to function without fragmentation even if **M** and/or **n** exceeds the limits imposed by the Robson proof. The Robson proof shows that it is possible for a memory allocation to break down and fail in this circumstance, but such a failure requires an especially despicable sequence of allocations and deallocations - a sequence that SQLite has never been observed to follow. So in practice it is usually the case that the limits imposed by Robson can be exceeded by a considerable margin with no ill effect.

Nevertheless, application developers are admonished to monitor the state of the memory allocation subsystems and raise alarms when memory usage approaches or exceeds Robson limits. In this way, the application will provide operators with abundant warning well in advance of failure. The memory statistics interfaces of SQLite provide the application with all the mechanism necessary to complete the monitoring portion of this task.

5.0 Stability Of Memory Interfaces

Update: As of SQLite version 3.7.0 (2010-07-22), all of SQLite memory allocation interfaces are considered stable and will be supported in future releases.

Custom Builds Of SQLite

or Porting SQLite To New Operating Systems

1.0 Introduction

For most applications, the recommended method for building SQLite is to use the amalgamation code file, **sqlite3.c**, and its corresponding header file **sqlite3.h**. The sqlite3.c code file should compile and run on any unix, Windows system without any changes or special compiler options. Most applications can simply include the sqlite3.c file together with the other C code files that make up the application, compile them all together, and have working and well configured version of SQLite.

> *Most applications work great with SQLite in its default configuration and with no special compile-time configuration. Most developers should be able to completely ignore this document and simply build SQLite from the amalgamation without any special knowledge and without taking any special actions.*

However, highly tuned and specialized applications may want or need to replace some of SQLite's built-in system interfaces with alternative implementations more suitable for the needs of the application. SQLite is designed to be easily reconfigured at compile-time to meet the specific needs of individual projects. Among the compile-time configuration options for SQLite are these:

- Replace the built-in mutex subsystem with an alternative implementation.
- Completely disable all mutexing for use in single-threaded applications.
- Reconfigure the memory allocation subsystem to use a memory allocator other the malloc() implementation from the standard library.
- Realign the memory allocation subsystem so that it never calls malloc() at all but instead satisfies all memory requests using a fixed-size memory buffer assigned to SQLite at startup.
- Replace the interface to the file system with an alternative design. In other words, override all of the system calls that SQLite makes in order to talk to the disk with a completely different set of system calls.
- Override other operating system interfaces such as calls to obtain Zulu or local time.

Generally speaking, there are three separate subsystems within SQLite that can be modified or overridden at compile-time. The mutex subsystem is used to serialize access to SQLite resources that are shared among threads. The memory allocation subsystem is used to allocate memory required by SQLite objects and for the database cache. Finally, the Virtual File System subsystem is used to provide a portable interface between SQLite and the underlying operating system and especially the file system. We call these three subsystems the "interface" subsystems of SQLite.

We emphasis that most applications are well-served by the built-in default implementations of the SQLite interface subsystems. Developers are encouraged to use the default built-in implementations whenever possible and to build SQLite without any special compile-time options or parameters. However, some highly specialized applications may benefit from substituting or modifying one or more of these built-in SQLite interface subsystems. Or, if SQLite is used on an operating system other than Unix (Linux or Mac OS X), Windows (Win32 or WinCE), or OS/2 then none of the interface subsystems that come built into SQLite will work and the application will need to provide alternative implementations suitable for the target platform.

2.0 Configuring Or Replacing The Mutex Subsystem

In a multithreaded environment, SQLite uses mutexes to serialize access to shared resources. The mutex subsystem is only required for applications that access SQLite from multiple threads. For single-threaded applications, or applications which only call SQLite from a single thread, the mutex subsystem can be completely disabled by recompiling with the following option:

```
-DSQLITE_THREADSAFE=0
```

Mutexes are cheap but they are not free, so performance will be better when mutexes are completely disabled. The resulting library footprint will also be a little smaller. Disabling the mutexes at compile-time is a recommended optimization for applications where it makes sense.

When using SQLite as a shared library, an application can test to see whether or not mutexes have been disabled using the sqlite3_threadsafe() API. Applications that link against SQLite at run-time and use SQLite from multiple threads should probably check this API to make sure they did not accidentally get linked against a version of the SQLite library that has its mutexes disabled. Single-threaded applications will, of course, work correctly regardless of whether or not SQLite is configured to be threadsafe, though they will be a little bit faster when using versions of SQLite with mutexes disabled.

SQLite mutexes can also be disabled at run-time using the sqlite3_config() interface. To completely disable all mutexing, the application can invoke:

```
sqlite3_config(SQLITE_CONFIG_SINGLETHREAD);
```

Disabling mutexes at run-time is not as effective as disabling them at compile-time since SQLite still must do a test of a boolean variable to see if mutexes are enabled or disabled at each point where a mutex might be required. But there is still a performance advantage for disabling mutexes at run-time.

For multi-threaded applications that are careful about how they manage threads, SQLite supports an alternative run-time configuration that is half way between not using any mutexes and the default situation of mutexing everything in sight. This in-the-middle mutex alignment can be established as follows:

```
sqlite3_config(SQLITE_CONFIG_MULTITHREAD);
sqlite3_config(SQLITE_CONFIG_MEMSTATUS, 0);
```

There are two separate configuration changes here which can be used either togethr or separately. The SQLITE_CONFIG_MULTITHREAD setting disables the mutexes that serialize access to database connection objects and prepared statement objects. With this setting, the application is free to use SQLite from multiple threads, but it must make sure than no two threads try to access the same database connection or any prepared statements associated with the same database connection at the same time. Two threads can use SQLite at the same time, but they must use separate database connections. The second SQLITE_CONFIG_MEMSTATUS setting disables the mechanism in SQLite that tracks the total size of all outstanding memory allocation requests. This omits the need to mutex each call to sqlite3_malloc() and sqlite3_free(), which saves a huge number of mutex operations. But a consequence of disabling the memory statistics mechanism is that the sqlite3_memory_used(), sqlite3_memory_highwater(), and sqlite3_soft_heap_limit64() interfaces cease to work.

SQLite uses pthreads for its mutex implementation on Unix and SQLite requires a recursive mutex. Most modern pthread implementations support recursive mutexes, but not all do. For systems that do not support recursive mutexes, it is recommended that applications operate in single-threaded mode only. If this is not possible, SQLite provides an alternative recursive mutex implementation built on top of the standard "fast" mutexes of pthreads. This alternative implementation should work correctly as long as pthread_equal() is atomic and the processor has a coherent data cache. The alternative recursive mutex implementation is enabled by the following compiler command-line switch:

```
-DSQLITE_HOMEGROWN_RECURSIVE_MUTEX=1
```

When porting SQLite to a new operating system, it is usually necessary to completely replace the built-in mutex subsystem with an alternative built around the mutex primitives of the new operating system. This is accomplished by compiling SQLite with the following option:

```
-DSQLITE_MUTEX_APPDEF=1
```

When SQLite is compiled with the SQLITE_MUTEX_APPDEF=1 option, it completely omits the implementation of its mutex primitive functions. But the SQLite library still attempts to call these functions where necessary, so the application must itself implement the mutex primitive functions and link them together with SQLite.

3.0 Configuring Or Replacing The Memory Allocation Subsystem

By default, SQLite obtains the memory it needs for objects and cache from the malloc()/free() implementation of the standard library. There is also on-going work with experimental memory allocators that satisfy all memory requests from a single fixed memory buffer handed to SQLite at application start. Additional information on these experimental memory allocators will be provided in a future revision of this document.

SQLite supports the ability of an application to specify an alternative memory allocator at run-time by filling in an instance of the sqlite3_mem_methods object with pointers to the routines of the alternative implementation then registering the new alternative implementation using the sqlite3_config() interface. For example:

```
sqlite3_config(SQLITE_CONFIG_MALLOC, &my_malloc_implementation);
```

SQLite makes a copy of the content of the sqlite3_mem_methods object so the object can be modified after the sqlite3_config() call returns.

4.0 Adding New Virtual File Systems

Since version 3.5.0, SQLite has supported an interface called the virtual file system or "VFS". This object is somewhat misnamed since it is really an interface to the whole underlying operating system, not just the filesystem.

One of the interesting features of the VFS interface is that SQLite can support multiple VFSes at the same time. Each database connection has to choose a single VFS for its use when the connection is first opened using sqlite3_open_v2(). But if a process contains multiple database connections each can choose a different VFS. VFSes can be added at run-time using the sqlite3_vfs_register() interface.

The default builds for SQLite on Unix, Windows, and OS/2 include a VFS appropriate for the target platform. SQLite builds for other operating systems do not contain a VFS by default, but the application can register one or more at run-time.

5.0 Porting SQLite To A New Operating System

In order to port SQLite to a new operating system - an operating system not supported by default - the application must provide...

- a working mutex subsystem (but only if it is multithreaded),
- a working memory allocation subsystem (assuming it lacks malloc() in its standard library), and
- a working VFS implementation.

All of these things can be provided in a single auxiliary C code file and then linked with the stock "sqlite3.c" code file to generate a working SQLite build for the target operating system. In addition to the alternative mutex and memory allocation subsystems and the new VFS, the auxiliary C code file should contain implementations for the following two routines:

- sqlite3_os_init()
- sqlite3_os_end()

The "sqlite3.c" code file contains default implementations of a VFS and of the sqlite3_initialize() and sqlite3_shutdown() functions that are appropriate for Unix, Windows, and OS/2. To prevent one of these default components from being loaded when sqlite3.c is compiled, it is necessary to add the following compile-time option:

```
-DSQLITE_OS_OTHER=1
```

The SQLite core will call sqlite3_initialize() early. The auxiliary C code file can contain an implementation of sqlite3_initialize() that registers an appropriate VFS and also perhaps initializes an alternative mutex system (if mutexes are required) or does any memory allocation subsystem initialization that is required. The SQLite core never calls sqlite3_shutdown() but it is part of the official SQLite API and is not otherwise provided when compiled with -DSQLITE_OS_OTHER=1, so the auxiliary C code file should probably provide it for completeness.

Locking And Concurrency In SQLite Version 3

This document was originally created in early 2004 when SQLite version 2 was still in widespread use and was written to introduce the new concepts of SQLite version 3 to readers who were already familiar with SQLite version 2. But these days, most readers of this document have probably never seen SQLite version 2 and are only familiar with SQLite version 3. Nevertheless, this document continues to serve as an authoritative reference to how database file locking works in SQLite version 3.

The document only describes locking for the older rollback-mode transaction mechanism. Locking for the newer write-ahead log or WAL mode is described separately.

1.0 File Locking And Concurrency In SQLite Version 3

SQLite Version 3.0.0 introduced a new locking and journaling mechanism designed to improve concurrency over SQLite version 2 and to reduce the writer starvation problem. The new mechanism also allows atomic commits of transactions involving multiple database files. This document describes the new locking mechanism. The intended audience is programmers who want to understand and/or modify the pager code and reviewers working to verify the design of SQLite version 3.

2.0 Overview

Locking and concurrency control are handled by the pager module. The pager module is responsible for making SQLite "ACID" (Atomic, Consistent, Isolated, and Durable). The pager module makes sure changes happen all at once, that either all changes occur or none of them do, that two or more processes do not try to access the database in incompatible ways at the same time, and that once changes have been written they persist until explicitly deleted. The pager also provides a memory cache of some of the contents of the disk file.

The pager is unconcerned with the details of B-Trees, text encodings, indices, and so forth. From the point of view of the pager the database consists of a single file of uniform-sized blocks. Each block is called a "page" and is usually 1024 bytes in size. The pages are numbered beginning with 1. So the first 1024 bytes of the database are called "page 1" and the second 1024 bytes are call "page 2" and so forth. All other encoding details are handled by higher layers of the library. The pager communicates with the operating system using one of several modules (Examples: os_unix.c, os_win.c) that provides a uniform abstraction for operating system services.

The pager module effectively controls access for separate threads, or separate processes, or both. Throughout this document whenever the word "process" is written you may substitute the word "thread" without changing the truth of the statement.

3.0 Locking

From the point of view of a single process, a database file can be in one of five locking states:

UNLOCKED	No locks are held on the database. The database may be neither read nor written. Any internally cached data is considered suspect and subject to verification against the database file before being used. Other processes can read or write the database as their own locking states permit. This is the default state.
SHARED	The database may be read but not written. Any number of processes can hold SHARED locks at the same time, hence there can be many simultaneous readers. But no other thread or process is allowed to write to the database file while one or more SHARED locks are active.
RESERVED	A RESERVED lock means that the process is planning on writing to the database file at some point in the future but that it is currently just reading from the file. Only a single RESERVED lock may be active at one time, though multiple SHARED locks can coexist with a single RESERVED lock. RESERVED differs from PENDING in that new SHARED locks can be acquired while there is a RESERVED lock.
PENDING	A PENDING lock means that the process holding the lock wants to write to the database as soon as possible and is just waiting on all current SHARED locks to clear so that it can get an EXCLUSIVE lock. No new SHARED locks are permitted against the database if a PENDING lock is active, though existing SHARED locks are allowed to continue.
EXCLUSIVE	An EXCLUSIVE lock is needed in order to write to the database file. Only one EXCLUSIVE lock is allowed on the file and no other locks of any kind are allowed to coexist with an EXCLUSIVE lock. In order to maximize concurrency, SQLite works to minimize the amount of time that EXCLUSIVE locks are held.

The operating system interface layer understands and tracks all five locking states described above. The pager module only tracks four of the five locking states. A PENDING lock is always just a temporary stepping stone on the path to an EXCLUSIVE lock and so the pager module does not track PENDING locks.

4.0 The Rollback Journal

When a process wants to change a database file (and it is not in WAL mode), it first records the original unchanged database content in a *rollback journal*. The rollback journal is an ordinary disk file that is always located in the same directory or folder as the database file and has the same name as the database file with the addition of a `-journal` suffix. The rollback journal also records the initial size of the database so that if the database file grows it can be truncated back to its original size on a rollback.

If SQLite is working with multiple databases at the same time (using the ATTACH command) then each database has its own rollback journal. But there is also a separate aggregate journal called the *master journal*. The master journal does not contain page data used for rolling back changes. Instead the master journal contains the names of the individual database rollback journals for each of the ATTACHed databases. Each of the individual database rollback journals also contain the name of the master journal. If there are no ATTACHed databases (or if none of the ATTACHed database is participating in the current transaction) no master journal is created and the normal rollback journal contains an empty string in the place normally reserved for recording the name of the master journal.

A rollback journal is said to be *hot* if it needs to be rolled back in order to restore the integrity of its database. A hot journal is created when a process is in the middle of a database update and a program or operating system crash or power failure prevents the update from completing. Hot journals are an exception condition. Hot journals exist to recover from crashes and power failures. If everything is working correctly (that is, if there are no crashes or power failures) you will never get a hot journal.

If no master journal is involved, then a journal is hot if it exists and has a non-zero header and its corresponding database file does not have a RESERVED lock. If a master journal is named in the file journal, then the file journal is hot if its master journal exists and there is no RESERVED lock on the corresponding database file. It is important to understand when a journal is hot so the preceding rules will be repeated in bullets:

- A journal is hot if...
    - It exists, and
    - Its size is greater than 512 bytes, and
    - The journal header is non-zero and well-formed, and
    - Its master journal exists or the master journal name is an empty string, and
    - There is no RESERVED lock on the corresponding database file.

4.1 Dealing with hot journals

Before reading from a database file, SQLite always checks to see if that database file has a hot journal. If the file does have a hot journal, then the journal is rolled back before the file is read. In this way, we ensure that the database file is in a consistent state before it is read.

When a process wants to read from a database file, it followed the following sequence of steps:

1. Open the database file and obtain a SHARED lock. If the SHARED lock cannot be obtained, fail immediately and return SQLITE_BUSY.
2. Check to see if the database file has a hot journal. If the file does not have a hot journal, we are done. Return immediately. If there is a hot journal, that journal must be rolled back by the subsequent steps of this algorithm.
3. Acquire a PENDING lock then an EXCLUSIVE lock on the database file. (Note: Do not acquire a RESERVED lock because that would make other processes think the journal was no longer hot.) If we fail to acquire these locks it means another process is already trying to do the rollback. In that case, drop all locks, close the database, and return SQLITE_BUSY.
4. Read the journal file and roll back the changes.
5. Wait for the rolled back changes to be written onto persistent storage. This protects the integrity of the database in case another power failure or crash occurs.
6. Delete the journal file (or truncate the journal to zero bytes in length if PRAGMA journal_mode=TRUNCATE is set, or zero the journal header if PRAGMA journal_mode=PERSIST is set).
7. Delete the master journal file if it is safe to do so. This step is optional. It is here only to prevent stale master journals from cluttering up the disk drive. See the discussion below for details.
8. Drop the EXCLUSIVE and PENDING locks but retain the SHARED lock.

After the algorithm above completes successfully, it is safe to read from the database file. Once all reading has completed, the SHARED lock is dropped.

4.2 Deleting stale master journals

A stale master journal is a master journal that is no longer being used for anything. There is no requirement that stale master journals be deleted. The only reason for doing so is to free up disk space.

A master journal is stale if no individual file journals are pointing to it. To figure out if a master journal is stale, we first read the master journal to obtain the names of all of its file journals. Then we check each of those file journals. If any of the file journals named in the master journal exists and points back to the master journal, then the master journal is not stale. If all file journals are either missing or refer to other master journals or no master journal at all, then the master journal we are testing is stale and can be safely deleted.

5.0 Writing to a database file

To write to a database, a process must first acquire a SHARED lock as described above (possibly rolling back incomplete changes if there is a hot journal). After a SHARED lock is obtained, a RESERVED lock must be acquired. The RESERVED lock signals that the process intends to write to the database at some point in the future. Only one process at a time can hold a RESERVED lock. But other processes can continue to read the database while the RESERVED lock is held.

If the process that wants to write is unable to obtain a RESERVED lock, it must mean that another process already has a RESERVED lock. In that case, the write attempt fails and returns SQLITE_BUSY.

After obtaining a RESERVED lock, the process that wants to write creates a rollback journal. The header of the journal is initialized with the original size of the database file. Space in the journal header is also reserved for a master journal name, though the master journal name is initially empty.

Before making changes to any page of the database, the process writes the original content of that page into the rollback journal. Changes to pages are held in memory at first and are not written to the disk. The original database file remains unaltered, which means that other processes can continue to read the database.

Eventually, the writing process will want to update the database file, either because its memory cache has filled up or because it is ready to commit its changes. Before this happens, the writer must make sure no other process is reading the database and that the rollback journal data is safely on the disk surface so that it can be used to rollback incomplete changes in the event of a power failure. The steps are as follows:

1. Make sure all rollback journal data has actually been written to the surface of the disk (and is not just being held in the operating system's or disk controllers cache) so that if a power failure occurs the data will still be there after power is restored.
2. Obtain a PENDING lock and then an EXCLUSIVE lock on the database file. If other processes still have SHARED locks, the writer might have to wait until those SHARED locks clear before it is able to obtain an EXCLUSIVE lock.
3. Write all page modifications currently held in memory out to the original database disk file.

If the reason for writing to the database file is because the memory cache was full, then the writer will not commit right away. Instead, the writer might continue to make changes to other pages. Before subsequent changes are written to the database file, the rollback journal must be flushed to disk again. Note also that the EXCLUSIVE lock that the writer obtained in order to write to the database initially must be held until all changes are committed. That means that no other processes are able to access the database from the time the memory cache first spills to disk until the transaction commits.

When a writer is ready to commit its changes, it executes the following steps:

1. Obtain an EXCLUSIVE lock on the database file and make sure all memory changes have been written to the database file using the algorithm of steps 1-3 above.
2. Flush all database file changes to the disk. Wait for those changes to actually be written onto the disk surface.
3. Delete the journal file. (Or if the PRAGMA journal_mode is TRUNCATE or PERSIST, truncate the journal file or zero the header of the journal file, respectively.) This is the instant when the changes are committed. Prior to deleting the journal file, if a power failure or crash occurs, the next process to open the database will see that it has a hot journal and will roll the changes back. After the journal is deleted, there will no longer be a hot journal and the changes will persist.
4. Drop the EXCLUSIVE and PENDING locks from the database file.

As soon as the PENDING lock is released from the database file, other processes can begin reading the database again. In the current implementation, the RESERVED lock is also released, but that is not essential for correct operation.

If a transaction involves multiple databases, then a more complex commit sequence is used, as follows:

1. Make sure all individual database files have an EXCLUSIVE lock and a valid journal.
2. Create a master-journal. The name of the master-journal is arbitrary. (The current implementation appends random suffixes to the name of the main database file until it finds a name that does not previously exist.) Fill the master journal with the names of all the individual journals and flush its contents to disk.
3. Write the name of the master journal into all individual journals (in space set aside for that purpose in the headers of the individual journals) and flush the contents of the individual journals to disk and wait for those changes to reach the disk surface.
4. Flush all database file changes to the disk. Wait for those changes to actually be written onto the disk surface.
5. Delete the master journal file. This is the instant when the changes are committed. Prior to deleting the master journal file, if a power failure or crash occurs, the individual file journals will be considered hot and will be rolled back by the next process that attempts to read them. After the master journal has been deleted, the file journals will no longer

be considered hot and the changes will persist.

6. Delete all individual journal files.
7. Drop the EXCLUSIVE and PENDING locks from all database files.

5.1 Writer starvation

In SQLite version 2, if many processes are reading from the database, it might be the case that there is never a time when there are no active readers. And if there is always at least one read lock on the database, no process would ever be able to make changes to the database because it would be impossible to acquire a write lock. This situation is called *writer starvation*.

SQLite version 3 seeks to avoid writer starvation through the use of the PENDING lock. The PENDING lock allows existing readers to continue but prevents new readers from connecting to the database. So when a process wants to write a busy database, it can set a PENDING lock which will prevent new readers from coming in. Assuming existing readers do eventually complete, all SHARED locks will eventually clear and the writer will be given a chance to make its changes.

6.0 How To Corrupt Your Database Files

The pager module is very robust but it can be subverted. This section attempts to identify and explain the risks. (See also the Things That Can Go Wrong section of the article on Atomic Commit.

Clearly, a hardware or operating system fault that introduces incorrect data into the middle of the database file or journal will cause problems. Likewise, if a rogue process opens a database file or journal and writes malformed data into the middle of it, then the database will become corrupt. There is not much that can be done about these kinds of problems so they are given no further attention.

SQLite uses POSIX advisory locks to implement locking on Unix. On Windows it uses the LockFile(), LockFileEx(), and UnlockFile() system calls. SQLite assumes that these system calls all work as advertised. If that is not the case, then database corruption can result. One should note that POSIX advisory locking is known to be buggy or even unimplemented on many NFS implementations (including recent versions of Mac OS X) and that there are reports of locking problems for network filesystems under Windows. Your best defense is to not use SQLite for files on a network filesystem.

SQLite uses the fsync() system call to flush data to the disk under Unix and it uses the FlushFileBuffers() to do the same under Windows. Once again, SQLite assumes that these operating system services function as advertised. But it has been reported that fsync() and

FlushFileBuffers() do not always work correctly, especially with inexpensive IDE disks. Apparently some manufactures of IDE disks have controller chips that report that data has reached the disk surface when in fact the data is still in volatile cache memory in the disk drive electronics. There are also reports that Windows sometimes chooses to ignore FlushFileBuffers() for unspecified reasons. The author cannot verify any of these reports. But if they are true, it means that database corruption is a possibility following an unexpected power loss. These are hardware and/or operating system bugs that SQLite is unable to defend against.

If a Linux ext3 filesystem is mounted without the "barrier=1" option in the /etc/fstab and the disk drive write cache is enabled then filesystem corruption can occur following a power loss or OS crash. Whether or not corruption can occur depends on the details of the disk control hardware; corruption is more likely with inexpensive consumer-grade disks and less of a problem for enterprise-class storage devices with advanced features such as non-volatile write caches. Various ext3 experts confirm this behavior. We are told that most Linux distributions do not use barrier=1 and do not disable the write cache so most Linux distributions are vulnerable to this problem. Note that this is an operating system and hardware issue and that there is nothing that SQLite can do to work around it. Other database engines have also run into this same problem.

If a crash or power failure occurs and results in a hot journal but that journal is deleted, the next process to open the database will not know that it contains changes that need to be rolled back. The rollback will not occur and the database will be left in an inconsistent state. Rollback journals might be deleted for any number of reasons:

- An administrator might be cleaning up after an OS crash or power failure, see the journal file, think it is junk, and delete it.
- Someone (or some process) might rename the database file but fail to also rename its associated journal.
- If the database file has aliases (hard or soft links) and the file is opened by a different alias than the one used to create the journal, then the journal will not be found. To avoid this problem, you should not create links to SQLite database files.
- Filesystem corruption following a power failure might cause the journal to be renamed or deleted.

The last (fourth) bullet above merits additional comment. When SQLite creates a journal file on Unix, it opens the directory that contains that file and calls fsync() on the directory, in an effort to push the directory information to disk. But suppose some other process is adding or removing unrelated files to the directory that contains the database and journal at the moment of a power failure. The supposedly unrelated actions of this other process might result in the journal file being dropped from the directory and moved into "lost+found". This is an unlikely scenario, but it could happen. The best defenses are to use a journaling filesystem or to keep the database and journal in a directory by themselves.

For a commit involving multiple databases and a master journal, if the various databases were on different disk volumes and a power failure occurs during the commit, then when the machine comes back up the disks might be remounted with different names. Or some disks might not be mounted at all. When this happens the individual file journals and the master journal might not be able to find each other. The worst outcome from this scenario is that the commit ceases to be atomic. Some databases might be rolled back and others might not. All databases will continue to be self-consistent. To defend against this problem, keep all databases on the same disk volume and/or remount disks using exactly the same names after a power failure.

7.0 Transaction Control At The SQL Level

The changes to locking and concurrency control in SQLite version 3 also introduce some subtle changes in the way transactions work at the SQL language level. By default, SQLite version 3 operates in *autocommit* mode. In autocommit mode, all changes to the database are committed as soon as all operations associated with the current database connection complete.

The SQL command "BEGIN TRANSACTION" (the TRANSACTION keyword is optional) is used to take SQLite out of autocommit mode. Note that the BEGIN command does not acquire any locks on the database. After a BEGIN command, a SHARED lock will be acquired when the first SELECT statement is executed. A RESERVED lock will be acquired when the first INSERT, UPDATE, or DELETE statement is executed. No EXCLUSIVE lock is acquired until either the memory cache fills up and must be spilled to disk or until the transaction commits. In this way, the system delays blocking read access to the file file until the last possible moment.

The SQL command "COMMIT" does not actually commit the changes to disk. It just turns autocommit back on. Then, at the conclusion of the command, the regular autocommit logic takes over and causes the actual commit to disk to occur. The SQL command "ROLLBACK" also operates by turning autocommit back on, but it also sets a flag that tells the autocommit logic to rollback rather than commit.

If the SQL COMMIT command turns autocommit on and the autocommit logic then tries to commit change but fails because some other process is holding a SHARED lock, then autocommit is turned back off automatically. This allows the user to retry the COMMIT at a later time after the SHARED lock has had an opportunity to clear.

If multiple commands are being executed against the same SQLite database connection at the same time, the autocommit is deferred until the very last command completes. For example, if a SELECT statement is being executed, the execution of the command will pause as each row of the result is returned. During this pause other INSERT, UPDATE, or DELETE commands can be executed against other tables in the database. But none of these changes will commit until the original SELECT statement finishes.

Isolation In SQLite

The "isolation" property of a database determines when changes made to the database by one operation become visible to other concurrent operations.

Isolation Between Database Connections

If the same database is being read and written using two different database connections (two different sqlite3 objects returned by separate calls to sqlite3_open()) and the two database connections do not have a shared cache, then the reader is only able to see complete committed transactions from the writer. Partial changes by the writer that have not been committed are invisible to the reader. This is true regardless of whether the two database connections are in the same thread, in different threads of the same process, or in different processes. This is the usual and expected behavior for SQL database systems.

The previous paragraph is also true (separate database connections are isolated from one another) in shared cache mode as long as the read_uncommitted pragma remains turned off. The read_uncommitted pragma is off by default and so if the application does nothing to turn it on, it will remain off. Hence, unless the read_uncommitted pragma is used to change the default behavior, changes made by one database connection are invisible to readers on a different database connection sharing the same cache until the writer commits its transaction.

If two database connections shared the same cache and the reader has enabled the read_uncommitted pragma, then the reader will be able to see changes made by the writer before the writer transaction commits. The combined use of shared cache mode and the read_uncommitted pragma is the only way that one database connection can see uncommitted changes on a different database connection. In all other circumstances, separate database connections are completely isolated from one another.

Except in the case of shared cache database connections with PRAGMA read_uncommitted turned on, all transactions in SQLite show "serializable" isolation. SQLite implements serializable transactions by actually serializing the writes. There can only be a single writer at a time to an SQLite database. There can be multiple database connections open at the same time, and all of those database connections can write to the database file, but they have to take turns. SQLite uses locks to serialization of the writes automatically; this is not something that the applications using SQLite need to worry with.

Isolation And Concurrency

SQLite implements isolation and concurrency control (and atomicity) using transient journal files that appear in the same directory in as the database file. There are two major "journal modes". The older "rollback mode" corresponds to using the "DELETE", "PERSIST", or "TRUNCATE" options to the journal_mode pragma. In rollback mode, changes are written directly into the database file, while simultaneously a separate rollback journal file is constructed that is able to restore the database to its original state if the transaction rolls back. Rollback mode (specifically DELETE mode, meaning that the rollback journal is deleted from disk at the conclusion of each transaction) is the current default behavior.

Since version 3.7.0, SQLite also supports "WAL mode". In WAL mode, changes are not written to the original database file. Instead, changes go into a separate "write-ahead log" or "WAL" file. Later, after the transaction commits, those changes will be moved from the WAL file back into the original database in an operation called "checkpoint". WAL mode is enabled by running "PRAGMA journal_mode=WAL".

In rollback mode, SQLite implements isolation by locking the database file and preventing any reads by other database connections while each write transaction is underway. Readers can be be active at the beginning of a write, before any content is flushed to disk and while all changes are still held in the writer's private memory space. But before any changes are made to the database file on disk, all readers must be (temporally) expelled in order to give the writer exclusive access to the database file. Hence, readers are prohibited from seeing incomplete transactions by virtue of being locked out of the database while the transaction is being written to disk. Only after the transaction is completely written and synced to disk and commits are the readers allowed back into the database. Hence readers never get a chance to see partially written changes.

WAL mode permits simultaneous readers and writers. It can do this because changes do not overwrite the original database file, but rather go into the separate write-ahead log file. That means that readers can continue to read the old, original, unaltered content from the original database file at the same time that the writer is appending to the write-ahead log. In WAL mode, SQLite exhibits "snapshot isolation". When a read transaction starts, that reader continues to see an unchanging "snapshot" of the database file as it existed at the moment in time when the read transaction started. Any write transactions that commit while the read transaction is active are still invisible to the read transaction, because the reader is seeing a snapshot of database file from a prior moment in time.

An example: Suppose there are two database connections X and Y. X starts a read transaction using BEGIN followed by one or more SELECT statements. Then Y comes along and runs an UPDATE statement to modify the database. X can subsequently do a SELECT against the records that Y modified but X will see the older unmodified entries because Y's changes are all invisible to X while X is holding a read transaction. If X wants to see the changes that Y made, then X must ends its read transaction and start a new one (by running COMMIT followed by another BEGIN.)

Another example: X starts a read transaction using BEGIN and SELECT, then Y makes a changes to the database using UPDATE. Then X tries to make a change to the database using UPDATE. The attempt by X to escalate its transaction from a read transaction to a write transaction fails with an SQLITE_BUSY_SNAPSHOT error because the snapshot of the database being viewed by X is no longer the latest version of the database. If X were allowed to write, it would fork the history of the database file, which is something SQLite does not support. In order for X to write to the database, it must first release its snapshot (using ROLLBACK for example) then start a new transaction with a subsequent BEGIN.

If X starts a transaction that will initially only read but X knows it will eventually want to write and does not want to be troubled with possible SQLITE_BUSY_SNAPSHOT errors that arise because another connection jumped ahead of it in line, then X can issue BEGIN IMMEDIATE to start its transaction instead of just an ordinary BEGIN. The BEGIN IMMEDIATE command goes ahead and starts a write transaction, and thus blocks all other writers. If the BEGIN IMMEDIATE operation succeeds, then no subsequent operations in that transaction will ever fail with an SQLITE_BUSY error.

No Isolation Between Operations On The Same Database Connection

SQLite provides isolation between operations in separate database connections. However, there is no isolation between operations that occur within the same database connection.

In other words, if X begins a write transaction using BEGIN IMMEDIATE then issues one or more UPDATE, DELETE, and/or INSERT statements, then those changes are visible to subsequent SELECT statements that are evaluated in database connection X. SELECT statements on a different database connection Y will show no changes until the X transaction commits. But SELECT statements in X will show the changes prior to the commit.

Within a single database connection X, a SELECT statement always sees all changes to the database that are completed prior to the start of the SELECT statement, whether committed or uncommitted. And the SELECT statement obviously does not see any changes that occur after the SELECT statement completes. But what about changes that occur while the SELECT statement is running? What if a SELECT statement is started and the sqlite3_step() interface steps through roughly half of its output, then some UPDATE statements are run by the application that modify the table that the SELECT statement is reading, then more calls to sqlite3_step() are made to finish out the SELECT statement? Will the later steps of the SELECT statement see the changes made by the UPDATE or not? The answer is that this behavior is undefined. In particular, whether or not the SELECT statement sees the concurrent changes depends on which release of SQLite is running, the schema of the database file, whether or not ANALYZE has been run, and the details of the query. In some cases, it might depend on the content of the database file, too. There is no good way to know whether or not a SELECT statement will see changes that were made to the database by the same database connection after the SELECT statement was started. And hence, developers should diligently avoid writing applications that make assumptions about what will occur in that circumstance.

If an application issues a SELECT statement on a single table like "*SELECT rowid, * FROM table WHERE ...*" and starts stepping through the output of that statement using sqlite3_step() and examining each row, then it is safe for the application to delete the current row or any prior row using "DELETE FROM table WHERE rowid=?". It is also safe (in the sense that it will not harm the database) for the application to delete a row that expected to appear later in the query but has not appeared yet. If a future row is deleted, however, it might happen that the row turns up after a subsequent sqlite3_step(), even after it has allegedly been deleted. Or it might not. That behavior is undefined. The application can also INSERT new rows into the table while the SELECT statement is running, but whether or not the new rows appear in subsequent sqlite3_step()s of the query is undefined. And the application can UPDATE the current row or any prior row, though doing so might cause that row to reappear in a subsequent sqlite3_step(). As long as the application is prepared to deal with these ambiguities, the operations themselves are safe and will not harm the database file.

For the purposes of the previous two paragraphs, two database connections that have the same shared cache and which have enabled PRAGMA read_uncommitted are considered to be the same database connection.

Summary

1. Transactions in SQLite are SERIALIZABLE.
2. Changes made in one database connection are invisible to all other database connections prior to commit.
3. A query sees all changes that are completed on the same database connection prior to the start of the query, regardless of whether or not those changes have been committed.

4. If changes occur on the same database connection after a query starts running but before the query completes, then it is undefined whether or not the query will see those changes.
5. If changes occur on the same database connection after a query starts running but before the query completes, then the query might return a changed row more than once, or it might return a row that was previously deleted.
6. For the purposes of the previous four items, two database connections that use the same shared cache and which enable PRAGMA read_uncommitted are considered to be the same database connection, not separate database connections.

The SQLite Query Planner

This document provides overview of how the query planner and optimizer for SQLite works.

Given a single SQL statement, there might be dozens, hundreds, or even thousands of ways to implement that statement, depending on the complexity of the statement itself and of the underlying database schema. The task of the query planner is to select an algorithm from among the many choices that provides the answer with a minimum of disk I/O and CPU overhead.

With release 3.8.0, the SQLite query planner was reimplemented as the Next Generation Query Planner or "NGQP". All of the features, techniques, and algorithms described in this document are applicable to both the pre-3.8.0 legacy query planner and to the NGQP. For further information on how the NGQP differs from the legacy query planner, see the detailed description of the NGQP.

1.0 WHERE clause analysis

The WHERE clause on a query is broken up into "terms" where each term is separated from the others by an AND operator. If the WHERE clause is composed of constraints separate by the OR operator then the entire clause is considered to be a single "term" to which the OR-clause optimization is applied.

All terms of the WHERE clause are analyzed to see if they can be satisfied using indices. To be usable by an index a term must be of one of the following forms:

```
column = expressioncolumn IS expressioncolumn &gt; expressioncolumn &gt;= expression
```

If an index is created using a statement like this:

```
CREATE INDEX idx_ex1 ON ex1(a,b,c,d,e,...,y,z);
```

Then the index might be used if the initial columns of the index (columns a, b, and so forth) appear in WHERE clause terms. The initial columns of the index must be used with the `**&lt;big&gt;=&lt;/big&gt;**` or `**&lt;big&gt;IN&lt;/big&gt;**` or `**&lt;big&gt;IS&lt;/big&gt;**` operators. The right-most column that is used can employ inequalities. For the right-most column of an index that is used, there can be up to two inequalities that must sandwich the allowed values of the column between two extremes.

It is not necessary for every column of an index to appear in a WHERE clause term in order for that index to be used. But there cannot be gaps in the columns of the index that are used. Thus for the example index above, if there is no WHERE clause term that constraints column c, then terms that constrain columns a and b can be used with the index but not terms that constraint columns d through z. Similarly, index columns will not normally be used (for indexing purposes) if they are to the right of a column that is constrained only by inequalities. (See the skip-scan optimization below for the exception.)

In the case of indexes on expressions, whenever the word "column" is used in the foregoing text, one can substitute "indexed expression" (meaning a copy of the expression that appears in the CREATE INDEX statement) and everything will work the same.

1.1 Index term usage examples

For the index above and WHERE clause like this:

```
... WHERE a=5 AND b IN (1,2,3) AND c IS NULL AND d='hello'
```

The first four columns a, b, c, and d of the index would be usable since those four columns form a prefix of the index and are all bound by equality constraints.

For the index above and WHERE clause like this:

```
... WHERE a=5 AND b IN (1,2,3) AND c&gt;12 AND d='hello'
```

Only columns a, b, and c of the index would be usable. The d column would not be usable because it occurs to the right of c and c is constrained only by inequalities.

For the index above and WHERE clause like this:

```
... WHERE a=5 AND b IN (1,2,3) AND d='hello'
```

Only columns a and b of the index would be usable. The d column would not be usable because column c is not constrained and there can be no gaps in the set of columns that usable by the index.

For the index above and WHERE clause like this:

```
... WHERE b IN (1,2,3) AND c NOT NULL AND d='hello'
```

The index is not usable at all because the left-most column of the index (column "a") is not constrained. Assuming there are no other indices, the query above would result in a full table scan.

For the index above and WHERE clause like this:

```
  ... WHERE a=5 OR b IN (1,2,3) OR c NOT NULL OR d='hello'
```

The index is not usable because the WHERE clause terms are connected by OR instead of AND. This query would result in a full table scan. However, if three additional indices where added that contained columns b, c, and d as their left-most columns, then the OR-clause optimization might apply.

2.0 The BETWEEN optimization

If a term of the WHERE clause is of the following form:

```
expr1 BETWEEN expr2 AND expr3
```

Then two "virtual" terms are added as follows:

```
expr1 &gt;= expr2 AND expr1 &lt;= expr3
```

Virtual terms are used for analysis only and do not cause any VDBE code to be generated. If both virtual terms end up being used as constraints on an index, then the original BETWEEN term is omitted and the corresponding test is not performed on input rows. Thus if the BETWEEN term ends up being used as an index constraint no tests are ever performed on that term. On the other hand, the virtual terms themselves never causes tests to be performed on input rows. Thus if the BETWEEN term is not used as an index constraint and instead must be used to test input rows, the *expr1* expression is only evaluated once.

3.0 OR optimizations

WHERE clause constraints that are connected by OR instead of AND can be handled in two different ways. If a term consists of multiple subterms containing a common column name and separated by OR, like this:

```
column = expr1 OR column = expr2 OR column = expr3 OR ...
```

Then that term is rewritten as follows:

```
column IN (expr1,expr2,expr3,...)
```

The rewritten term then might go on to constrain an index using the normal rules for `**&lt;big&gt;IN&lt;/big&gt;**` operators. Note that *column* must be the same column in every OR-connected subterm, although the column can occur on either the left or the right side of the `**&lt;big&gt;=&lt;/big&gt;**` operator.

If and only if the previously described conversion of OR to an IN operator does not work, the second OR-clause optimization is attempted. Suppose the OR clause consists of multiple subterms as follows:

```
expr1 OR expr2 OR expr3
```

Individual subterms might be a single comparison expression like `**&lt;big&gt;a=5&lt;/big&gt;**` or `**&lt;big&gt;x&gt;y&lt;/big&gt;**` or they can be LIKE or BETWEEN expressions, or a subterm can be a parenthesized list of AND-connected sub-subterms. Each subterm is analyzed as if it were itself the entire WHERE clause in order to see if the subterm is indexable by itself. If every subterm of an OR clause is separately indexable then the OR clause might be coded such that a separate index is used to evaluate each term of the OR clause. One way to think about how SQLite uses separate indices for each OR clause term is to imagine that the WHERE clause where rewritten as follows:

```
 rowid IN (SELECT rowid FROM table WHERE expr1 UNION SELECT rowid FROM table WHERE e
```

The rewritten expression above is conceptual; WHERE clauses containing OR are not really rewritten this way. The actual implementation of the OR clause uses a mechanism that is more efficient and that works even for WITHOUT ROWID tables or tables in which the "rowid" is inaccessible. But the essence of the implementation is captured by the statement above: Separate indices are used to find candidate result rows from each OR clause term and the final result is the union of those rows.

Note that in most cases, SQLite will only use a single index for each table in the FROM clause of a query. The second OR-clause optimization described here is the exception to that rule. With an OR-clause, a different index might be used for each subterm in the OR-clause.

For any given query, the fact that the OR-clause optimization described here can be used does not guarantee that it will be used. SQLite uses a cost-based query planner that estimates the CPU and disk I/O costs of various competing query plans and chooses the plan that it thinks will be the fastest. If there are many OR terms in the WHERE clause or if some of the indices on individual OR-clause subterms are not very selective, then SQLite might decide that it is faster to use a different query algorithm, or even a full-table scan. Application developers can use the EXPLAIN QUERY PLAN prefix on a statement to get a high-level overview of the chosen query strategy.

4.0 The LIKE optimization

Terms that are composed of the LIKE or GLOB operator can sometimes be used to constrain indices. There are many conditions on this use:

1. The left-hand side of the LIKE or GLOB operator must be the name of an indexed column with TEXT affinity.
2. The right-hand side of the LIKE or GLOB must be either a string literal or a parameter bound to a string literal that does not begin with a wildcard character.
3. The ESCAPE clause cannot appear on the LIKE operator.
4. The built-in functions used to implement LIKE and GLOB must not have been overloaded using the sqlite3_create_function() API.
5. For the GLOB operator, the column must be indexed using the built-in BINARY collating sequence.
6. For the LIKE operator, if case_sensitive_like mode is enabled then the column must indexed using BINARY collating sequence, or if case_sensitive_like mode is disabled then the column must indexed using built-in NOCASE collating sequence.

The LIKE operator has two modes that can be set by a pragma. The default mode is for LIKE comparisons to be insensitive to differences of case for latin1 characters. Thus, by default, the following expression is true:

```
'a' LIKE 'A'
```

But if the case_sensitive_like pragma is enabled as follows:

```
PRAGMA case_sensitive_like=ON;
```

Then the LIKE operator pays attention to case and the example above would evaluate to false. Note that case insensitivity only applies to latin1 characters - basically the upper and lower case letters of English in the lower 127 byte codes of ASCII. International character sets are case sensitive in SQLite unless an application-defined collating sequence and like() SQL function are provided that take non-ASCII characters into account. But if an application-defined collating sequence and/or like() SQL function are provided, the LIKE optimization described here will never be taken.

The LIKE operator is case insensitive by default because this is what the SQL standard requires. You can change the default behavior at compile time by using the SQLITE_CASE_SENSITIVE_LIKE command-line option to the compiler.

The LIKE optimization might occur if the column named on the left of the operator is indexed using the built-in BINARY collating sequence and case_sensitive_like is turned on. Or the optimization might occur if the column is indexed using the built-in NOCASE collating sequence and the case_sensitive_like mode is off. These are the only two combinations under which LIKE operators will be optimized.

The GLOB operator is always case sensitive. The column on the left side of the GLOB operator must always use the built-in BINARY collating sequence or no attempt will be made to optimize that operator with indices.

The LIKE optimization will only be attempted if the right-hand side of the GLOB or LIKE operator is either literal string or a parameter that has been bound to a string literal. The string literal must not begin with a wildcard; if the right-hand side begins with a wildcard character then this optimization is attempted. If the right-hand side is a parameter that is bound to a string, then this optimization is only attempted if the prepared statement containing the expression was compiled with sqlite3_prepare_v2() or sqlite3_prepare16_v2(). The LIKE optimization is not attempted if the right-hand side is a parameter and the statement was prepared using sqlite3_prepare() or sqlite3_prepare16(). The LIKE optimization is not attempted if there is an ESCAPE phrase on the LIKE operator.

Suppose the initial sequence of non-wildcard characters on the right-hand side of the LIKE or GLOB operator is *x*. We are using a single character to denote this non-wildcard prefix but the reader should understand that the prefix can consist of more than 1 character. Let *y* be the smallest string that is the same length as /x/ but which compares greater than *x*. For example, if *x* is `**&lt;big&gt;hello&lt;/big&gt;**` then *y* would be `**&lt;big&gt;hellp&lt;/big&gt;**`. The LIKE and GLOB optimizations consist of adding two virtual terms like this:

```
column &gt;= x AND column &lt; y
```

Under most circumstances, the original LIKE or GLOB operator is still tested against each input row even if the virtual terms are used to constrain an index. This is because we do not know what additional constraints may be imposed by characters to the right of the *x* prefix. However, if there is only a single global wildcard to the right of *x*, then the original LIKE or GLOB test is disabled. In other words, if the pattern is like this:

```
column LIKE x% column GLOB x*
```

then the original LIKE or GLOB tests are disabled when the virtual terms constrain an index because in that case we know that all of the rows selected by the index will pass the LIKE or GLOB test.

Note that when the right-hand side of a LIKE or GLOB operator is a parameter and the statement is prepared using sqlite3_prepare_v2() or sqlite3_prepare16_v2() then the statement is automatically reparsed and recompiled on the first sqlite3_step() call of each run if the binding to the right-hand side parameter has changed since the previous run. This reparse and recompile is essentially the same action that occurs following a schema change. The recompile is necessary so that the query planner can examine the new value bound to the right-hand side of the LIKE or GLOB operator and determine whether or not to employ the optimization described above.

5.0 The Skip-Scan Optimization

The general rule is that indexes are only useful if there are WHERE-clause constraints on the left-most columns of the index. However, in some cases, SQLite is able to use an index even if the first few columns of the index are omitted from the WHERE clause but later columns are included.

Consider a table such as the following:

```
CREATE TABLE people(
  name TEXT PRIMARY KEY,
  role TEXT NOT NULL,
  height INT NOT NULL, -- in cm
  CHECK( role IN ('student','teacher') )
);
CREATE INDEX people_idx1 ON people(role, height);
```

The people table has one entry for each person in a large organization. Each person is either a "student" or a "teacher", as determined by the "role" field. And we record the height in centimeters of each person. The role and height are indexed. Notice that the left-most column of the index is not very selective - it only contains two possible values.

Now consider a query to find the names of everyone in the organization that is 180cm tall or taller:

```
SELECT name FROM people WHERE height&gt;=180;
```

Because the left-most column of the index does not appear in the WHERE clause of the query, one is tempted to conclude that the index is not usable here. But SQLite is able to use the index. Conceptually, SQLite uses the index as if the query were more like the following:

```
SELECT name FROM people
 WHERE role IN (SELECT DISTINCT role FROM people)
   AND height&gt;=180;
```

Or this:

```
SELECT name FROM people WHERE role='teacher' AND height&gt;=180
UNION ALL
SELECT name FROM people WHERE role='student' AND height&gt;=180;
```

The alternative query formulations shown above are conceptual only. SQLite does not really transform the query. The actual query plan is like this: SQLite locates the first possible value for "role", which it can do by rewinding the "people_idx1" index to the beginning and reading the first record. SQLite stores this first "role" value in an internal variable that we will here call "$role". Then SQLite runs a query like: "SELECT name FROM people WHERE role=$role AND height>=180". This query has an equality constraint on the left-most column of the index and so the index can be used to resolve that query. Once that query is finished, SQLite then uses the "people_idx1" index to locate the next value of the "role" column, using code that is logically similar to "SELECT role FROM people WHERE role>$role LIMIT 1". This new "role" value overwrites the $role variable, and the process repeats until all possible values for "role" have been examined.

We call this kind of index usage a "skip-scan" because the database engine is basically doing a full scan of the index but it optimizes the scan (making it less than "full") by occasionally skipping ahead to the next candidate value.

SQLite might use a skip-scan on an index if it knows that the first one or more columns contain many duplication values. If there are too few duplicates in the left-most columns of the index, then it would be faster to simply step ahead to the next value, and thus do a full table scan, than to do a binary search on an index to locate the next left-column value.

The only way that SQLite can know that the left-most columns of an index have many duplicate is if the ANALYZE command has been run on the database. Without the results of ANALYZE, SQLite has to guess at the "shape" of the data in the table, and the default guess is that there are an average of 10 duplicates for every value in the left-most column of the index. But skip-scan only becomes profitable (it only gets to be faster than a full table scan) when the number of duplicates is about 18 or more. Hence, a skip-scan is never used on a database that has not been analyzed.

6.0 Joins

The ON and USING clauses of an inner join are converted into additional terms of the WHERE clause prior to WHERE clause analysis described above in paragraph 1.0. Thus with SQLite, there is no computational advantage to use the newer SQL92 join syntax over the older SQL89 comma-join syntax. They both end up accomplishing exactly the same thing on inner joins.

For a LEFT OUTER JOIN the situation is more complex. The following two queries are not equivalent:

```
SELECT * FROM tab1 LEFT JOIN tab2 ON tab1.x=tab2.y;
SELECT * FROM tab1 LEFT JOIN tab2 WHERE tab1.x=tab2.y;
```

For an inner join, the two queries above would be identical. But special processing applies to the ON and USING clauses of an OUTER join: specifically, the constraints in an ON or USING clause do not apply if the right table of the join is on a null row, but the constraints do apply in the WHERE clause. The net effect is that putting the ON or USING clause expressions for a LEFT JOIN in the WHERE clause effectively converts the query to an ordinary INNER JOIN - albeit an inner join that runs more slowly.

6.1 Order of tables in a join

The current implementation of SQLite uses only loop joins. That is to say, joins are implemented as nested loops.

The default order of the nested loops in a join is for the left-most table in the FROM clause to form the outer loop and the right-most table to form the inner loop. However, SQLite will nest the loops in a different order if doing so will help it to select better indices.

Inner joins can be freely reordered. However a left outer join is neither commutative nor associative and hence will not be reordered. Inner joins to the left and right of the outer join might be reordered if the optimizer thinks that is advantageous but the outer joins are always evaluated in the order in which they occur.

SQLite treats the CROSS JOIN operator specially. The CROSS JOIN operator is commutative in theory. But SQLite chooses to never reorder tables in a CROSS JOIN. This provides a mechanism by which the programmer can force SQLite to choose a particular loop nesting order.

When selecting the order of tables in a join, SQLite uses an efficient polynomial-time algorithm. Because of this, SQLite is able to plan queries with 50- or 60-way joins in a matter of microseconds.

Join reordering is automatic and usually works well enough that programmers do not have to think about it, especially if ANALYZE has been used to gather statistics about the available indices. But occasionally some hints from the programmer are needed. Consider, for example, the following schema:

```
CREATE TABLE node(
   id INTEGER PRIMARY KEY,
   name TEXT
);
CREATE INDEX node_idx ON node(name);
CREATE TABLE edge(
   orig INTEGER REFERENCES node,
   dest INTEGER REFERENCES node,
   PRIMARY KEY(orig, dest)
);
CREATE INDEX edge_idx ON edge(dest,orig);
```

The schema above defines a directed graph with the ability to store a name at each node. Now consider a query against this schema:

```
SELECT *
  FROM edge AS e,
       node AS n1,
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

This query asks for is all information about edges that go from nodes labeled "alice" to nodes labeled "bob". The query optimizer in SQLite has basically two choices on how to implement this query. (There are actually six different choices, but we will only consider two of them here.) Pseudocode below demonstrating these two choices.

Option 1:

```
foreach n1 where n1.name='alice' do:
  foreach n2 where n2.name='bob' do:
    foreach e where e.orig=n1.id and e.dest=n2.id
      return n1.*, n2.*, e.*
    end
  end
end
```

Option 2:

```
foreach n1 where n1.name='alice' do:
  foreach e where e.orig=n1.id do:
    foreach n2 where n2.id=e.dest and n2.name='bob' do:
      return n1.*, n2.*, e.*
    end
  end
end
```

The same indices are used to speed up every loop in both implementation options. The only difference in these two query plans is the order in which the loops are nested.

So which query plan is better? It turns out that the answer depends on what kind of data is found in the node and edge tables.

Let the number of alice nodes be M and the number of bob nodes be N. Consider two scenarios. In the first scenario, M and N are both 2 but there are thousands of edges on each node. In this case, option 1 is preferred. With option 1, the inner loop checks for the existence of an edge between a pair of nodes and outputs the result if found. But because there are only 2 alice and bob nodes each, the inner loop only has to run 4 times and the query is very quick. Option 2 would take much longer here. The outer loop of option 2 only executes twice, but because there are a large number of edges leaving each alice node, the middle loop has to iterate many thousands of times. It will be much slower. So in the first scenario, we prefer to use option 1.

Now consider the case where M and N are both 3500. Alice nodes are abundant. But suppose each of these nodes is connected by only one or two edges. In this case, option 2 is preferred. With option 2, the outer loop still has to run 3500 times, but the middle loop only runs once or twice for each outer loop and the inner loop will only run once for each middle loop, if at all. So the total number of iterations of the inner loop is around 7000. Option 1, on the other hand, has to run both its outer loop and its middle loop 3500 times each, resulting in 12 million iterations of the middle loop. Thus in the second scenario, option 2 is nearly 2000 times faster than option 1.

So you can see that depending on how the data is structured in the table, either query plan 1 or query plan 2 might be better. Which plan does SQLite choose by default? As of version 3.6.18, without running ANALYZE, SQLite will choose option 2. But if the ANALYZE command is run in order to gather statistics, a different choice might be made if the statistics indicate that the alternative is likely to run faster.

6.2 Manual Control Of Query Plans Using SQLITE_STAT Tables

SQLite provides the ability for advanced programmers to exercise control over the query plan chosen by the optimizer. One method for doing this is to fudge the ANALYZE results in the sqlite_stat1, sqlite_stat3, and/or sqlite_stat4 tables. That approach is not recommended except for the one scenario described in the next paragraph.

For a program that uses an SQLite database as its application file-format, when a new database instance is first created the ANALYZE command is ineffective because the database contain no data from which to gather statistics. In that case, one could construct a large prototype database containing typical data during development and run the ANALYZE command on this prototype database to gather statistics, then save the prototype statistics as part of the application. After deployment, when the application goes to create a new database file, it can run the ANALYZE command in order to create the statistics tables, then copy the precomputed statistics obtained from the prototype database into these new statistics tables. In that way, statistics from large working data sets can be preloaded into newly created application files.

6.3 Manual Control Of Query Plans Using CROSS JOIN

Programmers can force SQLite to use a particular loop nesting order for a join by using the CROSS JOIN operator instead of just JOIN, INNER JOIN, NATURAL JOIN, or a "," join. Though CROSS JOINs are commutative in theory, SQLite chooses to never reorder the tables in a CROSS JOIN. Hence, the left table of a CROSS JOIN will always be in an outer loop relative to the right table.

In the following query, the optimizer is free to reorder the tables of FROM clause anyway it sees fit:

```
SELECT *
  FROM node AS n1,
       edge AS e,
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

But in the following logically equivalent formulation of the same query, the substitution of "CROSS JOIN" for the "," means that the order of tables must be N1, E, N2.

```
SELECT *
  FROM node AS n1 CROSS JOIN
       edge AS e CROSS JOIN
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

In the latter query, the query plan must be option 2. Note that you must use the keyword "CROSS" in order to disable the table reordering optimization; INNER JOIN, NATURAL JOIN, JOIN, and other similar combinations work just like a comma join in that the optimizer is free to reorder tables as it sees fit. (Table reordering is also disabled on an outer join, but that is because outer joins are not associative or commutative. Reordering tables in OUTER JOIN changes the result.)

See "The Fossil NGQP Upgrade Case Study" for another real-world example of using CROSS JOIN to manually control the nesting order of a join. The query planner checklist found later in the same document provides further guidance on manual control of the query planner.

7.0 Choosing between multiple indices

Each table in the FROM clause of a query can use at most one index (except when the OR-clause optimization comes into play) and SQLite strives to use at least one index on each table. Sometimes, two or more indices might be candidates for use on a single table. For example:

```
CREATE TABLE ex2(x,y,z);
CREATE INDEX ex2i1 ON ex2(x);
CREATE INDEX ex2i2 ON ex2(y);
SELECT z FROM ex2 WHERE x=5 AND y=6;
```

For the SELECT statement above, the optimizer can use the ex2i1 index to lookup rows of ex2 that contain x=5 and then test each row against the y=6 term. Or it can use the ex2i2 index to lookup rows of ex2 that contain y=6 then test each of those rows against the x=5 term.

When faced with a choice of two or more indices, SQLite tries to estimate the total amount of work needed to perform the query using each option. It then selects the option that gives the least estimated work.

To help the optimizer get a more accurate estimate of the work involved in using various indices, the user may optionally run the ANALYZE command. The ANALYZE command scans all indices of database where there might be a choice between two or more indices and gathers statistics on the selectiveness of those indices. The statistics gathered by this scan are stored in special database tables names shows names all begin with "**sqlite_stat**". The content of these tables is not updated as the database changes so after making significant changes it might be prudent to rerun ANALYZE. The results of an ANALYZE command are only available to database connections that are opened after the ANALYZE command completes.

The various **sqlite_stat***N* tables contain information on how selective the various indices are. For example, the sqlite_stat1 table might indicate that an equality constraint on column x reduces the search space to 10 rows on average, whereas an equality constraint on column y reduces the search space to 3 rows on average. In that case, SQLite would prefer to use index ex2i2 since that index is more selective.

7.1 Disqualifying WHERE Clause Terms Using Unary-"+"

Terms of the WHERE clause can be manually disqualified for use with indices by prepending a unary `**&lt;big&gt;+&lt;/big&gt;**` operator to the column name. The unary `**&lt;big&gt;+&lt;/big&gt;**` is a no-op and will not generate any byte code in the prepared statement. But the unary `**&lt;big&gt;+&lt;/big&gt;**` operator will prevent the term from constraining an index. So, in the example above, if the query were rewritten as:

```
SELECT z FROM ex2 WHERE +x=5 AND y=6;
```

The `**&lt;big&gt;+&lt;/big&gt;**` operator on the `**&lt;big&gt;x&lt;/big&gt;**` column will prevent that term from constraining an index. This would force the use of the ex2i2 index.

Note that the unary `**&lt;big&gt;+&lt;/big&gt;**` operator also removes type affinity from an expression, and in some cases this can cause subtle changes in the meaning of an expression. In the example above, if column `**&lt;big&gt;x&lt;/big&gt;**` has TEXT affinity then the comparison "x=5" will be done as text. But the `**&lt;big&gt;+&lt;/big&gt;**` operator removes the affinity. So the comparison "+x=5" will compare the text in column `**&lt;big&gt;x&lt;/big&gt;**` with the numeric value 5 and will always be false.

7.2 Range Queries

Consider a slightly different scenario:

```
CREATE TABLE ex2(x,y,z);
CREATE INDEX ex2i1 ON ex2(x);
CREATE INDEX ex2i2 ON ex2(y);
SELECT z FROM ex2 WHERE x BETWEEN 1 AND 100 AND y BETWEEN 1 AND 100;
```

Further suppose that column x contains values spread out between 0 and 1,000,000 and column y contains values that span between 0 and 1,000. In that scenario, the range constraint on column x should reduce the search space by a factor of 10,000 whereas the range constraint on column y should reduce the search space by a factor of only 10. So the ex2i1 index should be preferred.

SQLite will make this determination, but only if it has been compiled with SQLITE_ENABLE_STAT3 or SQLITE_ENABLE_STAT4. The SQLITE_ENABLE_STAT3 and SQLITE_ENABLE_STAT4 options causes the ANALYZE command to collect a histogram of column content in the sqlite_stat3 or sqlite_stat4 tables and to use this histogram to make a better guess at the best query to use for range constraints such as the above. The main difference between STAT3 and STAT4 is that STAT3 records histogram data for only the left-most column of an index whereas STAT4 records histogram data for all columns of an index. For single-column indexes, STAT3 and STAT4 work the same.

The histogram data is only useful if the right-hand side of the constraint is a simple compile-time constant or parameter and not an expression.

Another limitation of the histogram data is that it only applies to the left-most column on an index. Consider this scenario:

```
CREATE TABLE ex3(w,x,y,z);
CREATE INDEX ex3i1 ON ex2(w, x);
CREATE INDEX ex3i2 ON ex2(w, y);
SELECT z FROM ex3 WHERE w=5 AND x BETWEEN 1 AND 100 AND y BETWEEN 1 AND 100;
```

Here the inequalities are on columns x and y which are not the left-most index columns. Hence, the histogram data which is collected no left-most column of indices is useless in helping to choose between the range constraints on columns x and y.

8.0 Covering Indices

When doing an indexed lookup of a row, the usual procedure is to do a binary search on the index to find the index entry, then extract the rowid from the index and use that rowid to do a binary search on the original table. Thus a typical indexed lookup involves two binary searches. If, however, all columns that were to be fetched from the table are already available in the index itself, SQLite will use the values contained in the index and will never look up the original table row. This saves one binary search for each row and can make many queries run twice as fast.

When an index contains all of the data needed for a query and when the original table never needs to be consulted, we call that index a "covering index".

9.0 ORDER BY optimizations

SQLite attempts to use an index to satisfy the ORDER BY clause of a query when possible. When faced with the choice of using an index to satisfy WHERE clause constraints or satisfying an ORDER BY clause, SQLite does the same cost analysis described above and chooses the index that it believes will result in the fastest answer.

SQLite will also attempt to use indices to help satisfy GROUP BY clauses and the DISTINCT keyword. If the nested loops of the join can be arranged such that rows that are equivalent for the GROUP BY or for the DISTINCT are consecutive, then the GROUP BY or DISTINCT logic can determine if the current row is part of the same group or if the current row is distinct simply by comparing the current row to the previous row. This can be much faster than the alternative of comparing each row to all prior rows.

9.1 Partial ORDER BY Via Index

If a query contains an ORDER BY clause with multiple terms, it might be that SQLite can use indices to cause rows to come out in the order of some prefix of the terms in the ORDER BY but that later terms in the ORDER BY are not satisfied. In that case, SQLite does block sorting. Suppose the ORDER BY clause has four terms and the natural order of the query results in rows appearing in order of the first two terms. As each row is output by the query engine and enters the sorter, the outputs in the current row corresponding to the first two terms of the ORDER BY are compared against the previous row. If they have changed, the current sort is finished and output and a new sort is started. This results in a slightly faster sort. But the bigger advantages are that many fewer rows need to be held in memory, reducing memory requirements, and outputs can begin to appear before the core query has run to completion.

10.0 Subquery flattening

When a subquery occurs in the FROM clause of a SELECT, the simplest behavior is to evaluate the subquery into a transient table, then run the outer SELECT against the transient table. But such a plan can be suboptimal since the transient table will not have any indices and the outer query (which is likely a join) will be forced to do a full table scan on the transient table.

To overcome this problem, SQLite attempts to flatten subqueries in the FROM clause of a SELECT. This involves inserting the FROM clause of the subquery into the FROM clause of the outer query and rewriting expressions in the outer query that refer to the result set of the subquery. For example:

```
SELECT a FROM (SELECT x+y AS a FROM t1 WHERE z&lt;100) where="" a=""&gt;5
```

Would be rewritten using query flattening as:

```
SELECT x+y AS a FROM t1 WHERE z&lt;100 and="" a=""&gt;5
```

There is a long list of conditions that must all be met in order for query flattening to occur. Some of the constraints are marked as obsolete by italic text. These extra constraints are retained in the documentation to preserve the numbering of the other constraints.

1. The subquery and the outer query do not both use aggregates.
2. The subquery is not an aggregate or the outer query is not a join.
3. The subquery is not the right operand of a left outer join.
4. The subquery is not DISTINCT.
5. *(Subsumed into constraint 4)*
6. The subquery does not use aggregates or the outer query is not DISTINCT.
7. The subquery has a FROM clause.
8. The subquery does not use LIMIT or the outer query is not a join.
9. The subquery does not use LIMIT or the outer query does not use aggregates.
10. The subquery does not use aggregates or the outer query does not use LIMIT.
11. The subquery and the outer query do not both have ORDER BY clauses.
12. *(Subsumed into constraint 3)*
13. The subquery and outer query do not both use LIMIT.
14. The subquery does not use OFFSET.
15. The outer query is not part of a compound select or the subquery does not have a LIMIT clause.
16. The outer query is not an aggregate or the subquery does not contain ORDER BY.
17. The sub-query is not a compound select, or it is a UNION ALL compound clause made up entirely of non-aggregate queries, and the parent query:
    - is not itself part of a compound select,
    - is not an aggregate or DISTINCT query, and
    - is not a join.The parent and sub-query may contain WHERE clauses. Subject to rules (11), (12) and (13), they may also contain ORDER BY, LIMIT and OFFSET clauses.
18. If the sub-query is a compound select, then all terms of the ORDER by clause of the parent must be simple references to columns of the sub-query.
19. The subquery does not use LIMIT or the outer query does not have a WHERE clause.
20. If the sub-query is a compound select, then it must not use an ORDER BY clause.
21. The subquery does not use LIMIT or the outer query is not DISTINCT.
22. The subquery is not a recursive CTE.
23. The parent is not a recursive CTE, or the sub-query is not a compound query.

The casual reader is not expected to understand or remember any part of the list above. The point of this list is to demonstrate that the decision of whether or not to flatten a query is complex.

Query flattening is an important optimization when views are used as each use of a view is translated into a subquery.

11.0 The MIN/MAX optimization

Queries that contain a single MIN() or MAX() aggregate function whose argument is the left-most column of an index might be satisfied by doing a single index lookup rather than by scanning the entire table. Examples:

```
SELECT MIN(x) FROM table;
SELECT MAX(x)+1 FROM table;
```

12.0 Automatic Indexes

When no indices are available to aid the evaluation of a query, SQLite might create an automatic index that lasts only for the duration of a single SQL statement. Since the cost of constructing the automatic index is O(NlogN) (where N is the number of entries in the table) and the cost of doing a full table scan is only O(N), 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. Consider an example:

```
CREATE TABLE t1(a,b);
CREATE TABLE t2(c,d);
-- Insert many rows into both t1 and t2
SELECT * FROM t1, t2 WHERE a=c;
```

In the query above, if both t1 and t2 have approximately N rows, then without any indices the query will require O(N*N) time. On the other hand, creating an index on table t2 requires O(NlogN) time and then using that index to evaluate the query requires an additional O(NlogN) time. In the absence of ANALYZE information, SQLite guesses that N is one million and hence it believes that constructing the automatic index will be the cheaper approach.

An automatic index might also be used for a subquery:

```
CREATE TABLE t1(a,b);
CREATE TABLE t2(c,d);
-- Insert many rows into both t1 and t2
SELECT a, (SELECT d FROM t2 WHERE c=b) FROM t1;
```

In this example, the t2 table is used in a subquery to translate values of the t1.b column. If each table contains N rows, SQLite expects that the subquery will run N times, and hence it will believe it is faster to construct an automatic, transient index on t2 first and then using that index to satisfy the N instances of the subquery.

The automatic indexing capability can be disabled at run-time using the automatic_index pragma. Automatic indexing is turned on by default, but this can be changed so that automatic indexing is off by default using the SQLITE_DEFAULT_AUTOMATIC_INDEX compile-time option. The ability to create automatic indices can be completely disabled by compiling with the SQLITE_OMIT_AUTOMATIC_INDEX compile-time option.

In SQLite version 3.8.0 and later, an SQLITE_WARNING_AUTOINDEX message is sent to the error log every time a statement is prepared that uses an automatic index. Application developers can and should use these warnings to identify the need for new persistent indices in the schema.

Do not confuse automatic indexes with the internal indexes (having names like "sqlite*autoindex**table**_N*") that are sometimes created to implement a PRIMARY KEY constraint or UNIQUE constraint. The automatic indexes described here exist only for the duration of a single query, are never persisted to disk, and are only visible to a single database connection. Internal indexes are part of the implementation of PRIMARY KEY and UNIQUE constraints, are long-lasting and persisted to disk, and are visible to all database connections. The term "autoindex" appears in the names of internal indexes for legacy reasons and does not indicate that internal indexes and automatic indexes are related.

The Next Generation Query Planner

1.0 Introduction

The task of the "query planner" is to figure out the best algorithm or "query plan" to accomplish an SQL statement. Beginning with SQLite version 3.8.0, the query planner component has been rewritten so that it runs faster and generates better plans. The rewrite is called the "next generation query planner" or "NGQP".

This article overviews the importance of query planning, describes some of the problems inherent to query planning, and outlines how the NGQP solves those problems.

The NGQP is almost always better than the legacy query planner. However, there may exist legacy applications that unknowingly depend on undefined and/or suboptimal behavior in the legacy query planner, and upgrading to the NGQP on those legacy applications could cause performance regressions. This risk is considered and a checklist is provided for reducing the risk and for fixing any issues that do arise.

This document focuses on the NGQP. For a more general overview of the SQLite query planner that encompasses the entire history of SQLite, see "The SQLite Query Optimizer Overview".

2.0 Background

For simple queries against a single table with few indices, there is usually an obvious choice for the best algorithm. But for larger and more complex queries, such as multi-way joins with many indices and subqueries, there can be hundreds, thousands, or millions of reasonable algorithms for computing the result. The job of the query planner is to choose the single "best" query plan from this multitude of possibilities.

Query planners are what make SQL database engines so amazingly useful and powerful. (This is true of all SQL database engines, not just SQLite.) The query planner frees the programmer from the chore of selecting a particular query plan, and thereby allows the programmer to focus more mental energy on higher-level application issues and on providing more value to the end user. For simple queries where the choice of query plan is obvious, this is convenient but not hugely important. But as applications and schemas and queries grow more complex, a clever query planner can greatly speed and simplify the work of application development. There is amazing power in being about to tell the database engine what content is desired, and then let the database engine figure out the best way to retrieve that content.

Writing a good query planner is more art than science. The query planner must work with incomplete information. It cannot determine how long any particular plan will take without actually running that plan. So when comparing two or more plans to figure out which is "best", the query planner has to make some guesses and assumptions and those guesses and assumptions will sometimes be wrong. A good query planner is one that will find the correct solution often enough that application programmers rarely need to get involved.

2.1 Query Planning In SQLite

SQLite computes joins using nested loops, one loop for each table in the join. (Additional loops might be inserted for IN and OR operators in the WHERE clause. SQLite considers those too, but for simplicity we will ignore them in this essay.) One or more indices might be used on each loop to speed the search, or a loop might be a "full table scan" that reads every row in the table. Thus query planning decomposes into two subtasks:

1. Picking the nested order of the various loops
2. Choosing good indices for each loop

Picking the nesting order is generally the more challenging problem. Once the nesting order of the join is established, the choice of indices for each loop is normally obvious.

2.2 The SQLite Query Planner Stability Guarantee

SQLite will always pick the same query plan for any given SQL statement as long as:

1. the database schema does not change in significant ways such as adding or dropping indices,
2. the ANALYZE command is not rerun,
3. SQLite is not compiled with SQLITE_ENABLE_STAT3 or SQLITE_ENABLE_STAT4, and
4. the same version of SQLite is used.

The SQLite stability guarantee means that if all of your queries run efficiently during testing, and if your application does not change the schema, then SQLite will not suddenly decide to start using a different query plan, possibly causing a performance problem, after your application is released to users. If your application works in the lab, it will continue working the same way after deployment.

Enterprise-class client/server SQL database engines do not normally make this guarantee. In client/server SQL database engines, the server keeps track of statistics on the sizes of tables and on the quality of indices and the query planner uses those statistics to help select the best plans. As content is added, deleted, or changed in the database, the statistics will evolve and may cause the query planner to begin using a different query plan for some particular query. Usually the new plan will be better for the evolving structure of the data. But sometimes the new query plan will cause a performance reduction. With a client/server database engine, there is typically a Database Administrator (DBA) on hand to deal with these rare problems as they come up. But DBAs are not available to fix problems in an embedded database like SQLite, and hence SQLite is careful to ensure that plans do not change unexpectedly after deployment.

The SQLite stability guarantee applies equally to the legacy query planner and to the NGQP.

It is important to note that changing versions of SQLite might cause changes in query plans. The same version of SQLite will always pick the same query plan, but if you relink your application to use a different version of SQLite, then query plans might change. In rare cases, an SQLite version change might lead to a performance regression. This is one reason you should consider statically linking your applications against SQLite rather than use a system-wide SQLite shared library which might change without your knowledge or control.

3.0 A Difficult Case

"TPC-H Q8" is a test query from the Transaction Processing Performance Council. The query planners in SQLite versions 3.7.17 and earlier do not choose good plans for TPC-H Q8. And it has been determined that no amount of tweaking of the legacy query planner will fix that. In order to find a good solution to the TPC-H Q8 query, and to continue improving the quality of SQLite's query planner, it became necessary to redesign the query planner. This section tries to explain why this redesign was necessary and how the NGQP is different and addresses the TPC-H Q8 problem.

3.1 Query Details

TPC-H Q8 is an eight-way join. As observed above, the main task of the query planner is to figure out the best nesting order of the eight loops in order to minimize the work needed to complete the join. A simplified model of this problem for the case of TPC-H Q8 is shown by the following diagram:

In the diagram, each of the 8 tables in the FROM clause of the query is identified by a large circle with the label of the FROM-clause term: N2, S, L, P, O, C, N1 and R. The arcs in the graph represent the estimated cost of computing each term assuming that the origin of the arc is in an outer loop. For example, the cost of running the S loop as an inner loop to L is 2.30 whereas the cost of running the S loop as an outer loop to L is 9.17.

The "cost" here is logarithmic. With nested loops, the work is multiplied, not added. But it is customary to think of graphs with additive weights and so the graph shows the logarithm of the various costs. The graph shows a cost advantage of S being inside of L of about 6.87, but this translates into the query running about 963 times faster when S loop is inside of the L loop rather than being outside of it.

The arrows from the small circles labeled with "*" indicate the cost of running each loop with no dependencies. The outer loop must use this* -cost. Inner loops have the option of using the *-cost or a cost assuming one of the other terms is in an outer loop, whichever gives the best result. One can think of the* -costs as a short-hand notation indicating multiple arcs, one from each of the other nodes in the graph. The graph is therefore "complete", meaning that there are arcs (some explicit and some implied) in both directions between every pair of nodes in the graph.

The problem of finding the best query plan is equivalent to finding a minimum-cost path through the graph that visits each node exactly once.

(Side note: The costs estimates in the TPC-H Q8 graph were computed by the query planner in SQLite 3.7.16 and converted using a natural logarithm.)

3.2 Complications

The presentation of the query planner problem above is a simplification. The costs are estimates. We cannot know what the true cost of running a loop is until we actually run the loop. SQLite makes guesses for the cost of running a loop based on the availability of indices and constraints found in the WHERE clause. These guesses are usually pretty good, but they can sometimes be off. Using the ANALYZE command to collect additional statistical information about the database can sometimes enable SQLite to make better guesses about the cost.

The costs are comprised of multiple numbers, not a single number as shown in the graph. SQLite computes several different estimated costs for each loop that apply at different times. For example, there is a "setup" cost that is incurred just once when the query starts. The setup cost is the cost of computing an automatic index for a table that does not already have an index. Then there is the cost of running each step of the loop. Finally, there is an estimate of the number rows generated by the loop, which is information needed in estimating the costs of inner loops. Sorting costs may come into play if the query has an ORDER BY clause.

In a general query, dependencies need not be on a single loop, and hence the matrix of dependencies might not be representable as a graph. For example, one of the WHERE clause constraints might be S.a=L.b+P.c, implying that the S loop must be an inner loop of both L and P. Such dependencies cannot be drawn as a graph since there is no way for an arc to originate at two or more nodes at once.

If the query contains an ORDER BY clause or a GROUP BY clause or if the query uses the DISTINCT keyword then it is advantageous to select a path through the graph that causes rows to naturally appear in sorted order, so that no separate sorting step is required. Automatic elimination of ORDER BY clauses can make a large performance difference, so this is another factor that needs to be considered in a complete implementation.

In the TPC-H Q8 query, the setup costs are all negligible, all dependencies are between individual nodes, and there is no ORDER BY, GROUP BY, or DISTINCT clause. So for TPC-H Q8, the graph above is a reasonable representation of what needs to be computed. The general case involves a lot of extra complication, which for clarity is neglected in the remainder of this article.

3.3 Finding The Best Query Plan

Prior to version 3.8.0, SQLite always used the "Nearest Neighbor" or "NN" heuristic when searching for the best query plan. The NN heuristic makes a single traversal of the graph, always choosing the lowest-cost arc as the next step. The NN heuristic works surprisingly well in most cases. And NN is fast, so that SQLite is able to quickly find good plans for even large 64-way joins. In contrast, other SQL database engines that do more extensive searching tend to bog down when the number of tables in a join goes above 10 or 15.

Unfortunately, the query plan computed by NN for TPC-H Q8 is not optimal. The plan computed using NN is R-N1-N2-S-C-O-L-P with a cost of 36.92. The notation in the previous sentence means that the R table is run in the outer loop, N1 is in the next inner loop, N2 is in the third loop, and so forth down to P which is in the inner-most loop. The shortest path through the graph (as found via exhaustive search) is P-L-O-C-N1-R-S-N2 with a cost of 27.38. The difference might not seem like much, but remember that the costs are logarithmic, so the shortest path is nearly 750 times faster than that path found using the NN heuristic.

One solution to this problem is to change SQLite to do an exhaustive search for the best path. But an exhaustive search requires time proportional to K! (where K is the number of tables in the join) and so when you get beyond a 10-way join, the time to run sqlite3_prepare() becomes very large.

3.4 The N Nearest Neighbors or "N3" Heuristic

The NGQP uses a new heuristic for seeking the best path through the graph: "N Nearest Neighbors" (hereafter "N3"). With N3, instead of choosing just one nearest neighbor for each step, the algorithm keeps track of the N bests paths at each step for some small integer N.

Suppose N=4. Then for the TPC-H Q8 graph, the first step finds the four shortest paths to visit any single node in the graph:

> R (cost: 3.56) N1 (cost: 5.52) N2 (cost: 5.52) P (cost: 7.71)

The second step finds the four shortest paths to visit two nodes beginning with one of the four paths from the previous step. In the case where two or more paths are equivalent (they have the same set of visited nodes, though possibly in a different order) only the first and lowest-cost path is retained. We have:

> R-N1 (cost: 7.03) R-N2 (cost: 9.08) N2-N1 (cost: 11.04) R-P {cost: 11.27}

The third step starts with the four shortest two-node paths and finds the four shortest three-node paths:

> R-N1-N2 (cost: 12.55) R-N1-C (cost: 13.43) R-N1-P (cost: 14.74) R-N2-S (cost: 15.08)

And so forth. There are 8 nodes in the TPC-H Q8 query, so this process repeats a total of 8 times. In the general case of a K-way join, the storage requirement is O(N) and the computation time is O(K*N), which is significantly faster than the O(2<small><sup>K</sup></small>) exact solution.

But what value to choose for N? One might try N=K. This makes the algorithm O(K<small><sup>2</sup></small>) which is actually still quite efficient, since the maximum value of K is 64 and K rarely exceeds 10. But that is not enough for the TPC-H Q8 problem. With N=8 on TPC-H Q8 the N3 algorithm finds the solution R-N1-C-O-L-S-N2-P with a cost of 29.78. That is a big improvement over NN, but it is still not optimal. N3 finds the optimal solution for TPC-H Q8 when N is 10 or greater.

The initial implementation of NGQP chooses N=1 for simple queries, N=5 for two-way joins and N=10 for all joins with three or more tables. This formula for selecting N might change in subsequent releases.

4.0 Hazards Of Upgrading To NGQP

For most applications, upgrading from the legacy query planner to the NGQP requires little thought or effort. Simply replace the older SQLite version with the newer version of SQLite and recompile and the application will run faster. There are no API changes nor modifications to compilation procedures.

But as with any query planner change, upgrading to the NGQP does carry a small risk of introducing performance regressions. The problem here is not that the NGQP is incorrect or buggy or inferior to the legacy query planner. Given reliable information about the selectivity of indices, the NGQP should always pick a plan that is as good or better than before. The problem is that some applications may be using low-quality and low-selectivity indices without having run ANALYZE. The older query planners look at many fewer possible implementations for each query and so they may have stumbled over a good plan by stupid luck. The NGQP, on the other hand, looks at many more query plan possibilities, and it may choose a different query plan that works better in theory, assuming good indices, but which gives a performance regression in practice, because of the shape of the data.

Key points:

- The NGQP will always find an equal or better query plan, compared to prior query planners, as long as it has access to accurate ANALYZE data in the SQLITE_STAT1 file.
- The NGQP will always find a good query plan as long as the schema does not contain indices that have more than about 10 or 20 rows with the same value in the left-most column of the index.

Not all applications meet these conditions. Fortunately, the NGQP will still usually find good query plans, even without these conditions. However, cases do arise (rarely) where performance regressions can occur.

4.1 Case Study: Upgrading Fossil to the NGQP

The Fossil DVCS is the version control system used to track all of the SQLite source code. A Fossil repository is an SQLite database file. (Readers are invited to ponder this recursion as an independent exercise.) Fossil is both the version-control system for SQLite and a test platform for SQLite. Whenever enhancements are made to SQLite, Fossil is one of the first applications to test and evaluate those enhancements. So Fossil was an early adopter of the NGQP.

Unfortunately, the NGQP caused a performance regression in Fossil.

One of the many reports that Fossil makes available is a timeline of changes to a single branch showing all merges in and out of that branch. See http://www.sqlite.org/src/timeline?nd&n=200&r=trunk for a typical example of such a report. Generating such a report normally takes just a few milliseconds. But after upgrading to the NGQP we noticed that this one report was taking closer to 10 seconds for the trunk of the repository.

The core query used to generate the branch timeline is shown below. (Readers are not expected to understand the details of this query. Commentary will follow.)

```
SELECT
     blob.rid AS blobRid,
     uuid AS uuid,
     datetime(event.mtime,'localtime') AS timestamp,
     coalesce(ecomment, comment) AS comment,
     coalesce(euser, user) AS user,
     blob.rid IN leaf AS leaf,
     bgcolor AS bgColor,
     event.type AS eventType,
     (SELECT group_concat(substr(tagname,5), ', ')
        FROM tag, tagxref
       WHERE tagname GLOB 'sym-*'
         AND tag.tagid=tagxref.tagid
         AND tagxref.rid=blob.rid
         AND tagxref.tagtype&gt;0) AS tags,
     tagid AS tagid,
     brief AS brief,
     event.mtime AS mtime
  FROM event CROSS JOIN blob
 WHERE blob.rid=event.objid
   AND (EXISTS(SELECT 1 FROM tagxref
                WHERE tagid=11 AND tagtype&gt;0 AND rid=blob.rid)
        OR EXISTS(SELECT 1 FROM plink JOIN tagxref ON rid=cid
                   WHERE tagid=11 AND tagtype&gt;0 AND pid=blob.rid)
        OR EXISTS(SELECT 1 FROM plink JOIN tagxref ON rid=pid
                   WHERE tagid=11 AND tagtype&gt;0 AND cid=blob.rid))
 ORDER BY event.mtime DESC
 LIMIT 200;
```

This query is not especially complicated, but even so it replaces hundreds or perhaps thousands of lines of procedural code. The gist of the query is this: Scan down the EVENT table looking for the most recent 200 check-ins that satisfy any one of three conditions:

1. The check-in has a "trunk" tag.
2. The check-in has a child that has a "trunk" tag.
3. The check-in has a parent that has a "trunk" tag.

The first condition causes all of the trunk check-ins to be displayed and the second and third cause check-ins that merge into or fork from the trunk to also be included. The three conditions are implemented by the three OR-connected EXISTS statements in the WHERE clause of the query. The slowdown that occurred with the NGQP was caused by the second and third conditions. The problem is the same in each, so we will examine just the second one. The subquery of the second condition can be rewritten (with minor and immaterial simplifications) as follows:

```
SELECT 1
  FROM plink JOIN tagxref ON tagxref.rid=plink.cid
 WHERE tagxref.tagid=$trunk
   AND plink.pid=$ckid;
```

The PLINK table holds parent-child relationships between check-ins. The TAGXREF table maps tags into check-ins. For reference, the relevant portions of the schemas for these two tables is shown here:

```
CREATE TABLE plink(
  pid INTEGER REFERENCES blob,
  cid INTEGER REFERENCES blob
);
CREATE UNIQUE INDEX plink_i1 ON plink(pid,cid);

CREATE TABLE tagxref(
  tagid INTEGER REFERENCES tag,
  mtime TIMESTAMP,
  rid INTEGER REFERENCE blob,
  UNIQUE(rid, tagid)
);
CREATE INDEX tagxref_i1 ON tagxref(tagid, mtime);
```

There are only two reasonable ways to implement this query. (There are many other possible algorithms, but none of the others are contenders for being the "best" algorithm.)

1. Find all children of check-in $ckid and test each one to see if it has the $trunk tag.
2. Find all check-ins with the $trunk tag and test each one to see if it is a child of $ckid.

Intuitively, we humans understand that algorithm-1 is best. Each check-in is likely to have few children (one child is the most common case) and each child can be tested for the $trunk tag in logarithmic time. Indeed, algorithm-1 is the faster choice in practice. But the NGQP has no intuition. The NGQP must use hard math, and algorithm-2 is slightly better mathematically. This is because, in the absence of other information, the NGQP must assume that the indices PLINK_I1 and TAGXREF_I1 are of equal quality and are equally selective. Algorithm-2 uses one field of the TAGXREF_I1 index and both fields of the PLINK_I1 index whereas algorithm-1 only uses the first field of each index. Since algorithm-2 uses more index material, the NGQP is correct to judge it to be the better algorithm. The scores are close and algorithm-2 just barely squeaks ahead of algorithm-1. But algorithm-2 really is the correct choice here.

Unfortunately, algorithm-2 is slower than algorithm-1 in this application.

The problem is that the indices are not of equal quality. A check-in is likely to only have one child. So the first field of PLINK_I1 will usually narrow down the search to just a single row. But there are thousands and thousands check-ins tagged with "trunk", so the first field of TAGXREF_I1 will be of little help in narrowing down the search.

The NGQP has no way of knowing that TAGXREF_I1 is almost useless in this query, unless ANALYZE has been run on the database. The ANALYZE command gathers statistics on the quality of the various indices and stores those statistics in SQLITE_STAT1 table. Having access to this statistical information, the NGQP easily chooses algorithm-1 as the best algorithm, by a wide margin.

Why didn't the legacy query planner choose algorithm-2? Easy: because the NN algorithm never even considered algorithm-2. Graphs of the planning problem look like this:

In the "without ANALYZE" case on the left, the NN algorithm chooses loop P (PLINK) as the outer loop because 4.9 is less than 5.2, resulting in path P-T which is algorithm-1. NN only looks at the single best choice at each step so it completely misses the fact that 5.2+4.4 makes a slightly cheaper plan than 4.9+4.8. But the N3 algorithm keeps track of the 5 best paths for a 2-way join, so it ends up selecting path T-P because of its slightly lower overall cost. Path T-P is algorithm-2.

Note that with ANALYZE the cost estimates are better aligned with reality and algorithm-1 is selected by both NN and N3.

(Side note: The costs estimates in the two most recent graphs were computed by the NGQP using a base-2 logarithm and slightly different cost assumptions compared to the legacy query planner. Hence, the cost estimates in these latter two graphs are not directly comparable to the cost estimates in the TPC-H Q8 graph.)

4.2 Fixing The Problem

Running ANALYZE on the repository database immediately fixed the performance problem. However, we want Fossil to be robust and to always work quickly regardless of whether or not its repository has been analyzed. For this reason, the query was modified to use the CROSS JOIN operator instead of the plain JOIN operator. SQLite will not reorder the tables of a CROSS JOIN. This is a long-standing feature of SQLite that is specifically designed to allow knowledgeable programmers to enforce a particular loop nesting order. Once the join was changed to CROSS JOIN (the addition of a single keyword) the NGQP was forced to choose the faster algorithm-1 regardless of whether or not statistical information had been gathered using ANALYZE.

We say that algorithm-1 is "faster", but this is not strictly true. Algorithm-1 is faster in common repositories, but it is possible to construct a repository in which every check-in is on a different uniquely-named branch and all check-ins are children of the root check-in. In that case, TAGXREF_I1 would become more selective than PLINK_I1 and algorithm-2 really would be the faster choice. However such repositories are very unlikely to appear in practice and so hard-coding the loop nested order using the CROSS JOIN syntax is a reasonable solution to the problem in this case.

5.0 Checklist For Avoiding Or Fixing Query Planner Problems

1. **Don't panic!** Cases where the query planner picks an inferior plan are actually quite rare. You are unlikely to run across any problems in your application. If you are not having performance issues, you do not need to worry about any of this.

2. **Create appropriate indices.** Most SQL performance problems arise not because of query planner issues but rather due to lack of appropriate indices. Make sure indices are available to assist all large queries. Most performance issues can be resolved by one or two CREATE INDEX commands and with no changes to application code.

3. **Avoid creating low-quality indices.**. A low-quality index (for the purpose of this checklist) is one where there are more than 10 or 20 rows in the table that have the same value for the left-most column of the index. In particular, avoid using boolean or "enum" columns as the left-most columns of your indices.

   The Fossil performance problem described in the previous section of this document arose because there were over ten-thousand entries in the TAGXREF table with the same value for the left-most column (the TAGID column) of the TAGXREF_I1 index.

4. **If you must use a low-quality index, be sure to run ANALYZE.** Low-quality indices will not confuse the query planner as long as the query planner knows that the indices are of low quality. And the way the query planner knows this is by the content of the SQLITE_STAT1 table, which is computed by the ANALYZE command.

   Of course, ANALYZE only works effectively if you have a significant amount of content in your database in the first place. When creating a new database that you expect to accumulate a lot of data, you can run the command "ANALYZE sqlite_master" to create the SQLITE_STAT1 table, then prepopulate the SQLITE_STAT1 table (using ordinary INSERT statements) with content that describes a typical database for your application - perhaps content that you extracted after running ANALYZE on a well-populated template database in the lab.

5. **Instrument your code.** Add logic that lets you know quickly and easily which queries are taking too much time. Then work on just those specific queries.

6. **Use unlikely() and likelihood() SQL functions.** SQLite normally assumes that terms in the WHERE clause that cannot be used by indices have a strong probability of being true. If this assumption is incorrect, it could lead to a suboptimal query plan. The unlikely() and likelihood() SQL functions can be used to provide hints to the query planner about WHERE clause terms that are probably not true, and thus aid the query planner in selecting the best possible plan.

7. **Use the CROSS JOIN syntax to enforce a particular loop nesting order on queries that might use low-quality indices in an unanalyzed database.** SQLite treats the CROSS JOIN operator specially, forcing the table to the left to be an outer loop relative to the table on the right.

   Avoid this step if possible, as it defeats one of the huge advantages of the whole SQL language concept, specifically that the application programmer does not need to get involved with query planning. If you do use CROSS JOIN, wait until late in your development cycle to do so, and comment the use of CROSS JOIN carefully so that you can take it out later if possible. Avoid using CROSS JOIN early in the development cycle as doing so is a premature optimization, which is well known to be the root of all evil.

8. **Use unary "+" operators to disqualify WHERE clause terms.** If the query planner insists on selecting a poor-quality index for a particular query when a much higher-quality index is available, then careful use of unary "+" operators in the WHERE clause can force the query planner away from the poor-quality index. Avoid using this trick if at all possible, and especially avoid it early in the application development cycle. Beware that adding a unary "+" operator to an equality expression might change the result of that expression if type affinity is involved.

9. **Use the INDEXED BY syntax to enforce the selection of particular indices on problem queries.** As with the previous two bullets, avoid this step if possible, and especially avoid doing this early in development as it is clearly a premature optimization.

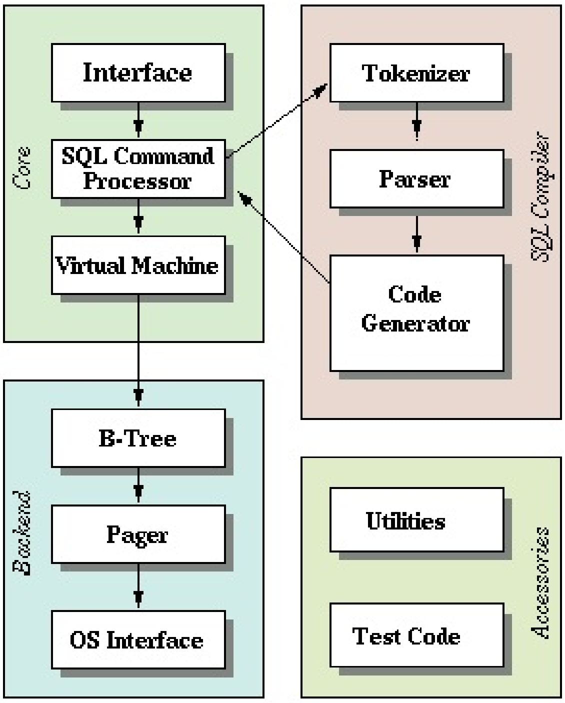
6.0 Summary

The query planner in SQLite normally does a terrific job of selecting fast algorithms for running your SQL statements. This is true of the legacy query planner and even more true of the new NGQP. There may be an occasional situation where, due to incomplete information, the query planner selects a suboptimal plan. This will happen less often with the NGQP than with the legacy query planner, but it might still happen. Only in those rare cases do application developers need to get involved and help the query planner to do the right thing. In the common case, the NGQP is just a new enhancement to SQLite that makes the application run a little faster and which requires no new developer thought or action.

The Architecture Of SQLite

Introduction

Block Diagram Of SQLite



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 block diagram showing the main components of SQLite and how they interrelate is shown at the right. The text that follows will provide a quick overview of each of these components.

This document describes SQLite version 3.0. Version 2.8 and earlier are similar but the details differ.

Interface

Much of the public interface to the SQLite library is implemented by functions found in the **main.c**, **legacy.c**, and **vdbeapi.c** source files 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**. **sqlite3_mprintf()** is found in **printf.c**. **sqlite3_complete()** is in **tokenize.c**. The Tcl interface is implemented by **tclsqlite.c**. More information on the C interface to SQLite is available separately.

To avoid name collisions with other software, 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) begin with **sqlite3_**.

Tokenizer

When a string containing SQL statements is to be executed, the interface passes that string to the tokenizer. The job of the tokenizer is to break the original string up into tokens and pass those tokens one by one to the parser. The tokenizer is hand-coded in C 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 used to doing things the other way around -- having the parser call the tokenizer. The author of SQLite has done it both ways and finds things generally work out nicer for the tokenizer to call the parser. YACC has it backwards.

Parser

The parser is the piece that 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 source file that drives Lemon 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. Documentation on lemon is found in the "doc" subdirectory of the distribution.

Code Generator

After the parser assembles tokens into complete SQL statements, it calls the code generator to produce virtual machine code that will do the work that the SQL statements request. There are many files in the code generator: **attach.c**, **auth.c**, **build.c**, **delete.c**, **expr.c**, **insert.c**, **pragma.c**, **select.c**, **trigger.c**, **update.c**, **vacuum.c** and **where.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()**.

Virtual Machine

The program generated by the code generator is executed by the virtual machine. Additional information about the virtual machine is available separately. To summarize, the virtual machine implements an abstract computing engine specifically designed to manipulate database files. The machine has a stack which is used for intermediate storage. Each instruction contains an opcode and up to three additional operands.

The virtual machine itself is entirely contained in a single source file **vdbe.c**. The virtual machine also has its own header files: **vdbe.h** that defines an interface between the virtual machine and the rest of the SQLite library and **vdbeInt.h** which defines structure private 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*...** family of functions. 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: **coalesce()**, **count()**, **substr()**, and so forth) can be found in **func.c**. Date and time conversion functions are found in **date.c**.

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. Details of the file format are recorded in a large comment at the beginning of **btree.c**.

The interface to the B-tree subsystem is defined by the header file **btree.h**.

Page Cache

The B-tree module requests information from the disk in fixed-size chunks. The default chunk size is 1024 bytes but can vary between 512 and 65536 bytes. The page cache is responsible for reading, writing, and caching these chunks. 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 code to implement the page cache is contained in the single C source file **pager.c**. The interface to the page cache subsystem is defined by the header file **pager.h**.

OS Interface

In order to provide portability between POSIX and Win32 operating systems, SQLite uses an abstraction layer to interface with the operating system. The interface to the OS abstraction layer is defined in **os.h**. Each supported operating system has its own implementation: **os_unix.c** for Unix, **os_win.c** for Windows, and so forth. Each of these operating-specific implements typically has its own header file: **os_unix.h**, **os_win.h**, etc.

Utilities

Memory allocation and caseless string comparison routines 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 random number generator in **random.c**.

Test Code

If you count regression test scripts, more than half the total code base of SQLite is devoted to testing. There are many **assert()** statements in the main code files. In additional, the source files **test1.c** through **test5.c** together with **md5.c** implement extensions used for testing purposes only. The **os_test.c** backend interface is used to simulate power failures to verify the crash-recovery mechanism in the pager.

The SQLite Virtual Machine

Introduction

In order to execute an SQL statement, the SQLite library first parses the SQL, analyzes the statement, then generates a short program to execute the statement. The program is generated for a "virtual machine" implemented by the SQLite library. That virtual machine is sometimes called the "VDBE" or "Virtual DataBase Engine". This document describes the operation of the VDBE.

This document is intended as a reference, not a tutorial. A separate Virtual Machine Tutorial is available. If you are looking for a narrative description of how the virtual machine works, you should read the tutorial and not this document. Once you have a basic idea of what the virtual machine does, you can refer back to this document for the details on a particular opcode. Unfortunately, the virtual machine tutorial was written for SQLite version 1.0. There are substantial changes in the virtual machine for version 2.0 and again for version 3.0.0 and again for version 3.5.5 and the tutorial document has not been updated. But the basic concepts behind the virtual machine still apply.

The source code to the virtual machine is in the vdbe.c source file. All of the opcode definitions further down in this document are contained in comments in the source file. In fact, the opcode table in this document was generated by scanning the vdbe.c source file and extracting the necessary information from comments. So the source code comments are really the canonical source of information about the virtual machine. When in doubt, refer to the source code.

Each instruction in the virtual machine consists of an opcode and up to five operands named P1, P2 P3, P4, and P5. The P1, P2, and P3 operands are 32-bit signed integers. These operands often refer to registers but can also be used for other purposes. The P1 operand is usually the cursor number for opcodes that operate on cursors. P2 is usually the jump destination jump instructions. P4 may be a 32-bit signed integer, a 64-bit signed integer, a 64-bit floating point value, a string literal, a Blob literal, a pointer to a collating sequence comparison function, or a pointer to the implementation of an application-defined SQL function, or various other things. P5 is an unsigned character normally used as a flag. Some operators use all five operands. Some use one or two. Some operators use none of the operands.

The virtual machine begins execution on instruction number 0. Execution continues until a Halt instruction is seen, or until the program counter becomes one greater than the address of last instruction, or there is an execution error. When the virtual machine halts, all memory that it allocated is released and all database cursors it may have had open are closed. If the execution stopped due to an error, any pending transactions are terminated and changes made to the database are rolled back.

The virtual machine can have zero or more cursors. Each cursor is a pointer into a single table or index within the database. There can be multiple cursors pointing at the same index or table. All cursors operate independently, even cursors pointing to the same indices or tables. The only way for the virtual machine to interact with a database file is through a cursor. Instructions in the virtual machine can create a new cursor (OpenRead or OpenWrite), read data from a cursor (Column), advance the cursor to the next entry in the table (Next) or index (NextIdx), and many other operations. All cursors are automatically closed when the virtual machine terminates.

The virtual machine contains an arbitrary number of registers with addresses beginning at one and growing upward. Each register can hold a single SQL value (a string, a BLOB, a signed 64-bit integer, a 64-bit floating point number, or a NULL). A register might also hold objects used internally by SQLite, such as a RowSet or Frame.

Viewing Programs Generated By SQLite

Every SQL statement that SQLite interprets results in a program for the virtual machine. But if you precede the SQL statement with the keyword EXPLAIN the virtual machine will not execute the program. Instead, the instructions of the program will be returned like a query result. This feature is useful for debugging and for learning how the virtual machine operates. For example:

```
$ sqlite3 ex1.db
sqlite> .explain
sqlite> explain delete from tbl1 where two&lt;20;
addr  opcode         p1    p2    p3    p4             p5  comment      
----  -------------  ----  ----  ----  -------------  --  -------------
0     Trace          0     0     0                    00               
1     Goto           0     23    0                    00               
2     Null           0     1     0                    00  r[1]=NULL    
3     OpenRead       0     2     0     2              00  root=2 iDb=0; tbl1
4     Explain        0     0     0     SCAN TABLE tbl1  00
5     Noop           0     0     0                    00  Begin WHERE-loop0: tbl1
6     Rewind         0     14    0                    00               
7       Column         0     1     2                    00  r[2]=tbl1.two
8       Ge             3     13    2     (BINARY)       6a  if r[3]&gt;=r[2] goto 13
9       Noop           0     0     0                    00  Begin WHERE-core
10      Rowid          0     4     0                    00  r[4]=rowid   
11      RowSetAdd      1     4     0                    00  rowset(1)=r[4]
12      Noop           0     0     0                    00  End WHERE-core
13    Next           0     7     0                    01               
14    Noop           0     0     0                    00  End WHERE-loop0: tbl1
15    Close          0     0     0                    00               
16    OpenWrite      0     2     0     3              00  root=2 iDb=0; tbl1
17      RowSetRead     1     21    4                    00  r[4]=rowset(1)
18      NotExists      0     20    4     1              00  intkey=r[4]  
19      Delete         0     1     0     tbl1           00               
20    Goto           0     17    0                    00               
21    Close          0     0     0                    00               
22    Halt           0     0     0                    00               
23    Transaction    0     1     0                    00               
24    VerifyCookie   0     1     0                    00               
25    TableLock      0     2     1     tbl1           00  iDb=0 root=2 write=1
26    Integer        20    3     0                    00  r[3]=20      
27    Goto           0     2     0                    00
```

All you have to do is add the EXPLAIN keyword to the front of the SQL statement. But if you use the ".explain" command in the CLI, it will set up the output mode to make the VDBE code easier for humans to read.

When SQLite is compiled with the SQLITE_DEBUG compile-time option, extra PRAGMA commands are available that are useful for debugging and for exploring the operation of the VDBE. For example the vdbe_trace pragma can be enabled to cause a disassembly of each VDBE opcode to be printed on standard output as the opcode is executed. These debugging pragmas include:

- PRAGMA parser_trace
- PRAGMA vdbe_addoptrace
- PRAGMA vdbe_debug
- PRAGMA vdbe_listing
- PRAGMA vdbe_trace

The Opcodes

There are currently 158 opcodes defined by the virtual machine. All currently defined opcodes are described in the table below. This table was generated automatically by scanning the source code from the file vdbe.c.

Remember: The VDBE opcodes are not part of the interface definition for SQLite. The number of opcodes and their names and meanings are subject to change from one release of SQLite to the next.

Opcode Name	Description
Add	Add the value in register P1 to the value in register P2 and store the result in register P3. If either input is NULL, the result is NULL.
AddImm	Add the constant P2 to the value in register P1. The result is always an integer.To force any register to be an integer, just add 0.
Affinity	Apply affinities to a range of P2 registers starting with P1.P4 is a string that is P2 characters long. The nth character of the string indicates the column affinity that should be used for the nth memory cell in the range.
AggFinal	Execute the finalizer function for an aggregate. P1 is the memory location that is the accumulator for the aggregate.P2 is the number of arguments that the step function takes and P4 is a pointer to the FuncDef for this function. The P2 argument is not used by this opcode. It is only there to disambiguate functions that can take varying numbers of arguments. The P4 argument is only needed for the degenerate case where the step function was not previously called.
AggStep	Execute the step function for an aggregate. The function has P5 arguments. P4 is a pointer to an sqlite3_context object that is used to run the function. Register P3 is as the accumulator.The P5 arguments are taken from register P2 and its successors.This opcode is initially coded as AggStep0. On first evaluation, the FuncDef stored in P4 is converted into an sqlite3_context and the opcode is changed. In this way, the initialization of the sqlite3_context only happens once, instead of on each call to the step function.
AggStep0	Execute the step function for an aggregate. The function has P5 arguments. P4 is a pointer to the FuncDef structure that specifies the function. Register P3 is the accumulator.The P5 arguments are taken from register P2 and its successors.
And	Take the logical AND of the values in registers P1 and P2 and write the result into register P3.If either P1 or P2 is 0 (false) then the result is 0 even if the other input is NULL. A NULL and true or two NULLs give a NULL output.
AutoCommit	Set the database auto-commit flag to P1 (1 or 0). If P2 is true, roll back any currently active btree transactions. If there are any active VMs (apart from this one), then a ROLLBACK fails. A COMMIT fails if there are active writing VMs or active VMs that use shared

	cache.This instruction causes the VM to halt.
BitAnd	Take the bit-wise AND of the values in register P1 and P2 and store the result in register P3. If either input is NULL, the result is NULL.
BitNot	Interpret the content of register P1 as an integer. Store the ones-complement of the P1 value into register P2. If P1 holds a NULL then store a NULL in P2.
BitOr	Take the bit-wise OR of the values in register P1 and P2 and store the result in register P3. If either input is NULL, the result is NULL.
Blob	P4 points to a blob of data P1 bytes long. Store this blob in register P2.
Cast	Force the value in register P1 to be the type defined by P2.<ul> <li value="97"> TEXT <li value="98"> BLOB <li value="99"> NUMERIC <li value="100"> INTEGER <li value="101"> REAL </ul>A NULL value is not changed by this routine. It remains NULL.
Checkpoint	Checkpoint database P1. This is a no-op if P1 is not currently in WAL mode. Parameter P2 is one of SQLITE_CHECKPOINT_PASSIVE, FULL, RESTART, or TRUNCATE. Write 1 or 0 into mem[P3] if the checkpoint returns SQLITE_BUSY or not, respectively. Write the number of pages in the WAL after the checkpoint into mem[P3+1] and the number of pages in the WAL that have been checkpointed after the checkpoint completes into mem[P3+2]. However on an error, mem[P3+1] and mem[P3+2] are initialized to -1.
Clear	Delete all contents of the database table or index whose root page in the database file is given by P1. But, unlike Destroy, do not remove the table or index from the database file.The table being clear is in the main database file if P2==0. If P2==1 then the table to be clear is in the auxiliary database file that is used to store tables create using CREATE TEMPORARY TABLE.If the P3 value is non-zero, then the table referred to must be an intkey table (an SQL table, not an index). In this case the row change count is incremented by the number of rows in the table being cleared. If P3 is greater than zero, then the value stored in register P3 is also incremented by the number of rows in the table being cleared.See also: Destroy
Close	Close a cursor previously opened as P1. If P1 is not currently open, this instruction is a no-op.
CollSeq	P4 is a pointer to a CollSeq struct. If the next call to a user function or aggregate calls sqlite3GetFuncCollSeq(), this collation sequence will be returned. This is used by the built-in min(), max() and nullif() functions.If P1 is not zero, then it is a register that a subsequent min() or max() aggregate will set to 1 if the current row is not the minimum or maximum. The P1 register is initialized to 0 by this instruction.The interface used by the implementation of the aforementioned functions to retrieve the collation sequence set by this opcode is not available publicly. Only built-in functions have access to this feature.

Column	Interpret the data that cursor P1 points to as a structure built using the MakeRecord instruction. (See the MakeRecord opcode for additional information about the format of the data.) Extract the P2-th column from this record. If there are less that (P2+1) values in the record, extract a NULL.The value extracted is stored in register P3.If the column contains fewer than P2 fields, then extract a NULL. Or, if the P4 argument is a P4_MEM use the value of the P4 argument as the result.If the OPFLAG_CLEARCACHE bit is set on P5 and P1 is a pseudo-table cursor, then the cache of the cursor is reset prior to extracting the column. The first Column against a pseudo-table after the value of the content register has changed should have this bit set.If the OPFLAG_LENGTHARG and OPFLAG_TYPEOFARG bits are set on P5 when the result is guaranteed to only be used as the argument of a length() or typeof() function, respectively. The loading of large blobs can be skipped for length() and all content loading can be skipped for typeof().
ColumnsUsed	This opcode (which only exists if SQLite was compiled with SQLITE_ENABLE_COLUMN_USED_MASK) identifies which columns of the table or index for cursor P1 are used. P4 is a 64-bit integer (P4_INT64) in which the first 63 bits are one for each of the first 63 columns of the table or index that are actually used by the cursor. The high-order bit is set if any column after the 64th is used.
Compare	Compare two vectors of registers in reg(P1)..reg(P1+P3-1) (call this vector "A") and in reg(P2)..reg(P2+P3-1) ("B"). Save the result of the comparison for use by the next Jump instruct.If P5 has the OPFLAG_PERMUTE bit set, then the order of comparison is determined by the most recent Permutation operator. If the OPFLAG_PERMUTE bit is clear, then register are compared in sequential order.P4 is a KeyInfo structure that defines collating sequences and sort orders for the comparison. The permutation applies to registers only. The KeyInfo elements are used sequentially.The comparison is a sort comparison, so NULLs compare equal, NULLs are less than numbers, numbers are less than strings, and strings are less than blobs.
Concat	Add the text in register P1 onto the end of the text in register P2 and store the result in register P3. If either the P1 or P2 text are NULL then store NULL in P3.P3 = P2 || P1It is illegal for P1 and P3 to be the same register. Sometimes, if P3 is the same register as P2, the implementation is able to avoid a memcpy().
Copy	Make a copy of registers P1..P1+P3 into registers P2..P2+P3.This instruction makes a deep copy of the value. A duplicate is made of any string or blob constant. See also SCopy.
Count	Store the number of entries (an integer value) in the table or index opened by cursor P1 in register P2
CreateIndex	Allocate a new index in the main database file if P1==0 or in the auxiliary database file if P1==1 or in an attached database if P1>1. Write the root page number of the new table into register P2.See documentation on CreateTable for additional information.

CreateTable	Allocate a new table in the main database file if P1==0 or in the auxiliary database file if P1==1 or in an attached database if P1>1. Write the root page number of the new table into register P2The difference between a table and an index is this: A table must have a 4-byte integer key and can have arbitrary data. An index has an arbitrary key but no data.See also: CreateIndex
CursorHint	Provide a hint to cursor P1 that it only needs to return rows that satisfy the Expr in P4. TK_REGISTER terms in the P4 expression refer to values currently held in registers. TK_COLUMN terms in the P4 expression refer to columns in the b-tree to which cursor P1 is pointing.
DecrJumpZero	Register P1 must hold an integer. Decrement the value in register P1 then jump to P2 if the new value is exactly zero.
Delete	Delete the record at which the P1 cursor is currently pointing.If the OPFLAG_SAVEPOSITION bit of the P5 parameter is set, then the cursor will be left pointing at either the next or the previous record in the table. If it is left pointing at the next record, then the next Next instruction will be a no-op. As a result, in this case it is ok to delete a record from within a Next loop. If OPFLAG_SAVEPOSITION bit of P5 is clear, then the cursor will be left in an undefined state.If the OPFLAG_AUXDELETE bit is set on P5, that indicates that this delete one of several associated with deleting a table row and all its associated index entries. Exactly one of those deletes is the "primary" delete. The others are all on OPFLAG_FORDELETE cursors or else are marked with the AUXDELETE flag.If the OPFLAG_NCHANGE flag of P2 (NB: P2 not P5) is set, then the row change count is incremented (otherwise not).P1 must not be pseudo-table. It has to be a real table with multiple rows.If P4 is not NULL, then it is the name of the table that P1 is pointing to. The update hook will be invoked, if it exists. If P4 is not NULL then the P1 cursor must have been positioned using NotFound prior to invoking this opcode.
Destroy	Delete an entire database table or index whose root page in the database file is given by P1.The table being destroyed is in the main database file if P3==0. If P3==1 then the table to be clear is in the auxiliary database file that is used to store tables create using CREATE TEMPORARY TABLE.If AUTOVACUUM is enabled then it is possible that another root page might be moved into the newly deleted root page in order to keep all root pages contiguous at the beginning of the database. The former value of the root page that moved - its value before the move occurred - is stored in register P2. If no page movement was required (because the table being dropped was already the last one in the database) then a zero is stored in register P2. If AUTOVACUUM is disabled then a zero is stored in register P2.See also: Clear
Divide	Divide the value in register P1 by the value in register P2 and store the result in register P3 (P3=P2/P1). If the value in register P1 is zero, then the result is NULL. If either input is NULL, the result is NULL.

DropIndex	Remove the internal (in-memory) data structures that describe the index named P4 in database P1. This is called after an index is dropped from disk (using the Destroy opcode) in order to keep the internal representation of the schema consistent with what is on disk.
DropTable	Remove the internal (in-memory) data structures that describe the table named P4 in database P1. This is called after a table is dropped from disk (using the Destroy opcode) in order to keep the internal representation of the schema consistent with what is on disk.
DropTrigger	Remove the internal (in-memory) data structures that describe the trigger named P4 in database P1. This is called after a trigger is dropped from disk (using the Destroy opcode) in order to keep the internal representation of the schema consistent with what is on disk.
EndCoroutine	The instruction at the address in register P1 is a Yield. Jump to the P2 parameter of that Yield. After the jump, register P1 becomes undefined.See also: InitCoroutine
Eq	This works just like the Lt opcode except that the jump is taken if the operands in registers P1 and P3 are equal. See the Lt opcode for additional information.If SQLITE_NULLEQ is set in P5 then the result of comparison is always either true or false and is never NULL. If both operands are NULL then the result of comparison is true. If either operand is NULL then the result is false. If neither operand is NULL the result is the same as it would be if the SQLITE_NULLEQ flag were omitted from P5.
Expire	Cause precompiled statements to expire. When an expired statement is executed using sqlite3_step() it will either automatically reprepare itself (if it was originally created using sqlite3_prepare_v2()) or it will fail with SQLITE_SCHEMA.If P1 is 0, then all SQL statements become expired. If P1 is non-zero, then only the currently executing statement is expired.
FkCounter	Increment a "constraint counter" by P2 (P2 may be negative or positive). If P1 is non-zero, the database constraint counter is incremented (deferred foreign key constraints). Otherwise, if P1 is zero, the statement counter is incremented (immediate foreign key constraints).
FkIfZero	This opcode tests if a foreign key constraint-counter is currently zero. If so, jump to instruction P2. Otherwise, fall through to the next instruction.If P1 is non-zero, then the jump is taken if the database constraint-counter is zero (the one that counts deferred constraint violations). If P1 is zero, the jump is taken if the statement constraint-counter is zero (immediate foreign key constraint violations).
	If P4==0 then register P3 holds a blob constructed by MakeRecord. If P4>0 then register P3 is the first of P4 registers that form an unpacked record.Cursor P1 is on an index btree. If the record

Found	identified by P3 and P4 is a prefix of any entry in P1 then a jump is made to P2 and P1 is left pointing at the matching entry.This operation leaves the cursor in a state where it can be advanced in the forward direction. The Next instruction will work, but not the Prev instruction.See also: NotFound, NoConflict, NotExists. SeekGe
Function	Invoke a user function (P4 is a pointer to an sqlite3_context object that contains a pointer to the function to be run) with P5 arguments taken from register P2 and successors. The result of the function is stored in register P3. Register P3 must not be one of the function inputs.P1 is a 32-bit bitmask indicating whether or not each argument to the function was determined to be constant at compile time. If the first argument was constant then bit 0 of P1 is set. This is used to determine whether meta data associated with a user function argument using the sqlite3_set_auxdata() API may be safely retained until the next invocation of this opcode.SQL functions are initially coded as Function0 with P4 pointing to a FuncDef object. But on first evaluation, the P4 operand is automatically converted into an sqlite3_context object and the operation changed to this Function opcode. In this way, the initialization of the sqlite3_context object occurs only once, rather than once for each evaluation of the function.See also: Function0, AggStep, AggFinal
Function0	Invoke a user function (P4 is a pointer to a FuncDef object that defines the function) with P5 arguments taken from register P2 and successors. The result of the function is stored in register P3. Register P3 must not be one of the function inputs.P1 is a 32-bit bitmask indicating whether or not each argument to the function was determined to be constant at compile time. If the first argument was constant then bit 0 of P1 is set. This is used to determine whether meta data associated with a user function argument using the sqlite3_set_auxdata() API may be safely retained until the next invocation of this opcode.See also: Function, AggStep, AggFinal
Ge	This works just like the Lt opcode except that the jump is taken if the content of register P3 is greater than or equal to the content of register P1. See the Lt opcode for additional information.
Gosub	Write the current address onto register P1 and then jump to address P2.
Goto	An unconditional jump to address P2. The next instruction executed will be the one at index P2 from the beginning of the program.The P1 parameter is not actually used by this opcode. However, it is sometimes set to 1 instead of 0 as a hint to the command-line shell that this Goto is the bottom of a loop and that the lines from P2 down to the current line should be indented for EXPLAIN output.
Gt	This works just like the Lt opcode except that the jump is taken if the content of register P3 is greater than the content of register P1. See the Lt opcode for additional information.
	Exit immediately. All open cursors, etc are closed automatically.P1 is the result code returned by sqlite3_exec(), sqlite3_reset(), or

Halt	sqlite3_finalize(). For a normal halt, this should be SQLITE_OK (0). For errors, it can be some other value. If P1!=0 then P2 will determine whether or not to rollback the current transaction. Do not rollback if P2==OE_Fail. Do the rollback if P2==OE_Rollback. If P2==OE_Abort, then back out all changes that have occurred during this execution of the VDBE, but do not rollback the transaction.If P4 is not null then it is an error message string.P5 is a value between 0 and 4, inclusive, that modifies the P4 string.0: (no change) 1: NOT NULL contraint failed: P4 2: UNIQUE constraint failed: P4 3: CHECK constraint failed: P4 4: FOREIGN KEY constraint failed: P4If P5 is not zero and P4 is NULL, then everything after the ":" is omitted.There is an implied "Halt 0 0 0" instruction inserted at the very end of every program. So a jump past the last instruction of the program is the same as executing Halt.
HaltIfNull	Check the value in register P3. If it is NULL then Halt using parameter P1, P2, and P4 as if this were a Halt instruction. If the value in register P3 is not NULL, then this routine is a no-op. The P5 parameter should be 1.
IdxDelete	The content of P3 registers starting at register P2 form an unpacked index key. This opcode removes that entry from the index opened by cursor P1.
IdxGE	The P4 register values beginning with P3 form an unpacked index key that omits the PRIMARY KEY. Compare this key value against the index that P1 is currently pointing to, ignoring the PRIMARY KEY or ROWID fields at the end.If the P1 index entry is greater than or equal to the key value then jump to P2. Otherwise fall through to the next instruction.
IdxGT	The P4 register values beginning with P3 form an unpacked index key that omits the PRIMARY KEY. Compare this key value against the index that P1 is currently pointing to, ignoring the PRIMARY KEY or ROWID fields at the end.If the P1 index entry is greater than the key value then jump to P2. Otherwise fall through to the next instruction.
IdxInsert	Register P2 holds an SQL index key made using the MakeRecord instructions. This opcode writes that key into the index P1. Data for the entry is nil.P3 is a flag that provides a hint to the b-tree layer that this insert is likely to be an append.If P5 has the OPFLAG_NCHANGE bit set, then the change counter is incremented by this instruction. If the OPFLAG_NCHANGE bit is clear, then the change counter is unchanged.If P5 has the OPFLAG_USESEEKRESULT bit set, then the cursor must have just done a seek to the spot where the new entry is to be inserted. This flag avoids doing an extra seek.This instruction only works for indices. The equivalent instruction for tables is Insert.
IdxLE	The P4 register values beginning with P3 form an unpacked index key that omits the PRIMARY KEY or ROWID. Compare this key value against the index that P1 is currently pointing to, ignoring the PRIMARY KEY or ROWID on the P1 index.If the P1 index entry is

	less than or equal to the key value then jump to P2. Otherwise fall through to the next instruction.
IdxLT	The P4 register values beginning with P3 form an unpacked index key that omits the PRIMARY KEY or ROWID. Compare this key value against the index that P1 is currently pointing to, ignoring the PRIMARY KEY or ROWID on the P1 index.If the P1 index entry is less than the key value then jump to P2. Otherwise fall through to the next instruction.
IdxRowid	Write into register P2 an integer which is the last entry in the record at the end of the index key pointed to by cursor P1. This integer should be the rowid of the table entry to which this index entry points.See also: Rowid, MakeRecord.
If	Jump to P2 if the value in register P1 is true. The value is considered true if it is numeric and non-zero. If the value in P1 is NULL then take the jump if and only if P3 is non-zero.
IfNot	Jump to P2 if the value in register P1 is False. The value is considered false if it has a numeric value of zero. If the value in P1 is NULL then take the jump if and only if P3 is non-zero.
IfNotZero	Register P1 must contain an integer. If the content of register P1 is initially nonzero, then subtract P3 from the value in register P1 and jump to P2. If register P1 is initially zero, leave it unchanged and fall through.
IfPos	Register P1 must contain an integer. If the value of register P1 is 1 or greater, subtract P3 from the value in P1 and jump to P2.If the initial value of register P1 is less than 1, then the value is unchanged and control passes through to the next instruction.
IncrVacuum	Perform a single step of the incremental vacuum procedure on the P1 database. If the vacuum has finished, jump to instruction P2. Otherwise, fall through to the next instruction.
Init	Programs contain a single instance of this opcode as the very first opcode.If tracing is enabled (by the sqlite3_trace()) interface, then the UTF-8 string contained in P4 is emitted on the trace callback. Or if P4 is blank, use the string returned by sqlite3_sql().If P2 is not zero, jump to instruction P2.
InitCoroutine	Set up register P1 so that it will Yield to the coroutine located at address P3.If P2!=0 then the coroutine implementation immediately follows this opcode. So jump over the coroutine implementation to address P2.See also: EndCoroutine
	Write an entry into the table of cursor P1. A new entry is created if it doesn't already exist or the data for an existing entry is overwritten. The data is the value MEM_Blob stored in register number P2. The key is stored in register P3. The key must be a MEM_Int.If the OPFLAG_NCHANGE flag of P5 is set, then the row change count is incremented (otherwise not). If the OPFLAG_LASTROWID flag of P5 is set, then rowid is stored for subsequent return by the sqlite3_last_insert_rowid() function (otherwise it is unmodified).If the

Insert	OPFLAG_USESEEKRESULT flag of P5 is set and if the result of the last seek operation (OP_NotExists) was a success, then this operation will not attempt to find the appropriate row before doing the insert but will instead overwrite the row that the cursor is currently pointing to. Presumably, the prior NotExists opcode has already positioned the cursor correctly. This is an optimization that boosts performance by avoiding redundant seeks.If the OPFLAG_ISUPDATE flag is set, then this opcode is part of an UPDATE operation. Otherwise (if the flag is clear) then this opcode is part of an INSERT operation. The difference is only important to the update hook.Parameter P4 may point to a string containing the table-name, or may be NULL. If it is not NULL, then the update-hook (sqlite3.xUpdateCallback) is invoked following a successful insert.(WARNING/TODO: If P1 is a pseudo-cursor and P2 is dynamically allocated, then ownership of P2 is transferred to the pseudo-cursor and register P2 becomes ephemeral. If the cursor is changed, the value of register P2 will then change. Make sure this does not cause any problems.)This instruction only works on tables. The equivalent instruction for indices is IdxInsert.
InsertInt	This works exactly like Insert except that the key is the integer value P3, not the value of the integer stored in register P3.
Int64	P4 is a pointer to a 64-bit integer value. Write that value into register P2.
IntCopy	Transfer the integer value held in register P1 into register P2.This is an optimized version of SCopy that works only for integer values.
Integer	The 32-bit integer value P1 is written into register P2.
IntegrityCk	Do an analysis of the currently open database. Store in register P1 the text of an error message describing any problems. If no problems are found, store a NULL in register P1.The register P3 contains the maximum number of allowed errors. At most reg(P3) errors will be reported. In other words, the analysis stops as soon as reg(P1) errors are seen. Reg(P1) is updated with the number of errors remaining.The root page numbers of all tables in the database are integers stored in P4_INTARRAY argument.If P5 is not zero, the check is done on the auxiliary database file, not the main database file.This opcode is used to implement the integrity_check pragma.
IsNull	Jump to P2 if the value in register P1 is NULL.
JournalMode	Change the journal mode of database P1 to P3. P3 must be one of the PAGER_JOURNALMODE_XXX values. If changing between the various rollback modes (delete, truncate, persist, off and memory), this is a simple operation. No IO is required.If changing into or out of WAL mode the procedure is more complicated.Write a string containing the final journal-mode to register P2.
Jump	Jump to the instruction at address P1, P2, or P3 depending on whether in the most recent Compare instruction the P1 vector was less than equal to, or greater than the P2 vector, respectively.

JumpZeroIncr	The register P1 must contain an integer. If register P1 is initially zero, then jump to P2. Increment register P1 regardless of whether or not the jump is taken.
Last	The next use of the Rowid or Column or Prev instruction for P1 will refer to the last entry in the database table or index. If the table or index is empty and P2>0, then jump immediately to P2. If P2 is 0 or if the table or index is not empty, fall through to the following instruction.This opcode leaves the cursor configured to move in reverse order, from the end toward the beginning. In other words, the cursor is configured to use Prev, not Next.
Le	This works just like the Lt opcode except that the jump is taken if the content of register P3 is less than or equal to the content of register P1. See the Lt opcode for additional information.
LoadAnalysis	Read the sqlite_stat1 table for database P1 and load the content of that table into the internal index hash table. This will cause the analysis to be used when preparing all subsequent queries.
Lt	Compare the values in register P1 and P3. If reg(P3)<reg(P1) then jump to address P2.If the SQLITE_JUMPIFNULL bit of P5 is set and either reg(P1) or reg(P3) is NULL then take the jump. If the SQLITE_JUMPIFNULL bit is clear then fall through if either operand is NULL.The SQLITE_AFF_MASK portion of P5 must be an affinity character - SQLITE_AFF_TEXT, SQLITE_AFF_INTEGER, and so forth. An attempt is made to coerce both inputs according to this affinity before the comparison is made. If the SQLITE_AFF_MASK is 0x00, then numeric affinity is used. Note that the affinity conversions are stored back into the input registers P1 and P3. So this opcode can cause persistent changes to registers P1 and P3.Once any conversions have taken place, and neither value is NULL, the values are compared. If both values are blobs then memcmp() is used to determine the results of the comparison. If both values are text, then the appropriate collating function specified in P4 is used to do the comparison. If P4 is not specified then memcmp() is used to compare text string. If both values are numeric, then a numeric comparison is used. If the two values are of different types, then numbers are considered less than strings and strings are considered less than blobs.If the SQLITE_STOREP2 bit of P5 is set, then do not jump. Instead, store a boolean result (either 0, or 1, or NULL) in register P2.If the SQLITE_NULLEQ bit is set in P5, then NULL values are considered equal to one another, provided that they do not have their MEM_Cleared bit set.
MakeRecord	Convert P2 registers beginning with P1 into the record format use as a data record in a database table or as a key in an index. The Column opcode can decode the record later.P4 may be a string that is P2 characters long. The nth character of the string indicates the column affinity that should be used for the nth field of the index key.The mapping from character to affinity is given by the SQLITE*AFF* macros defined in sqliteInt.h.If P4 is NULL then all index fields have the affinity BLOB.

MaxPgcnt	Try to set the maximum page count for database P1 to the value in P3. Do not let the maximum page count fall below the current page count and do not change the maximum page count value if P3==0.Store the maximum page count after the change in register P2.
MemMax	P1 is a register in the root frame of this VM (the root frame is different from the current frame if this instruction is being executed within a sub-program). Set the value of register P1 to the maximum of its current value and the value in register P2.This instruction throws an error if the memory cell is not initially an integer.
Move	Move the P3 values in register P1..P1+P3-1 over into registers P2..P2+P3-1. Registers P1..P1+P3-1 are left holding a NULL. It is an error for register ranges P1..P1+P3-1 and P2..P2+P3-1 to overlap. It is an error for P3 to be less than 1.
Multiply	Multiply the value in register P1 by the value in register P2 and store the result in register P3. If either input is NULL, the result is NULL.
MustBeInt	Force the value in register P1 to be an integer. If the value in P1 is not an integer and cannot be converted into an integer without data loss, then jump immediately to P2, or if P2==0 raise an SQLITE_MISMATCH exception.
Ne	This works just like the Lt opcode except that the jump is taken if the operands in registers P1 and P3 are not equal. See the Lt opcode for additional information.If SQLITE_NULLEQ is set in P5 then the result of comparison is always either true or false and is never NULL. If both operands are NULL then the result of comparison is false. If either operand is NULL then the result is true. If neither operand is NULL the result is the same as it would be if the SQLITE_NULLEQ flag were omitted from P5.
NewRowid	Get a new integer record number (a.k.a "rowid") used as the key to a table. The record number is not previously used as a key in the database table that cursor P1 points to. The new record number is written written to register P2.If P3>0 then P3 is a register in the root frame of this VDBE that holds the largest previously generated record number. No new record numbers are allowed to be less than this value. When this value reaches its maximum, an SQLITE_FULL error is generated. The P3 register is updated with the ' generated record number. This P3 mechanism is used to help implement the AUTOINCREMENT feature.
Next	Advance cursor P1 so that it points to the next key/data pair in its table or index. If there are no more key/value pairs then fall through to the following instruction. But if the cursor advance was successful, jump immediately to P2.The Next opcode is only valid following an SeekGT, SeekGE, or Rewind opcode used to position the cursor. Next is not allowed to follow SeekLT, SeekLE, or Last.The P1 cursor must be for a real table, not a pseudo-table. P1 must have been opened prior to this opcode or the program will segfault.The P3 value is a hint to the btree implementation. If P3==1, that means P1 is an SQL index and that this instruction

	could have been omitted if that index had been unique. P3 is usually 0. P3 is always either 0 or 1.P4 is always of type P4_ADVANCE. The function pointer points to sqlite3BtreeNext().If P5 is positive and the jump is taken, then event counter number P5-1 in the prepared statement is incremented.See also: Prev, NextIfOpen
NextIfOpen	This opcode works just like Next except that if cursor P1 is not open it behaves a no-op.
NoConflict	If P4==0 then register P3 holds a blob constructed by MakeRecord. If P4>0 then register P3 is the first of P4 registers that form an unpacked record.Cursor P1 is on an index btree. If the record identified by P3 and P4 contains any NULL value, jump immediately to P2. If all terms of the record are not-NULL then a check is done to determine if any row in the P1 index btree has a matching key prefix. If there are no matches, jump immediately to P2. If there is a match, fall through and leave the P1 cursor pointing to the matching row.This opcode is similar to NotFound with the exceptions that the branch is always taken if any part of the search key input is NULL.This operation leaves the cursor in a state where it cannot be advanced in either direction. In other words, the Next and Prev opcodes do not work after this operation.See also: NotFound, Found, NotExists
Noop	Do nothing. This instruction is often useful as a jump destination.
Not	Interpret the value in register P1 as a boolean value. Store the boolean complement in register P2. If the value in register P1 is NULL, then a NULL is stored in P2.
NotExists	P1 is the index of a cursor open on an SQL table btree (with integer keys). P3 is an integer rowid. If P1 does not contain a record with rowid P3 then jump immediately to P2. Or, if P2 is 0, raise an SQLITE_CORRUPT error. If P1 does contain a record with rowid P3 then leave the cursor pointing at that record and fall through to the next instruction.The NotFound opcode performs the same operation on index btrees (with arbitrary multi-value keys).This opcode leaves the cursor in a state where it cannot be advanced in either direction. In other words, the Next and Prev opcodes will not work following this opcode.See also: Found, NotFound, NoConflict
NotFound	If P4==0 then register P3 holds a blob constructed by MakeRecord. If P4>0 then register P3 is the first of P4 registers that form an unpacked record.Cursor P1 is on an index btree. If the record identified by P3 and P4 is not the prefix of any entry in P1 then a jump is made to P2. If P1 does contain an entry whose prefix matches the P3/P4 record then control falls through to the next instruction and P1 is left pointing at the matching entry.This operation leaves the cursor in a state where it cannot be advanced in either direction. In other words, the Next and Prev opcodes do not work after this operation.See also: Found, NotExists, NoConflict
NotNull	Jump to P2 if the value in register P1 is not NULL.
	Write a NULL into registers P2. If P3 greater than P2, then also

Null	write NULL into register P3 and every register in between P2 and P3. If P3 is less than P2 (typically P3 is zero) then only register P2 is set to NULL.If the P1 value is non-zero, then also set the MEM_Cleared flag so that NULL values will not compare equal even if SQLITE_NULLEQ is set on Ne or Eq.
NullRow	Move the cursor P1 to a null row. Any Column operations that occur while the cursor is on the null row will always write a NULL.
OffsetLimit	This opcode performs a commonly used computation associated with LIMIT and OFFSET process. r[P1] holds the limit counter. r[P3] holds the offset counter. The opcode computes the combined value of the LIMIT and OFFSET and stores that value in r[P2]. The r[P2] value computed is the total number of rows that will need to be visited in order to complete the query.If r[P3] is zero or negative, that means there is no OFFSET and r[P2] is set to be the value of the LIMIT, r[P1].if r[P1] is zero or negative, that means there is no LIMIT and r[P2] is set to -1.Otherwise, r[P2] is set to the sum of r[P1] and r[P3].
Once	Check the "once" flag number P1. If it is set, jump to instruction P2. Otherwise, set the flag and fall through to the next instruction. In other words, this opcode causes all following opcodes up through P2 (but not including P2) to run just once and to be skipped on subsequent times through the loop.All "once" flags are initially cleared whenever a prepared statement first begins to run.
OpenAutoindex	This opcode works the same as OpenEphemeral. It has a different name to distinguish its use. Tables created using by this opcode will be used for automatically created transient indices in joins.
OpenEphemeral	Open a new cursor P1 to a transient table. The cursor is always opened read/write even if the main database is read-only. The ephemeral table is deleted automatically when the cursor is closed.P2 is the number of columns in the ephemeral table. The cursor points to a BTree table if P4==0 and to a BTree index if P4 is not 0. If P4 is not NULL, it points to a KeyInfo structure that defines the format of keys in the index.The P5 parameter can be a mask of the BTREE_* flags defined in btree.h. These flags control aspects of the operation of the btree. The BTREE_OMIT_JOURNAL and BTREE_SINGLE flags are added automatically.
OpenPseudo	Open a new cursor that points to a fake table that contains a single row of data. The content of that one row is the content of memory register P2. In other words, cursor P1 becomes an alias for the MEM_Blob content contained in register P2.A pseudo-table created by this opcode is used to hold a single row output from the sorter so that the row can be decomposed into individual columns using the Column opcode. The Column opcode is the only cursor opcode that works with a pseudo-table.P3 is the number of fields in the records that will be stored by the pseudo-table.
	Open a read-only cursor for the database table whose root page is P2 in a database file. The database file is determined by P3. P3==0 means the main database, P3==1 means the database used for

OpenRead	temporary tables, and P3>1 means used the corresponding attached database. Give the new cursor an identifier of P1. The P1 values need not be contiguous but all P1 values should be small integers. It is an error for P1 to be negative.If P5!=0 then use the content of register P2 as the root page, not the value of P2 itself.There will be a read lock on the database whenever there is an open cursor. If the database was unlocked prior to this instruction then a read lock is acquired as part of this instruction. A read lock allows other processes to read the database but prohibits any other process from modifying the database. The read lock is released when all cursors are closed. If this instruction attempts to get a read lock but fails, the script terminates with an SQLITE_BUSY error code.The P4 value may be either an integer (P4_INT32) or a pointer to a KeyInfo structure (P4_KEYINFO). If it is a pointer to a KeyInfo structure, then said structure defines the content and collating sequence of the index being opened. Otherwise, if P4 is an integer value, it is set to the number of columns in the table.See also: OpenWrite, ReopenIdx
OpenWrite	Open a read/write cursor named P1 on the table or index whose root page is P2. Or if P5!=0 use the content of register P2 to find the root page.The P4 value may be either an integer (P4_INT32) or a pointer to a KeyInfo structure (P4_KEYINFO). If it is a pointer to a KeyInfo structure, then said structure defines the content and collating sequence of the index being opened. Otherwise, if P4 is an integer value, it is set to the number of columns in the table, or to the largest index of any column of the table that is actually used.This instruction works just like OpenRead except that it opens the cursor in read/write mode. For a given table, there can be one or more read-only cursors or a single read/write cursor but not both.See also OpenRead.
Or	Take the logical OR of the values in register P1 and P2 and store the answer in register P3.If either P1 or P2 is nonzero (true) then the result is 1 (true) even if the other input is NULL. A NULL and false or two NULLs give a NULL output.
Pagecount	Write the current number of pages in database P1 to memory cell P2.
Param	This opcode is only ever present in sub-programs called via the Program instruction. Copy a value currently stored in a memory cell of the calling (parent) frame to cell P2 in the current frames address space. This is used by trigger programs to access the new. *and old.* values.The address of the cell in the parent frame is determined by adding the value of the P1 argument to the value of the P1 argument to the calling Program instruction.
ParseSchema	Read and parse all entries from the SQLITE_MASTER table of database P1 that match the WHERE clause P4.This opcode invokes the parser to create a new virtual machine, then runs the new virtual machine. It is thus a re-entrant opcode.
	Set the permutation used by the Compare operator to be the array of integers in P4.The permutation is only valid until the next

Permutation	Compare that has the OPFLAG_PERMUTE bit set in P5. Typically the Permutation should occur immediately prior to the Compare.The first integer in the P4 integer array is the length of the array and does not become part of the permutation.
Prev	Back up cursor P1 so that it points to the previous key/data pair in its table or index. If there is no previous key/value pairs then fall through to the following instruction. But if the cursor backup was successful, jump immediately to P2.The Prev opcode is only valid following an SeekLT, SeekLE, or Last opcode used to position the cursor. Prev is not allowed to follow SeekGT, SeekGE, or Rewind.The P1 cursor must be for a real table, not a pseudo-table. If P1 is not open then the behavior is undefined.The P3 value is a hint to the btree implementation. If P3==1, that means P1 is an SQL index and that this instruction could have been omitted if that index had been unique. P3 is usually 0. P3 is always either 0 or 1.P4 is always of type P4_ADVANCE. The function pointer points to sqlite3BtreePrevious().If P5 is positive and the jump is taken, then event counter number P5-1 in the prepared statement is incremented.
PrevIfOpen	This opcode works just like Prev except that if cursor P1 is not open it behaves a no-op.
Program	Execute the trigger program passed as P4 (type P4_SUBPROGRAM).P1 contains the address of the memory cell that contains the first memory cell in an array of values used as arguments to the sub-program. P2 contains the address to jump to if the sub-program throws an IGNORE exception using the RAISE() function. Register P3 contains the address of a memory cell in this (the parent) VM that is used to allocate the memory required by the sub-vdbe at runtime.P4 is a pointer to the VM containing the trigger program.If P5 is non-zero, then recursive program invocation is enabled.
ReadCookie	Read cookie number P3 from database P1 and write it into register P2. P3==1 is the schema version. P3==2 is the database format. P3==3 is the recommended pager cache size, and so forth. P1==0 is the main database file and P1==1 is the database file used to store temporary tables.There must be a read-lock on the database (either a transaction must be started or there must be an open cursor) before executing this instruction.
Real	P4 is a pointer to a 64-bit floating point value. Write that value into register P2.
RealAffinity	If register P1 holds an integer convert it to a real value.This opcode is used when extracting information from a column that has REAL affinity. Such column values may still be stored as integers, for space efficiency, but after extraction we want them to have only a real value.
Remainder	Compute the remainder after integer register P2 is divided by register P1 and store the result in register P3. If the value in register P1 is zero the result is NULL. If either operand is NULL, the result is

	NULL.
ReopenIdx	The ReopenIdx opcode works exactly like ReadOpen except that it first checks to see if the cursor on P1 is already open with a root page number of P2 and if it is this opcode becomes a no-op. In other words, if the cursor is already open, do not reopen it.The ReopenIdx opcode may only be used with P5==0 and with P4 being a P4_KEYINFO object. Furthermore, the P3 value must be the same as every other ReopenIdx or OpenRead for the same cursor number.See the OpenRead opcode documentation for additional information.
ResetCount	The value of the change counter is copied to the database handle change counter (returned by subsequent calls to sqlite3_changes()). Then the VMs internal change counter resets to 0. This is used by trigger programs.
ResetSorter	Delete all contents from the ephemeral table or sorter that is open on cursor P1.This opcode only works for cursors used for sorting and opened with OpenEphemeral or SorterOpen.
ResultRow	The registers P1 through P1+P2-1 contain a single row of results. This opcode causes the sqlite3_step() call to terminate with an SQLITE_ROW return code and it sets up the sqlite3_stmt structure to provide access to the r(P1)..r(P1+P2-1) values as the result row.
Return	Jump to the next instruction after the address in register P1. After the jump, register P1 becomes undefined.
Rewind	The next use of the Rowid or Column or Next instruction for P1 will refer to the first entry in the database table or index. If the table or index is empty, jump immediately to P2. If the table or index is not empty, fall through to the following instruction.This opcode leaves the cursor configured to move in forward order, from the beginning toward the end. In other words, the cursor is configured to use Next, not Prev.
RowData	Write into register P2 the complete row data for cursor P1. There is no interpretation of the data. It is just copied onto the P2 register exactly as it is found in the database file.If the P1 cursor must be pointing to a valid row (not a NULL row) of a real table, not a pseudo-table.
Rowid	Store in register P2 an integer which is the key of the table entry that P1 is currently point to.P1 can be either an ordinary table or a virtual table. There used to be a separate OP_VRowid opcode for use with virtual tables, but this one opcode now works for both table types.
RowKey	Write into register P2 the complete row key for cursor P1. There is no interpretation of the data. The key is copied onto the P2 register exactly as it is found in the database file.If the P1 cursor must be pointing to a valid row (not a NULL row) of a real table, not a pseudo-table.
	Insert the integer value held by register P2 into a boolean index held

	in register P1.An assertion fails if P2 is not an integer.
RowSetRead	Extract the smallest value from boolean index P1 and put that value into register P3. Or, if boolean index P1 is initially empty, leave P3 unchanged and jump to instruction P2.
RowSetTest	Register P3 is assumed to hold a 64-bit integer value. If register P1 contains a RowSet object and that RowSet object contains the value held in P3, jump to register P2. Otherwise, insert the integer in P3 into the RowSet and continue on to the next opcode.The RowSet object is optimized for the case where successive sets of integers, where each set contains no duplicates. Each set of values is identified by a unique P4 value. The first set must have P4==0, the final set P4=-1. P4 must be either -1 or non-negative. For non-negative values of P4 only the lower 4 bits are significant.This allows optimizations: (a) when P4==0 there is no need to test the rowset object for P3, as it is guaranteed not to contain it, (b) when P4==-1 there is no need to insert the value, as it will never be tested for, and (c) when a value that is part of set X is inserted, there is no need to search to see if the same value was previously inserted as part of set X (only if it was previously inserted as part of some other set).
Savepoint	Open, release or rollback the savepoint named by parameter P4, depending on the value of P1. To open a new savepoint, P1==0. To release (commit) an existing savepoint, P1==1, or to rollback an existing savepoint P1==2.
SCopy	Make a shallow copy of register P1 into register P2.This instruction makes a shallow copy of the value. If the value is a string or blob, then the copy is only a pointer to the original and hence if the original changes so will the copy. Worse, if the original is deallocated, the copy becomes invalid. Thus the program must guarantee that the original will not change during the lifetime of the copy. Use Copy to make a complete copy.
Seek	P1 is an open index cursor and P3 is a cursor on the corresponding table. This opcode does a deferred seek of the P3 table cursor to the row that corresponds to the current row of P1.This is a deferred seek. Nothing actually happens until the cursor is used to read a record. That way, if no reads occur, no unnecessary I/O happens.P4 may be an array of integers (type P4_INTARRAY) containing one entry for each column in the P3 table. If array entry a(i) is non-zero, then reading column a(i)-1 from cursor P3 is equivalent to performing the deferred seek and then reading column i from P1. This information is stored in P3 and used to redirect reads against P3 over to P1, thus possibly avoiding the need to seek and read cursor P3.
	If cursor P1 refers to an SQL table (B-Tree that uses integer keys), use the value in register P3 as the key. If cursor P1 refers to an SQL index, then P3 is the first in an array of P4 registers that are used as an unpacked index key.Reposition cursor P1 so that it points to the smallest entry that is greater than or equal to the key value. If there are no records greater than or equal to the key and P2 is not zero,

SeekGE	are no records greater than or equal to the key and P2 is not zero, then jump to P2.If the cursor P1 was opened using the OPFLAG_SEEKEQ flag, then this opcode will always land on a record that equally equals the key, or else jump immediately to P2. When the cursor is OPFLAG_SEEKEQ, this opcode must be followed by an IdxLE opcode with the same arguments. The IdxLE opcode will be skipped if this opcode succeeds, but the IdxLE opcode will be used on subsequent loop iterations.This opcode leaves the cursor configured to move in forward order, from the beginning toward the end. In other words, the cursor is configured to use Next, not Prev.See also: Found, NotFound, SeekLt, SeekGt, SeekLe
SeekGT	If cursor P1 refers to an SQL table (B-Tree that uses integer keys), use the value in register P3 as a key. If cursor P1 refers to an SQL index, then P3 is the first in an array of P4 registers that are used as an unpacked index key.Reposition cursor P1 so that it points to the smallest entry that is greater than the key value. If there are no records greater than the key and P2 is not zero, then jump to P2.This opcode leaves the cursor configured to move in forward order, from the beginning toward the end. In other words, the cursor is configured to use Next, not Prev.See also: Found, NotFound, SeekLt, SeekGe, SeekLe
SeekLE	If cursor P1 refers to an SQL table (B-Tree that uses integer keys), use the value in register P3 as a key. If cursor P1 refers to an SQL index, then P3 is the first in an array of P4 registers that are used as an unpacked index key.Reposition cursor P1 so that it points to the largest entry that is less than or equal to the key value. If there are no records less than or equal to the key and P2 is not zero, then jump to P2.This opcode leaves the cursor configured to move in reverse order, from the end toward the beginning. In other words, the cursor is configured to use Prev, not Next.If the cursor P1 was opened using the OPFLAG_SEEKEQ flag, then this opcode will always land on a record that equally equals the key, or else jump immediately to P2. When the cursor is OPFLAG_SEEKEQ, this opcode must be followed by an IdxGE opcode with the same arguments. The IdxGE opcode will be skipped if this opcode succeeds, but the IdxGE opcode will be used on subsequent loop iterations.See also: Found, NotFound, SeekGt, SeekGe, SeekLt
SeekLT	If cursor P1 refers to an SQL table (B-Tree that uses integer keys), use the value in register P3 as a key. If cursor P1 refers to an SQL index, then P3 is the first in an array of P4 registers that are used as an unpacked index key.Reposition cursor P1 so that it points to the largest entry that is less than the key value. If there are no records less than the key and P2 is not zero, then jump to P2.This opcode leaves the cursor configured to move in reverse order, from the end toward the beginning. In other words, the cursor is configured to use Prev, not Next.See also: Found, NotFound, SeekGt, SeekGe, SeekLe
Sequence	Find the next available sequence number for cursor P1. Write the sequence number into register P2. The sequence number on the

SequenceTest	P1 is a sorter cursor. If the sequence counter is currently zero, jump to P2. Regardless of whether or not the jump is taken, increment the the sequence value.
SetCookie	Write the integer value P3 into cookie number P2 of database P1. P2==1 is the schema version. P2==2 is the database format. P2==3 is the recommended pager cache size, and so forth. P1==0 is the main database file and P1==1 is the database file used to store temporary tables.A transaction must be started before executing this opcode.
ShiftLeft	Shift the integer value in register P2 to the left by the number of bits specified by the integer in register P1. Store the result in register P3. If either input is NULL, the result is NULL.
ShiftRight	Shift the integer value in register P2 to the right by the number of bits specified by the integer in register P1. Store the result in register P3. If either input is NULL, the result is NULL.
SoftNull	Set register P1 to have the value NULL as seen by the MakeRecord instruction, but do not free any string or blob memory associated with the register, so that if the value was a string or blob that was previously copied using SCopy, the copies will continue to be valid.
Sort	This opcode does exactly the same thing as Rewind except that it increments an undocumented global variable used for testing.Sorting is accomplished by writing records into a sorting index, then rewinding that index and playing it back from beginning to end. We use the Sort opcode instead of Rewind to do the rewinding so that the global variable will be incremented and regression tests can determine whether or not the optimizer is correctly optimizing out sorts.
SorterCompare	P1 is a sorter cursor. This instruction compares a prefix of the record blob in register P3 against a prefix of the entry that the sorter cursor currently points to. Only the first P4 fields of r[P3] and the sorter record are compared.If either P3 or the sorter contains a NULL in one of their significant fields (not counting the P4 fields at the end which are ignored) then the comparison is assumed to be equal.Fall through to next instruction if the two records compare equal to each other. Jump to P2 if they are different.
SorterData	Write into register P2 the current sorter data for sorter cursor P1. Then clear the column header cache on cursor P3.This opcode is normally use to move a record out of the sorter and into a register that is the source for a pseudo-table cursor created using OpenPseudo. That pseudo-table cursor is the one that is identified by parameter P3. Clearing the P3 column cache as part of this opcode saves us from having to issue a separate NullRow instruction to clear that cache.
SorterOpen	This opcode works like OpenEphemeral except that it opens a transient index that is specifically designed to sort large tables using an external merge-sort algorithm.If argument P3 is non-zero, then it indicates that the sorter may assume that a stable sort considering

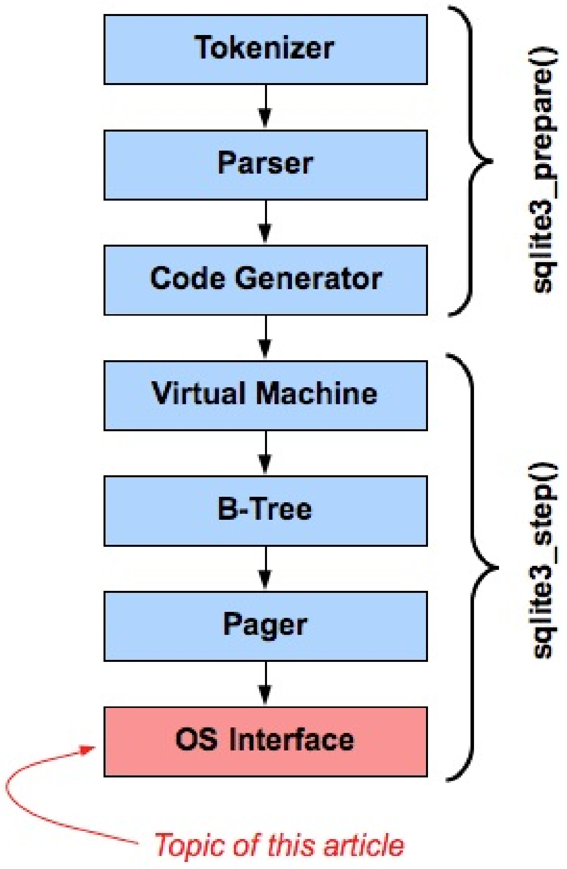
SorterOpen	indicates that the sorter may assume that a stable sort considering the first P3 fields of each key is sufficient to produce the required results.
String	The string value P4 of length P1 (bytes) is stored in register P2.If P5!=0 and the content of register P3 is greater than zero, then the datatype of the register P2 is converted to BLOB. The content is the same sequence of bytes, it is merely interpreted as a BLOB instead of a string, as if it had been CAST.
String8	P4 points to a nul terminated UTF-8 string. This opcode is transformed into a String opcode before it is executed for the first time. During this transformation, the length of string P4 is computed and stored as the P1 parameter.
Subtract	Subtract the value in register P1 from the value in register P2 and store the result in register P3. If either input is NULL, the result is NULL.
TableLock	Obtain a lock on a particular table. This instruction is only used when the shared-cache feature is enabled.P1 is the index of the database in sqlite3.aDb[] of the database on which the lock is acquired. A readlock is obtained if P3==0 or a write lock if P3==1.P2 contains the root-page of the table to lock.P4 contains a pointer to the name of the table being locked. This is only used to generate an error message if the lock cannot be obtained.
Transaction	Begin a transaction on database P1 if a transaction is not already active. If P2 is non-zero, then a write-transaction is started, or if a read-transaction is already active, it is upgraded to a write-transaction. If P2 is zero, then a read-transaction is started.P1 is the index of the database file on which the transaction is started. Index 0 is the main database file and index 1 is the file used for temporary tables. Indices of 2 or more are used for attached databases.If a write-transaction is started and the Vdbe.usesStmtJournal flag is true (this flag is set if the Vdbe may modify more than one row and may throw an ABORT exception), a statement transaction may also be opened. More specifically, a statement transaction is opened iff the database connection is currently not in autocommit mode, or if there are other active statements. A statement transaction allows the changes made by this VDBE to be rolled back after an error without having to roll back the entire transaction. If no error is encountered, the statement transaction will automatically commit when the VDBE halts.If P5!=0 then this opcode also checks the schema cookie against P3 and the schema generation counter against P4. The cookie changes its value whenever the database schema changes. This operation is used to detect when that the cookie has changed and that the current process needs to reread the schema. If the schema cookie in P3 differs from the schema cookie in the database header or if the schema generation counter in P4 differs from the current generation counter, then an SQLITE_SCHEMA error is raised and execution halts. The sqlite3_step() wrapper function might then reprepare the statement and rerun it from the beginning.

Vacuum	machines to be created and run. It may not be called from within a transaction.
Variable	Transfer the values of bound parameter P1 into register P2If the parameter is named, then its name appears in P4. The P4 value is used by sqlite3_bind_parameter_name().
VBegin	P4 may be a pointer to an sqlite3_vtab structure. If so, call the xBegin method for that table.Also, whether or not P4 is set, check that this is not being called from within a callback to a virtual table xSync() method. If it is, the error code will be set to SQLITE_LOCKED.
VColumn	Store the value of the P2-th column of the row of the virtual-table that the P1 cursor is pointing to into register P3.
VCreate	P2 is a register that holds the name of a virtual table in database P1. Call the xCreate method for that table.
VDestroy	P4 is the name of a virtual table in database P1. Call the xDestroy method of that table.
VFilter	P1 is a cursor opened using VOpen. P2 is an address to jump to if the filtered result set is empty.P4 is either NULL or a string that was generated by the xBestIndex method of the module. The interpretation of the P4 string is left to the module implementation.This opcode invokes the xFilter method on the virtual table specified by P1. The integer query plan parameter to xFilter is stored in register P3. Register P3+1 stores the argc parameter to be passed to the xFilter method. Registers P3+2..P3+1+argc are the argc additional parameters which are passed to xFilter as argv. Register P3+2 becomes argv[0] when passed to xFilter.A jump is made to P2 if the result set after filtering would be empty.
VNext	Advance virtual table P1 to the next row in its result set and jump to instruction P2. Or, if the virtual table has reached the end of its result set, then fall through to the next instruction.
VOpen	P4 is a pointer to a virtual table object, an sqlite3_vtab structure. P1 is a cursor number. This opcode opens a cursor to the virtual table and stores that cursor in P1.
VRename	P4 is a pointer to a virtual table object, an sqlite3_vtab structure. This opcode invokes the corresponding xRename method. The value in register P1 is passed as the zName argument to the xRename method.
	P4 is a pointer to a virtual table object, an sqlite3_vtab structure. This opcode invokes the corresponding xUpdate method. P2 values are contiguous memory cells starting at P3 to pass to the xUpdate invocation. The value in register (P3+P2-1) corresponds to the p2th element of the argv array passed to xUpdate.The xUpdate method will do a DELETE or an INSERT or both. The argv[0] element (which corresponds to memory cell P3) is the rowid of a row to delete. If argv[0] is NULL then no deletion occurs. The argv[1]

VUpdate	delete. If argv[0] is NULL then no deletion occurs. The argv[1] element is the rowid of the new row. This can be NULL to have the virtual table select the new rowid for itself. The subsequent elements in the array are the values of columns in the new row.If P2==1 then no insert is performed. argv[0] is the rowid of a row to delete.P1 is a boolean flag. If it is set to true and the xUpdate call is successful, then the value returned by sqlite3_last_insert_rowid() is set to the value of the rowid for the row just inserted.P5 is the error actions (OE_Replace, OE_Fail, OE_Ignore, etc) to apply in the case of a constraint failure on an insert or update.
Yield	Swap the program counter with the value in register P1. This has the effect of yielding to a coroutine.If the coroutine that is launched by this instruction ends with Yield or Return then continue to the next instruction. But if the coroutine launched by this instruction ends with EndCoroutine, then jump to P2 rather than continuing with the next instruction.See also: InitCoroutine

The SQLite OS Interface or "VFS"

This article describes the SQLite OS portability layer or "VFS" - the module at the bottom of the SQLite implementation stack that provides portability across operating systems.



1.0 The VFS In Relation To The Rest Of SQLite

The internal organization of the SQLite library can be viewed as the stack of modules shown to the right. The Tokenizer, Parser, and Code Generator components are used to process SQL statements and convert them into executable programs in a virtual machine language or byte code. Roughly speaking, these top three layers implement sqlite3_prepare_v2(). The byte code generated by the top three layers is a prepared statement. The Virtual Machine module is responsible for running the SQL statement byte code. The B-Tree module organizes a database file into multiple key/value stores with ordered keys and logarithmic performance. The Pager module is responsible for loading pages of the database file into memory, for implementing and controlling transactions, and for creating and maintaining the journal files that prevent database corruption following a crash or power failure. The OS

Interface is a thin abstraction that provides a common set of routines for adapting SQLite to run on different operating systems. Roughly speaking, the bottom four layers implement sqlite3_step().

This article is about the bottom layer.

The OS Interface - also called the "VFS" - is what makes SQLite portable across operating systems. Whenever any of the other modules in SQLite needs to communicate with the operating system, they invoke methods in the VFS. The VFS then invokes the operating-specific code needed to satisfy the request. Hence, porting SQLite to a new operating system is simply a matter of writing a new OS interface layer or "VFS".

2.0 Multiple VFSes

The standard SQLite source tree contains built-in VFSes for unix and windows. Alternative VFSes can be added at start-time or run-time using the sqlite3_vfs_register() interface.

Multiple VFSes can be registered at the same time. Each VFS has a unique names. Separate database connections within the same process can be using different VFSes at the same time. For that matter, if a single database connection has multiple database files open using the ATTACH command, then each attached database might be using a different VFS.

Unix builds come with multiple VFSes built-in. The default VFS for unix is called "unix" and is the VFS used in an overwhelming majority of applications. Other VFSes that can be found in unix may include:

1. **unix-dotfile** - uses dot-file locking rather than POSIX advisory locks.
2. **unix-excl** - obtains and holds an exclusive lock on database files, preventing other processes from accessing the database. Also keeps the wal-index in heap rather than in shared memory.
3. **unix-none** - all file locking operations are no-ops.
4. **unix-namedsem** - uses named semaphores for file locking. VXWorks only.

The various unix VFSes differ only in the way they handle file locking - they share most of their implementation in common with one another and are all located in the same SQLite source file: os_unix.c. Note that except for "unix" and "unix-excl", the various unix VFSes all use incompatible locking implementations. If two processes are accessing the same SQLite database using different unix VFSes, they may not see each others locks and may end up interfering with one another, resulting in database corruption. The "unix-none" VFS in particular does no locking at all and will easily result in database corruption if used by two or more database connections at the same time. Programmers are encouraged to use only "unix" or "unix-excl" unless there is a compelling reason to do otherwise.

2.1 Specifying Which VFS To Use

There is always one VFS which is the default VFS. On unix systems, the "unix" VFS comes up as the default and on windows it is "win32". If no other actions are taken, new database connections will make use of the default VFS.

The default VFS can be changed by registering or re-registering the VFS using the sqlite3_vfs_register() interface with a second parameter of 1. Hence, if a (unix) process wants to always use the "unix-nolock" VFS in place of "unix", the following code would work:

```
sqlite3_vfs_register(sqlite3_vfs_find("unix-nolock"), 1);
```

An alternate VFS can also be specified as the 4th parameter to the sqlite3_open_v2() function. For example:

```
int rc = sqlite3_open_v2("demo.db", &db, SQLITE_OPEN_READWRITE, "unix-nolock");
```

Finally, if URI filenames have been enabled, then the alternative VFS can be specified using the "vfs=" parameter on the URI. This technique works with sqlite3_open(), sqlite3_open16(), sqlite3_open_v2(), and when a new database is ATTACH-ed to an existing database connection. For example:

```
ATTACH 'file:demo2.db?vfs=unix-none' AS demo2;
```

The VFS specified by a URI has the highest priority. After that comes a VFS specified as the fourth argument to sqlite3_open_v2(). The default VFS is used if no VFS is specified otherwise.

2.2 VFS Shims

From the point of view of the uppers layers of the SQLite stack, each open database file uses exactly one VFS. But in practice, a particular VFS might just be a thin wrapper around another VFS that does the real work. We call a wrapper VFS a "shim".

A simple example of a shim is the "vfstrace" VFS. This is a VFS (implemented in the test_vfstrace.c source file) that writes a message associated with each VFS method call into a log file, then passes control off to another VFS to do the actual work.

2.3 Other Example VFSes

The following are other VFS implementations available in the public SQLite source tree:

- test_demovfs.c - This file implements a very simple VFS named "demo" that uses POSIX functions such as open(), read(), write(), fsync(), close(), fsync(), sleep(), time(), and so forth. This VFS only works on unix systems. But it is not intended as a replacement for the standard "unix" VFS used by default on unix platforms. The "demo" VFS is deliberately kept very simple so that it can be used as a learning aid or as template for building other VFSes or for porting SQLite to new operating systems.
- test_quota.c - This file implements a shim called "quota" that enforces cumulative file size limits on a collection of database files. An auxiliary interface is used to define "quote groups". A quota group is a set of files (database files, journals, and temporary files) whose names all match a GLOB pattern. The sum of the sizes of all files in each quota group is tracked, and if that sum exceeds a threshold defined for the quota group, a callback function is invoked. That callback can either increase the threshold or cause the operation that would have exceeded the quota to fail with an SQLITE_FULL error. One of the uses of this shim is used to enforce resource limits on application databases in Firefox.
- test_multiplex.c - This file implements a shim that allows database files to exceed the maximum file size of the underlying filesystem. This shim presents an interface to the upper six layers of SQLite that makes it look like very large files are being used, when in reality each such large file is split up into many smaller files on the underlying system. This shim has been used, for example, to allow databases to grow larger than 2 gibibytes on FAT16 filesystems.
- test_onefile.c - This file implements a demonstration VFS named "fs" that shows how SQLite can be used on an embedded device that lacks a filesystem. Content is written directly to the underlying media. A VFS derived from this demonstration code could be used by a gadget with a limited amount of flash memory to make SQLite behave as the filesystem for the flash memory on the device.
- test_journal.c - This file implements a shim used during SQLite testing that verifies that the database and rollback journal are written in the correct order and are "synced" at appropriate times in order to guarantee that the database can recover from a power lose are hard reset at any time. The shim checks several invariants on the operation of

databases and rollback journals and raises exceptions if any of those invariants are violated. These invariants, in turn, assure that the database is always recoverable. Running a large suite of test cases using this shim provides added assurance that SQLite databases will not be damaged by unexpected power failures or device resets.

- test_vfs.c - This file implements a shim that can be used to simulate filesystem faults. This shim is used during testing to verify that SQLite responses sanely to hardware malfunctions or to other error conditions such as running out of filesystem space that are difficult to test on a real system.

There are other VFS implementations both in the core SQLite source code library and in available extensions. The list above is not meant to be exhaustive but merely representative of the kinds of features that can be realized using the VFS interface.

3.0 VFS Implementations

A new VFS is implemented by subclassing three objects:

- sqlite3_vfs
- sqlite3_io_methods
- sqlite3_file

An sqlite3_vfs object defines the name of the VFS and the core methods that implement the interface to the operating system, such as checking for existence of files, deleting files, creating files and opening and for reading and/or writing, converting filenames into their canonical form. The sqlite3_vfs object also contains methods for obtaining randomness from the operating system, for suspending a process (sleeping) and for finding the current date and time.

The sqlite3_file object represents an open file. The xOpen method of sqlite3_vfs constructs an sqlite3_file object when the file is opened. The sqlite3_file keeps track of the state of the file while it is opened.

The sqlite3_io_methods object holds the methods used to interact with an open file. Each sqlite3_file contains a pointer to an sqlite3_io_methods object that is appropriate for the file it represents. The sqlite3_io_methods object contains methods to do things such as read and write from the file, to truncate the file, to flush any changes to persistent storage, to find the size of the file, to lock and unlock the file, and to close file and destroy the sqlite3_file object.

Writing the code for a new VFS involves constructing a subclass for the sqlite3_vfs object and then registering that VFS object using a call to sqlite3_vfs_register(). The VFS implementation also provides subclasses for sqlite3_file and sqlite3_io_methods but those objects are not registered directly with SQLite. Instead, the sqlite3_file object is returned from the xOpen method of sqlite3_vfs and the sqlite3_file object points to an instance of the sqlite3_io_methods object.

To Be Continued...

The Virtual Table Mechanism Of SQLite

1.0 Introduction

A virtual table is an object that is registered with an open SQLite database connection. From the perspective of an SQL statement, the virtual table object looks like any other table or view. But behind the scenes, queries and updates on a virtual table invoke callback methods of the virtual table object instead of reading and writing to the database file.

The virtual table mechanism allows an application to publish interfaces that are accessible from SQL statements as if they were tables. SQL statements can do almost anything to a virtual table that they can do to a real table, with the following exceptions:

- One cannot create a trigger on a virtual table.
- One cannot create additional indices on a virtual table. (Virtual tables can have indices but that must be built into the virtual table implementation. Indices cannot be added separately using CREATE INDEX statements.)
- One cannot run ALTER TABLE ... ADD COLUMN commands against a virtual table.

Individual virtual table implementations might impose additional constraints. For example, some virtual implementations might provide read-only tables. Or some virtual table implementations might allow INSERT or DELETE but not UPDATE. Or some virtual table implementations might limit the kinds of UPDATEs that can be made.

A virtual table might represent an in-memory data structures. Or it might represent a view of data on disk that is not in the SQLite format. Or the application might compute the content of the virtual table on demand.

Here are some existing and postulated uses for virtual tables:

- A full-text search interface
- Spatial indices using R-Trees
- Introspect the disk content of an SQLite database file (the dbstat virtual table)
- Read and/or write the content of a comma-separated value (CSV) file
- Access the filesystem of the host computer as if it were a database table
- Enabling SQL manipulation of data in statistics packages like R

1.1 Usage

A virtual table is created using a CREATE VIRTUAL TABLE statement.

create-virtual-table-stmt: <button id="x1475" onclick="hideorshow("x1475","x1476")">hide</button>

The CREATE VIRTUAL TABLE statement creates a new table called table-name derived from the class class module-name. The module-name is the name that is registered for the virtual table by the sqlite3_create_module() interface.

```
CREATE VIRTUAL TABLE tablename USING modulename;
```

One can also provide comma-separated arguments to the module following the module name:

```
CREATE VIRTUAL TABLE tablename USING modulename(arg1, arg2, ...);
```

The format of the arguments to the module is very general. Each module-argument may contain keywords, string literals, identifiers, numbers, and punctuation. Each module-argument is passed as written (as text) into the constructor method of the virtual table implementation when the virtual table is created and that constructor is responsible for parsing and interpreting the arguments. The argument syntax is sufficiently general that a virtual table implementation can, if it wants to, interpret its arguments as column definitions in an ordinary CREATE TABLE statement. The implementation could also impose some other interpretation on the arguments.

Once a virtual table has been created, it can be used like any other table with the exceptions noted above and imposed by specific virtual table implementations. A virtual table is destroyed using the ordinary DROP TABLE syntax.

1.1.1 Temporary virtual tables

There is no "CREATE TEMP VIRTUAL TABLE" statement. To create a temporary virtual table, add the "temp" schema before the virtual table name.

```
CREATE VIRTUAL TABLE temp.tablename USING module(arg1, ...);
```

1.1.2 Eponymous virtual tables

Some virtual tables exist automatically in the "main" schema of every database connection in which their module is registered, even without a CREATE VIRTUAL TABLE statement. Such virtual tables are called "eponymous virtual tables". To use an eponymous virtual table, simply use the module name as if it were a table. Eponymous virtual tables exist in the "main" schema only, so they will not work if prefixed with a different schema name.

An example of an eponymous virtual table is the dbstat virtual table. To use the dbstat virtual table as an eponymous virtual table, simply query against the "dbstat" module name, as if it were an ordinary table. (Note that SQLite must be compiled with the SQLITE_ENABLE_DBSTAT_VTAB option to include the dbstat virtual table in the build.)

```
SELECT * FROM dbstat;
```

A virtual table is eponymous if its xCreate method is the exact same function as the xConnect method, or if the xCreate method is NULL. The xCreate method is called when a virtual table is first created using the CREATE VIRTUAL TABLE statement. The xConnect method whenever a database connection attaches to or reparses a schema. When these two methods are the same, that indicates that the virtual table has no persistent state that needs to be created and destroyed.

1.1.2.1 Eponymous-only virtual tables

If the xCreate method is NULL, then CREATE VIRTUAL TABLE statements are prohibited for that virtual table, and the virtual table is an "eponymous-only virtual table". Eponymous-only virtual tables are useful as table-valued functions.

Note that SQLite versions prior to 3.9.0 did not check the xCreate method for NULL before invoking it. So if an eponymous-only virtual table is registered with SQLite version 3.8.11.1 or earlier and a CREATE VIRTUAL TABLE command is attempted against that virtual table module, a jump to a NULL pointer will occur, resulting in a crash.

1.2 Implementation

Several new C-level objects are used by the virtual table implementation:

```
typedef struct sqlite3_vtab sqlite3_vtab;
typedef struct sqlite3_index_info sqlite3_index_info;
typedef struct sqlite3_vtab_cursor sqlite3_vtab_cursor;
typedef struct sqlite3_module sqlite3_module;
```

The sqlite3_module structure defines a module object used to implement a virtual table. Think of a module as a class from which one can construct multiple virtual tables having similar properties. For example, one might have a module that provides read-only access to comma-separated-value (CSV) files on disk. That one module can then be used to create several virtual tables where each virtual table refers to a different CSV file.

The module structure contains methods that are invoked by SQLite to perform various actions on the virtual table such as creating new instances of a virtual table or destroying old ones, reading and writing data, searching for and deleting, updating, or inserting rows. The module structure is explained in more detail below.

Each virtual table instance is represented by an sqlite3_vtab structure. The sqlite3_vtab structure looks like this:

```
struct sqlite3_vtab {
  const sqlite3_module *pModule;
  int nRef;
  char *zErrMsg;
};
```

Virtual table implementations will normally subclass this structure to add additional private and implementation-specific fields. The nRef field is used internally by the SQLite core and should not be altered by the virtual table implementation. The virtual table implementation may pass error message text to the core by putting an error message string in zErrMsg. Space to hold this error message string must be obtained from an SQLite memory allocation function such as sqlite3_mprintf() or sqlite3_malloc(). Prior to assigning a new value to zErrMsg, the virtual table implementation must free any preexisting content of zErrMsg using sqlite3_free(). Failure to do this will result in a memory leak. The SQLite core will free and zero the content of zErrMsg when it delivers the error message text to the client application or when it destroys the virtual table. The virtual table implementation only needs to worry about freeing the zErrMsg content when it overwrites the content with a new, different error message.

The sqlite3_vtab_cursor structure represents a pointer to a specific row of a virtual table. This is what an sqlite3_vtab_cursor looks like:

```
struct sqlite3_vtab_cursor {
  sqlite3_vtab *pVtab;
};
```

Once again, practical implementations will likely subclass this structure to add additional private fields.

The sqlite3_index_info structure is used to pass information into and out of the xBestIndex method of the module that implements a virtual table.

Before a CREATE VIRTUAL TABLE statement can be run, the module specified in that statement must be registered with the database connection. This is accomplished using either of the sqlite3_create_module() or sqlite3_create_module_v2() interfaces:

```
int sqlite3_create_module(
  sqlite3 *db,               /* SQLite connection to register module with */
  const char *zName,         /* Name of the module */
  const sqlite3_module *,    /* Methods for the module */
  void *                     /* Client data for xCreate/xConnect */
);
int sqlite3_create_module_v2(
  sqlite3 *db,               /* SQLite connection to register module with */
  const char *zName,         /* Name of the module */
  const sqlite3_module *,    /* Methods for the module */
  void *,                    /* Client data for xCreate/xConnect */
  void(*xDestroy)(void*)     /* Client data destructor function */
);
```

The sqlite3_create_module() and sqlite3_create_module_v2() routines associates a module name with an sqlite3_module structure and a separate client data that is specific to each module. The only difference between the two create_module methods is that the _v2 method includes an extra parameter that specifies a destructor for client data pointer. The module structure is what defines the behavior of a virtual table. The module structure looks like this:

```

struct sqlite3_module {
  int iVersion;
  int (*xCreate)(sqlite3*, void *pAux,
               int argc, char **argv,
               sqlite3_vtab **ppVTab,
               char **pzErr);
  int (*xConnect)(sqlite3*, void *pAux,
               int argc, char **argv,
               sqlite3_vtab **ppVTab,
               char **pzErr);
  int (*xBestIndex)(sqlite3_vtab *pVTab, sqlite3_index_info*);
  int (*xDisconnect)(sqlite3_vtab *pVTab);
  int (*xDestroy)(sqlite3_vtab *pVTab);
  int (*xOpen)(sqlite3_vtab *pVTab, sqlite3_vtab_cursor **ppCursor);
  int (*xClose)(sqlite3_vtab_cursor*);
  int (*xFilter)(sqlite3_vtab_cursor*, int idxNum, const char *idxStr,
                int argc, sqlite3_value **argv);
  int (*xNext)(sqlite3_vtab_cursor*);
  int (*xEof)(sqlite3_vtab_cursor*);
  int (*xColumn)(sqlite3_vtab_cursor*, sqlite3_context*, int);
  int (*xRowid)(sqlite3_vtab_cursor*, sqlite_int64 *pRowid);
  int (*xUpdate)(sqlite3_vtab *, int, sqlite3_value **, sqlite_int64 *);
  int (*xBegin)(sqlite3_vtab *pVTab);
  int (*xSync)(sqlite3_vtab *pVTab);
  int (*xCommit)(sqlite3_vtab *pVTab);
  int (*xRollback)(sqlite3_vtab *pVTab);
  int (*xFindFunction)(sqlite3_vtab *pVtab, int nArg, const char *zName,
                     void (**pxFunc)(sqlite3_context*,int,sqlite3_value**),
                     void **ppArg);
  int (*Rename)(sqlite3_vtab *pVtab, const char *zNew);
  /* The methods above are in version 1 of the sqlite_module object. Those
  ** below are for version 2 and greater. */
  int (*xSavepoint)(sqlite3_vtab *pVTab, int);
  int (*xRelease)(sqlite3_vtab *pVTab, int);
  int (*xRollbackTo)(sqlite3_vtab *pVTab, int);
};
```

The module structure defines all of the methods for each virtual table object. The module structure also contains the iVersion field which defines the particular edition of the module table structure. Currently, iVersion is always 1, but in future releases of SQLite the module structure definition might be extended with additional methods and in that case the iVersion value will be increased.

The rest of the module structure consists of methods used to implement various features of the virtual table. Details on what each of these methods do are provided in the sequel.

1.3 Virtual Tables And Shared Cache

Prior to SQLite version 3.6.17, the virtual table mechanism assumes that each database connection kept its own copy of the database schema. Hence, the virtual table mechanism could not be used in a database that has shared cache mode enabled. The sqlite3_create_module() interface would return an error if shared cache mode is enabled. That restriction was relaxed beginning with SQLite version 3.6.17.

1.4 Creating New Virtual Table Implementations

Follow these steps to create your own virtual table:

1. Write all necessary methods.
2. Create an instance of the sqlite3_module structure containing pointers to all the methods from step 1.
3. Register your sqlite3_module structure using one of the sqlite3_create_module() or sqlite3_create_module_v2() interfaces.
4. Run a CREATE VIRTUAL TABLE command that specifies the new module in the USING clause.

The only really hard part is step 1. You might want to start with an existing virtual table implementation and modify it to suit your needs. There are several virtual table implementations in the SQLite source tree (for testing purposes). You might use one of those as a guide. Locate these test virtual table implementations by searching for "sqlite3_create_module".

You might also want to implement your new virtual table as a loadable extension.

2.0 Virtual Table Methods

2.1 The xCreate Method

```
int (*xCreate)(sqlite3 *db, void *pAux,
             int argc, char **argv,
             sqlite3_vtab **ppVTab,
             char **pzErr);
```

This method is called to create a new instance of a virtual table in response to a CREATE VIRTUAL TABLE statement. The db parameter is a pointer to the SQLite database connection that is executing the CREATE VIRTUAL TABLE statement. The pAux argument is the copy of the client data pointer that was the fourth argument to the sqlite3_create_module() or sqlite3_create_module_v2() call that registered the virtual table module. The argv parameter is an array of argc pointers to null terminated strings. The first string, argv[0], is the name of the module being invoked. The module name is the name provided as the second argument to sqlite3_create_module() and as the argument to the USING clause of the CREATE VIRTUAL TABLE statement that is running. The second, argv[1], is the name of the database in which the new virtual table is being created. The database name is "main" for the primary database, or "temp" for TEMP database, or the name given at the end of the ATTACH statement for attached databases. The third element of the array, argv[2], is the name of the new virtual table, as specified following the TABLE keyword in the CREATE VIRTUAL TABLE statement. If present, the fourth and subsequent strings in the argv[] array report the arguments to the module name in the CREATE VIRTUAL TABLE statement.

The job of this method is to construct the new virtual table object (an sqlite3_vtab object) and return a pointer to it in *ppVTab.

As part of the task of creating a new sqlite3_vtab structure, this method must invoke sqlite3_declare_vtab() to tell the SQLite core about the columns and datatypes in the virtual table. The sqlite3_declare_vtab() API has the following prototype:

```
  int sqlite3_declare_vtab(sqlite3 *db, const char *zCreateTable)
```

The first argument to sqlite3_declare_vtab() must be the same database connection pointer as the first parameter to this method. The second argument to sqlite3_declare_vtab() must a zero-terminated UTF-8 string that contains a well-formed CREATE TABLE statement that defines the columns in the virtual table and their data types. The name of the table in this CREATE TABLE statement is ignored, as are all constraints. Only the column names and datatypes matter. The CREATE TABLE statement string need not to be held in persistent memory. The string can be deallocated and/or reused as soon as the sqlite3_declare_vtab() routine returns.

The xCreate method need not initialize the pModule, nRef, and zErrMsg fields of the sqlite3_vtab object. The SQLite core will take care of that chore.

The xCreate should return SQLITE_OK if it is successful in creating the new virtual table, or SQLITE_ERROR if it is not successful. If not successful, the sqlite3_vtab structure must not be allocated. An error message may optionally be returned in *pzErr if unsuccessful. Space to hold the error message string must be allocated using an SQLite memory allocation function like sqlite3_malloc() or sqlite3_mprintf() as the SQLite core will attempt to free the space using sqlite3_free() after the error has been reported up to the application.

If the xCreate method is omitted (left as a NULL pointer) then the virtual table is an eponymous-only virtual table. New instances of the virtual table cannot be created using CREATE VIRTUAL TABLE and the virtual table can only be used via its module name. Note that SQLite versions prior to 3.9.0 do not understand eponymous-only virtual tables and will segfault if an attempt is made to CREATE VIRTUAL TABLE on an eponymous-only virtual table because the xCreate method was not checked for null.

If the xCreate method is the exact same pointer as the xConnect method, that indicates that the virtual table does not need to initialize backing store. Such a virtual table can be used as an eponymous virtual table or as a named virtual table using CREATE VIRTUAL TABLE or both.

2.1.1 Hidden columns in virtual tables

If a column datatype contains the special keyword "HIDDEN" (in any combination of upper and lower case letters) then that keyword it is omitted from the column datatype name and the column is marked as a hidden column internally. A hidden column differs from a normal column in three respects:

- Hidden columns are not listed in the dataset returned by "PRAGMA table_info",
- Hidden columns are not included in the expansion of a "*" expression in the result set of a SELECT, and
- Hidden columns are not included in the implicit column-list used by an INSERT statement that lacks an explicit column-list.

For example, if the following SQL is passed to sqlite3_declare_vtab():

```
    CREATE TABLE x(a HIDDEN VARCHAR(12), b INTEGER, c INTEGER Hidden);
```

Then the virtual table would be created with two hidden columns, and with datatypes of "VARCHAR(12)" and "INTEGER".

An example use of hidden columns can be seen in the FTS3 virtual table implementation, where every FTS virtual table contains an FTS hidden column that is used to pass information from the virtual table into FTS auxiliary functions and to the FTS MATCH operator.

2.1.2 Table-valued functions

A virtual table that contains hidden columns can be used like a table-valued function in the FROM clause of a SELECT statement. The arguments to the table-valued function become constraints on the HIDDEN columns of the virtual table.

For example, the "generate_series" extension (located in the ext/misc/series.c file in the source tree) implements an eponymous virtual table with the following schema:

```
CREATE TABLE generate_series(
  value,
  start HIDDEN,
  stop HIDDEN,
  step HIDDEN
);
```

The sqlite3_module.xBestIndex method in the implementation of this table checks for equality constraints against the HIDDEN columns, and uses those as input parameters to determine the range of integer "value" outputs to generate. Reasonable defaults are used for any unconstrained columns. For example, to list all integers between 5 and 50:

```
SELECT value FROM generate_series(5,50);
```

The previous query is equivalent to the following:

```
SELECT value FROM generate_series WHERE start=5 AND stop=50;
```

Arguments on the virtual table name are matched to hidden columns in order. The number of arguments can be less than the number of hidden columns, in which case the latter hidden columns are unconstrained. However, an error results if there are more arguments than there are hidden columns in the virtual table.

2.2 The xConnect Method

```
  int (*xConnect)(sqlite3*, void *pAux,
               int argc, char **argv,
               sqlite3_vtab **ppVTab,
               char **pzErr);
```

The xConnect method is very similar to xCreate. It has the same parameters and constructs a new sqlite3_vtab structure just like xCreate. And it must also call sqlite3_declare_vtab() like xCreate.

The difference is that xConnect is called to establish a new connection to an existing virtual table whereas xCreate is called to create a new virtual table from scratch.

The xCreate and xConnect methods are only different when the virtual table has some kind of backing store that must be initialized the first time the virtual table is created. The xCreate method creates and initializes the backing store. The xConnect method just connects to an existing backing store. When xCreate and xConnect are the same, the table is an eponymous virtual table.

As an example, consider a virtual table implementation that provides read-only access to existing comma-separated-value (CSV) files on disk. There is no backing store that needs to be created or initialized for such a virtual table (since the CSV files already exist on disk) so the xCreate and xConnect methods will be identical for that module.

Another example is a virtual table that implements a full-text index. The xCreate method must create and initialize data structures to hold the dictionary and posting lists for that index. The xConnect method, on the other hand, only has to locate and use an existing dictionary and posting lists that were created by a prior xCreate call.

The xConnect method must return SQLITE_OK if it is successful in creating the new virtual table, or SQLITE_ERROR if it is not successful. If not successful, the sqlite3_vtab structure must not be allocated. An error message may optionally be returned in *pzErr if unsuccessful. Space to hold the error message string must be allocated using an SQLite memory allocation function like sqlite3_malloc() or sqlite3_mprintf() as the SQLite core will attempt to free the space using sqlite3_free() after the error has been reported up to the application.

The xConnect method is required for every virtual table implementation, though the xCreate and xConnect pointers of the sqlite3_module object may point to the same function if the virtual table does not need to initialize backing store.

2.3 The xBestIndex Method

SQLite uses the xBestIndex method of a virtual table module to determine the best way to access the virtual table. The xBestIndex method has a prototype like this:

```
int (*xBestIndex)(sqlite3_vtab *pVTab, sqlite3_index_info*);
```

The SQLite core communicates with the xBestIndex method by filling in certain fields of the sqlite3_index_info structure and passing a pointer to that structure into xBestIndex as the second parameter. The xBestIndex method fills out other fields of this structure which forms the reply. The sqlite3_index_info structure looks like this:

```
struct sqlite3_index_info {
  /* Inputs */
  const int nConstraint;     /* Number of entries in aConstraint */
  const struct sqlite3_index_constraint {
     int iColumn;              /* Column constrained.  -1 for ROWID */
     unsigned char op;         /* Constraint operator */
     unsigned char usable;     /* True if this constraint is usable */
     int iTermOffset;          /* Used internally - xBestIndex should ignore */
  } *const aConstraint;      /* Table of WHERE clause constraints */
  const int nOrderBy;        /* Number of terms in the ORDER BY clause */
  const struct sqlite3_index_orderby {
     int iColumn;              /* Column number */
     unsigned char desc;       /* True for DESC.  False for ASC. */
  } *const aOrderBy;         /* The ORDER BY clause */

  /* Outputs */
  struct sqlite3_index_constraint_usage {
    int argvIndex;           /* if &gt;0, constraint is part of argv to xFilter */
    unsigned char omit;      /* Do not code a test for this constraint */
  } *const aConstraintUsage;
  int idxNum;                /* Number used to identify the index */
  char *idxStr;              /* String, possibly obtained from sqlite3_malloc */
  int needToFreeIdxStr;      /* Free idxStr using sqlite3_free() if true */
  int orderByConsumed;       /* True if output is already ordered */
  double estimatedCost;      /* Estimated cost of using this index */
  /* Fields below are only available in SQLite 3.8.2 and later */
  sqlite3_int64 estimatedRows;    /* Estimated number of rows returned */
  /* Fields below are only available in SQLite 3.9.0 and later */
  int idxFlags;              /* Mask of SQLITE_INDEX_SCAN_* flags */
  /* Fields below are only available in SQLite 3.10.0 and later */
  sqlite3_uint64 colUsed;    /* Input: Mask of columns used by statement */
};
```

Note the warnings on the "estimatedRows", "idxFlags", and colUsed fields. These fields were added with SQLite versions 3.8.2, 3.9.0, and 3.10.0, respectively. Any extension that reads or writes these fields must first check that the version of the SQLite library in use is greater than or equal to appropriate version - perhaps comparing the value returned from sqlite3_libversion_number() against constants 3008002, 3009000, and/or 3010000. The result of attempting to access these fields in an sqlite3_index_info structure created by an older version of SQLite are undefined.

In addition, there are some defined constants:

```
#define SQLITE_INDEX_CONSTRAINT_EQ      2
#define SQLITE_INDEX_CONSTRAINT_GT      4
#define SQLITE_INDEX_CONSTRAINT_LE      8
#define SQLITE_INDEX_CONSTRAINT_LT     16
#define SQLITE_INDEX_CONSTRAINT_GE     32
#define SQLITE_INDEX_CONSTRAINT_MATCH  64
#define SQLITE_INDEX_CONSTRAINT_LIKE   65     /* 3.10.0 and later only */
#define SQLITE_INDEX_CONSTRAINT_GLOB   66     /* 3.10.0 and later only */
#define SQLITE_INDEX_CONSTRAINT_REGEXP 67     /* 3.10.0 and later only */
#define SQLITE_INDEX_SCAN_UNIQUE        1     /* Scan visits at most 1 row */
```

The SQLite core calls the xBestIndex method when it is compiling a query that involves a virtual table. In other words, SQLite calls this method when it is running sqlite3_prepare() or the equivalent. By calling this method, the SQLite core is saying to the virtual table that it needs to access some subset of the rows in the virtual table and it wants to know the most efficient way to do that access. The xBestIndex method replies with information that the SQLite core can then use to conduct an efficient search of the virtual table.

While compiling a single SQL query, the SQLite core might call xBestIndex multiple times with different settings in sqlite3_index_info. The SQLite core will then select the combination that appears to give the best performance.

Before calling this method, the SQLite core initializes an instance of the sqlite3_index_info structure with information about the query that it is currently trying to process. This information derives mainly from the WHERE clause and ORDER BY or GROUP BY clauses of the query, but also from any ON or USING clauses if the query is a join. The information that the SQLite core provides to the xBestIndex method is held in the part of the structure that is marked as "Inputs". The "Outputs" section is initialized to zero.

The information in the sqlite3_index_info structure is ephemeral and may be overwritten or deallocated as soon as the xBestIndex method returns. If the xBestIndex method needs to remember any part of the sqlite3_index_info structure, it should make a copy. Care must be take to store the copy in a place where it will be deallocated, such as in the idxStr field with needToFreeIdxStr set to 1.

Note that xBestIndex will always be called before xFilter, since the idxNum and idxStr outputs from xBestIndex are required inputs to xFilter. However, there is no guarantee that xFilter will be called following a successful xBestIndex.

The xBestIndex method is required for every virtual table implementation.

2.3.1 Inputs

The main thing that the SQLite core is trying to communicate to the virtual table is the constraints that are available to limit the number of rows that need to be searched. The aConstraint[] array contains one entry for each constraint. There will be exactly nConstraint entries in that array.

Each constraint will correspond to a term in the WHERE clause or in a USING or ON clause that is of the form

> column OP EXPR

Where "column" is a column in the virtual table, OP is an operator like "=" or "<", and="" expr="" is="" an="" arbitrary="" expression.="" so,="" for="" example,="" if="" the="" where="" clause="" contained="" a="" term="" like="" this:=""

```
    a = 5
```

Then one of the constraints would be on the "a" column with operator "=" and an expression of "5". Constraints need not have a literal representation of the WHERE clause. The query optimizer might make transformations to the WHERE clause in order to extract as many constraints as it can. So, for example, if the WHERE clause contained something like this:

```
       x BETWEEN 10 AND 100 AND 999&gt;y
```

The query optimizer might translate this into three separate constraints:

```
       x &gt;= 10
       x &lt;= 100="" 999="" y="" &lt;="" pre=""&gt;
```

For each constraint, the aConstraint[].iColumn field indicates which column appears on the left-hand side of the constraint. The first column of the virtual table is column 0. The rowid of the virtual table is column -1. The aConstraint[].op field indicates which operator is used. The SQLITE*INDEX_CONSTRAINT** constants map integer constants into operator values. Columns occur in the order they were defined by the call to sqlite3_declare_vtab() in the xCreate or xConnect method. Hidden columns are counted when determining the column index.

The aConstraint[] array contains information about all constraints that apply to the virtual table. But some of the constraints might not be usable because of the way tables are ordered in a join. The xBestIndex method must therefore only consider constraints that have an aConstraint[].usable flag which is true.

In addition to WHERE clause constraints, the SQLite core also tells the xBestIndex method about the ORDER BY clause. (In an aggregate query, the SQLite core might put in GROUP BY clause information in place of the ORDER BY clause information, but this fact should not make any difference to the xBestIndex method.) If all terms of the ORDER BY clause are columns in the virtual table, then nOrderBy will be the number of terms in the ORDER BY clause and the aOrderBy[] array will identify the column for each term in the order by clause and whether or not that column is ASC or DESC.

In SQLite version 3.10.0 and later, the colUsed field is available to indicate which fields of the virtual table are actually used by the statement being prepared. If the lowest bit of colUsed is set, that means that the first column is used. The second lowest bit corresponds to the second column. And so forth. If the most significant bit of colUsed is set, that means that one or more columns other than the first 63 columns are used. If column usage information is needed by the xFilter method, then the required bits must be encoded into either the idxNum or idxStr output fields.

2.3.2 Outputs

Given all of the information above, the job of the xBestIndex method it to figure out the best way to search the virtual table.

The xBestIndex method fills the idxNum and idxStr fields with information that communicates an indexing strategy to the xFilter method. The information in idxNum and idxStr is arbitrary as far as the SQLite core is concerned. The SQLite core just copies the information through to the xFilter method. Any desired meaning can be assigned to idxNum and idxStr as long as xBestIndex and xFilter agree on what that meaning is.

The idxStr value may be a string obtained from an SQLite memory allocation function such as sqlite3_mprintf(). If this is the case, then the needToFreeIdxStr flag must be set to true so that the SQLite core will know to call sqlite3_free() on that string when it has finished with it, and thus avoid a memory leak.

If the virtual table will output rows in the order specified by the ORDER BY clause, then the orderByConsumed flag may be set to true. If the output is not automatically in the correct order then orderByConsumed must be left in its default false setting. This will indicate to the SQLite core that it will need to do a separate sorting pass over the data after it comes out of the virtual table.

The estimatedCost field should be set to the estimated number of disk access operations required to execute this query against the virtual table. The SQLite core will often call xBestIndex multiple times with different constraints, obtain multiple cost estimates, then choose the query plan that gives the lowest estimate.

If the current version of SQLite is 3.8.2 or greater, the estimatedRows field may be set to an estimate of the number of rows returned by the proposed query plan. If this value is not explicitly set, the default estimate of 25 rows is used.

If the current version of SQLite is 3.9.0 or greater, the idxFlags field may be set to SQLITE_INDEX_SCAN_UNIQUE to indicate that the virtual table will return only zero or one rows given the input constraints. Additional bits of the idxFlags field might be understood in later versions of SQLite.

The aConstraintUsage[] array contains one element for each of the nConstraint constraints in the inputs section of the sqlite3_index_info structure. The aConstraintUsage[] array is used by xBestIndex to tell the core how it is using the constraints.

The xBestIndex method may set aConstraintUsage[].argvIndex entries to values greater than zero. Exactly one entry should be set to 1, another to 2, another to 3, and so forth up to as many or as few as the xBestIndex method wants. The EXPR of the corresponding constraints will then be passed in as the argv[] parameters to xFilter.

For example, if the aConstraint[3].argvIndex is set to 1, then when xFilter is called, the argv[0] passed to xFilter will have the EXPR value of the aConstraint[3] constraint.

By default, the SQLite core double checks all constraints on each row of the virtual table that it receives. If such a check is redundant, the xBestFilter method can suppress that double-check by setting aConstraintUsage[].omit.

2.4 The xDisconnect Method

```
int (*xDisconnect)(sqlite3_vtab *pVTab);
```

This method releases a connection to a virtual table. Only the sqlite3_vtab object is destroyed. The virtual table is not destroyed and any backing store associated with the virtual table persists. This method undoes the work of xConnect.

This method is a destructor for a connection to the virtual table. Contrast this method with xDestroy. The xDestroy is a destructor for the entire virtual table.

The xDisconnect method is required for every virtual table implementation, though it is acceptable for the xDisconnect and xDestroy methods to be the same function if that makes sense for the particular virtual table.

2.5 The xDestroy Method

```
int (*xDestroy)(sqlite3_vtab *pVTab);
```

This method releases a connection to a virtual table, just like the xDisconnect method, and it also destroys the underlying table implementation. This method undoes the work of xCreate.

The xDisconnect method is called whenever a database connection that uses a virtual table is closed. The xDestroy method is only called when a DROP TABLE statement is executed against the virtual table.

The xDestroy method is required for every virtual table implementation, though it is acceptable for the xDisconnect and xDestroy methods to be the same function if that makes sense for the particular virtual table.

2.6 The xOpen Method

```
int (*xOpen)(sqlite3_vtab *pVTab, sqlite3_vtab_cursor **ppCursor);
```

The xOpen method creates a new cursor used for accessing (read and/or writing) a virtual table. A successful invocation of this method will allocate the memory for the sqlite3_vtab_cursor (or a subclass), initialize the new object, and make *ppCursor point to the new object. The successful call then returns SQLITE_OK.

For every successful call to this method, the SQLite core will later invoke the xClose method to destroy the allocated cursor.

The xOpen method need not initialize the pVtab field of the sqlite3_vtab_cursor structure. The SQLite core will take care of that chore automatically.

A virtual table implementation must be able to support an arbitrary number of simultaneously open cursors.

When initially opened, the cursor is in an undefined state. The SQLite core will invoke the xFilter method on the cursor prior to any attempt to position or read from the cursor.

The xOpen method is required for every virtual table implementation.

2.7 The xClose Method

```
int (*xClose)(sqlite3_vtab_cursor*);
```

The xClose method closes a cursor previously opened by xOpen. The SQLite core will always call xClose once for each cursor opened using xOpen.

This method must release all resources allocated by the corresponding xOpen call. The routine will not be called again even if it returns an error. The SQLite core will not use the sqlite3_vtab_cursor again after it has been closed.

The xClose method is required for every virtual table implementation.

2.8 The xEof Method

```
int (*xEof)(sqlite3_vtab_cursor*);
```

The xEof method must return false (zero) if the specified cursor currently points to a valid row of data, or true (non-zero) otherwise. This method is called by the SQL engine immediately after each xFilter and xNext invocation.

The xEof method is required for every virtual table implementation.

2.9 The xFilter Method

```
int (*xFilter)(sqlite3_vtab_cursor*, int idxNum, const char *idxStr,
              int argc, sqlite3_value **argv);
```

This method begins a search of a virtual table. The first argument is a cursor opened by xOpen. The next two arguments define a particular search index previously chosen by xBestIndex. The specific meanings of idxNum and idxStr are unimportant as long as xFilter and xBestIndex agree on what that meaning is.

The xBestIndex function may have requested the values of certain expressions using the aConstraintUsage[].argvIndex values of the sqlite3_index_info structure. Those values are passed to xFilter using the argc and argv parameters.

If the virtual table contains one or more rows that match the search criteria, then the cursor must be left point at the first row. Subsequent calls to xEof must return false (zero). If there are no rows match, then the cursor must be left in a state that will cause the xEof to return true (non-zero). The SQLite engine will use the xColumn and xRowid methods to access that row content. The xNext method will be used to advance to the next row.

This method must return SQLITE_OK if successful, or an sqlite error code if an error occurs.

The xFilter method is required for every virtual table implementation.

2.10 The xNext Method

```
int (*xNext)(sqlite3_vtab_cursor*);
```

The xNext method advances a virtual table cursor to the next row of a result set initiated by xFilter. If the cursor is already pointing at the last row when this routine is called, then the cursor no longer points to valid data and a subsequent call to the xEof method must return true (non-zero). If the cursor is successfully advanced to another row of content, then subsequent calls to xEof must return false (zero).

This method must return SQLITE_OK if successful, or an sqlite error code if an error occurs.

The xNext method is required for every virtual table implementation.

2.11 The xColumn Method

```
int (*xColumn)(sqlite3_vtab_cursor*, sqlite3_context*, int N);
```

The SQLite core invokes this method in order to find the value for the N-th column of the current row. N is zero-based so the first column is numbered 0. The xColumn method may return its result back to SQLite using one of the following interface:

- sqlite3_result_blob()
- sqlite3_result_double()
- sqlite3_result_int()
- sqlite3_result_int64()
- sqlite3_result_null()
- sqlite3_result_text()
- sqlite3_result_text16()
- sqlite3_result_text16le()
- sqlite3_result_text16be()
- sqlite3_result_zeroblob()

If the xColumn method implementation calls none of the functions above, then the value of the column defaults to an SQL NULL.

To raise an error, the xColumn method should use one of the result_text() methods to set the error message text, then return an appropriate error code. The xColumn method must return SQLITE_OK on success.

The xColumn method is required for every virtual table implementation.

2.12 The xRowid Method

```
int (*xRowid)(sqlite3_vtab_cursor *pCur, sqlite_int64 *pRowid);
```

A successful invocation of this method will cause *pRowid to be filled with the rowid of row that the virtual table cursor pCur is currently pointing at. This method returns SQLITE_OK on success. It returns an appropriate error code on failure.

The xRowid method is required for every virtual table implementation.

2.13 The xUpdate Method

```
int (*xUpdate)(
  sqlite3_vtab *pVTab,
  int argc,
  sqlite3_value **argv,
  sqlite_int64 *pRowid
);
```

All changes to a virtual table are made using the xUpdate method. This one method can be used to insert, delete, or update.

The argc parameter specifies the number of entries in the argv array. The value of argc will be 1 for a pure delete operation or N+2 for an insert or replace or update where N is the number of columns in the table. In the previous sentence, N includes any hidden columns.

Every argv entry will have a non-NULL value in C but may contain the SQL value NULL. In other words, it is always true that `argv[i]!=0` for **i** between 0 and `argc-1`. However, it might be the case that `sqlite3_value_type(argv[i])==SQLITE_NULL`.

The argv[0] parameter is the rowid of a row in the virtual table to be deleted. If argv[0] is an SQL NULL, then no deletion occurs.

The argv[1] parameter is the rowid of a new row to be inserted into the virtual table. If argv[1] is an SQL NULL, then the implementation must choose a rowid for the newly inserted row. Subsequent argv[] entries contain values of the columns of the virtual table, in the order that the columns were declared. The number of columns will match the table declaration that the xConnect or xCreate method made using the sqlite3_declare_vtab() call. All hidden columns are included.

When doing an insert without a rowid (argc>1, argv[1] is an SQL NULL), the implementation must set *pRowid to the rowid of the newly inserted row; this will become the value returned by the sqlite3_last_insert_rowid() function. Setting this value in all the other cases is a harmless no-op; the SQLite engine ignores the* pRowid return value if argc==1 or argv[1] is not an SQL NULL.

Each call to xUpdate will fall into one of cases shown below. Not that references to **argv[i]** mean the SQL value held within the argv[i] object, not the argv[i] object itself.

> **argc = 1**
>
> The single row with rowid equal to argv[0] is deleted. No insert occurs.
>
> **argc > 1 argv[0] = NULL**
>
> A new row is inserted with a rowid argv[1] and column values in argv[2] and following. If argv[1] is an SQL NULL, the a new unique rowid is generated automatically.
>
> **argc > 1 argv[0] ≠ NULL argv[0] = argv[1]**
>
> The row with rowid argv[0] is updated with new values in argv[2] and following parameters.
>
> **argc > 1 argv[0] ≠ NULL argv[0] ≠ argv[1]**
>
> The row with rowid argv[0] is updated with rowid argv[1] and new values in argv[2] and following parameters. This will occur when an SQL statement updates a rowid, as in the statement:
>
> > UPDATE table SET rowid=rowid+1 WHERE ...;

The xUpdate method must return SQLITE_OK if and only if it is successful. If a failure occurs, the xUpdate must return an appropriate error code. On a failure, the pVTab->zErrMsg element may optionally be replaced with error message text stored in memory allocated from SQLite using functions such as sqlite3_mprintf() or sqlite3_malloc().

If the xUpdate method violates some constraint of the virtual table (including, but not limited to, attempting to store a value of the wrong datatype, attempting to store a value that is too large or too small, or attempting to change a read-only value) then the xUpdate must fail with an appropriate error code.

There might be one or more sqlite3_vtab_cursor objects open and in use on the virtual table instance and perhaps even on the row of the virtual table when the xUpdate method is invoked. The implementation of xUpdate must be prepared for attempts to delete or modify rows of the table out from other existing cursors. If the virtual table cannot accommodate such changes, the xUpdate method must return an error code.

The xUpdate method is optional. If the xUpdate pointer in the sqlite3_module for a virtual table is a NULL pointer, then the virtual table is read-only.

2.14 The xFindFunction Method

```
int (*xFindFunction)(
  sqlite3_vtab *pVtab,
  int nArg,
  const char *zName,
  void (**pxFunc)(sqlite3_context*,int,sqlite3_value**),
  void **ppArg
);
```

This method is called during sqlite3_prepare() to give the virtual table implementation an opportunity to overload functions. This method may be set to NULL in which case no overloading occurs.

When a function uses a column from a virtual table as its first argument, this method is called to see if the virtual table would like to overload the function. The first three parameters are inputs: the virtual table, the number of arguments to the function, and the name of the function. If no overloading is desired, this method returns 0. To overload the function, this method writes the new function implementation into *pxFunc and writes user data into* ppArg and returns 1.

Note that infix functions (LIKE, GLOB, REGEXP, and MATCH) reverse the order of their arguments. So "like(A,B)" is equivalent to "B like A". For the form "B like A" the B term is considered the first argument to the function. But for "like(A,B)" the A term is considered the first argument.

The function pointer returned by this routine must be valid for the lifetime of the sqlite3_vtab object given in the first parameter.

2.15 The xBegin Method

```
int (*xBegin)(sqlite3_vtab *pVTab);
```

This method begins a transaction on a virtual table. This is method is optional. The xBegin pointer of sqlite3_module may be NULL.

This method is always followed by one call to either the xCommit or xRollback method. Virtual table transactions do not nest, so the xBegin method will not be invoked more than once on a single virtual table without an intervening call to either xCommit or xRollback. Multiple calls to other methods can and likely will occur in between the xBegin and the corresponding xCommit or xRollback.

2.16 The xSync Method

```
int (*xSync)(sqlite3_vtab *pVTab);
```

This method signals the start of a two-phase commit on a virtual table. This is method is optional. The xSync pointer of sqlite3_module may be NULL.

This method is only invoked after call to the xBegin method and prior to an xCommit or xRollback. In order to implement two-phase commit, the xSync method on all virtual tables is invoked prior to invoking the xCommit method on any virtual table. If any of the xSync methods fail, the entire transaction is rolled back.

2.17 The xCommit Method

```
int (*xCommit)(sqlite3_vtab *pVTab);
```

This method causes a virtual table transaction to commit. This is method is optional. The xCommit pointer of sqlite3_module may be NULL.

A call to this method always follows a prior call to xBegin and xSync.

2.18 The xRollback Method

```
int (*xRollback)(sqlite3_vtab *pVTab);
```

This method causes a virtual table transaction to rollback. This is method is optional. The xRollback pointer of sqlite3_module may be NULL.

A call to this method always follows a prior call to xBegin.

2.19 The xRename Method

```
int (*xRename)(sqlite3_vtab *pVtab, const char *zNew);
```

This method provides notification that the virtual table implementation that the virtual table will be given a new name. If this method returns SQLITE_OK then SQLite renames the table. If this method returns an error code then the renaming is prevented.

The xRename method is required for every virtual table implementation.

2.20 The xSavepoint, xRelease, and xRollbackTo Methods

```
int (*xSavepoint)(sqlite3_vtab *pVtab, int);
int (*xRelease)(sqlite3_vtab *pVtab, int);
int (*xRollbackTo)(sqlite3_vtab *pVtab, int);
```

These methods provide the virtual table implementation an opportunity to implement nested transactions. They are always optional and will only be called in SQLite version 3.7.7 and later.

When xSavepoint(X,N) is invoked, that is a signal to the virtual table X that it should save its current state as savepoint N. A subsequent call to xRollbackTo(X,R) means that the state of the virtual table should return to what it was when xSavepoint(X,R) was last called. The call to xRollbackTo(X,R) will invalidate all savepoints with N>R; none of the invalided savepoints will be rolled back or released without first being reinitialized by a call to xSavepoint(). A call to xRelease(X,M) invalidates all savepoints where N>=M.

None of the xSavepoint(), xRelease(), or xRollbackTo() methods will ever be called except in between calls to xBegin() and either xCommit() or xRollback().

The SQLite Database File Format

This document describes and defines the on-disk database file format used by SQLite.

1.0 The Database File

The complete state of an SQLite database is usually contained a single file on disk called the "main database file".

During a transaction, SQLite stores additional information in a second file called the "rollback journal", or if SQLite is in WAL mode, a write-ahead log file. If the application or host computer crashes before the transaction completes, then the rollback journal or write-ahead log contains critical state information needed to restore the main database file to a consistent state. When a rollback journal or write-ahead log contains information necessary for recovering the state of the database, they are called a "hot journal" or "hot WAL file". Hot journals and WAL files are only a factor during error recovery scenarios and so are uncommon, but they are part of the state of an SQLite database and so cannot be ignored. This document defines the format of a rollback journal and the write-ahead log file, but the focus is on the main database file.

1.1 Pages

The main database file consists of one or more pages. The size of a page is a power of two between 512 and 65536 inclusive. All pages within the same database are the same size. The page size for a database file is determined by the 2-byte integer located at an offset of 16 bytes from the beginning of the database file.

Pages are numbered beginning with 1. The maximum page number is 2147483646 (2<sup><small>31</small></sup> - 2). The minimum size SQLite database is a single 512-byte page. The maximum size database would be 2147483646 pages at 65536 bytes per page or 140,737,488,224,256 bytes (about 140 terabytes). Usually SQLite will hit the maximum file size limit of the underlying filesystem or disk hardware size limit long before it hits its own internal size limit.

In common use, SQLite databases tend to range in size from a few kilobytes to a few gigabytes.

At any point in time, every page in the main database has a single use which is one of the following:

- The lock-byte page
- A freelist page
    - A freelist trunk page
    - A freelist leaf page
- A b-tree page
    - A table b-tree interior page
    - A table b-tree leaf page
    - An index b-tree interior page
    - An index b-tree leaf page
- A payload overflow page
- A pointer map page

All reads from and writes to the main database file begin at a page boundary and all writes are an integer number of pages in size. Reads are also usually an integer number of pages in size, with the one exception that when the database is first opened, the first 100 bytes of the database file (the database file header) are read as a sub-page size unit.

Before any information-bearing page of the database is modified, the original unmodified content of that page is written into the rollback journal. If a transaction is interrupted and needs to be rolled back, the rollback journal can then be used to restore the database to its original state. Freelist leaf pages bear no information that would need to be restored on a rollback and so they are not written to the journal prior to modification, in order to reduce disk I/O.

1.2 The Database Header

The first 100 bytes of the database file comprise the database file header. The database file header is divided into fields as shown by the table below. All multibyte fields in the database file header are stored with the most significant byte first (big-endian).

Database Header Format

Offset	Size	Description
0	16	The header string: "SQLite format 3\000"
16	2	The database page size in bytes. Must be a power of two between 512 and 32768 inclusive, or the value 1 representing a page size of 65536.
18	1	File format write version. 1 for legacy; 2 for WAL.
19	1	File format read version. 1 for legacy; 2 for WAL.
20	1	Bytes of unused "reserved" space at the end of each page. Usually 0.
21	1	Maximum embedded payload fraction. Must be 64.
22	1	Minimum embedded payload fraction. Must be 32.
23	1	Leaf payload fraction. Must be 32.
24	4	File change counter.
28	4	Size of the database file in pages. The "in-header database size".
32	4	Page number of the first freelist trunk page.
36	4	Total number of freelist pages.
40	4	The schema cookie.
44	4	The schema format number. Supported schema formats are 1, 2, 3, and 4.
48	4	Default page cache size.
52	4	The page number of the largest root b-tree page when in auto-vacuum or incremental-vacuum modes, or zero otherwise.
56	4	The database text encoding. A value of 1 means UTF-8. A value of 2 means UTF-16le. A value of 3 means UTF-16be.
60	4	The "user version" as read and set by the user_version pragma.
64	4	True (non-zero) for incremental-vacuum mode. False (zero) otherwise.
68	4	The "Application ID" set by PRAGMA application_id.
72	20	Reserved for expansion. Must be zero.
92	4	The version-valid-for number.
96	4	SQLITE_VERSION_NUMBER

1.2.1 Magic Header String

Every valid SQLite database file begins with the following 16 bytes (in hex): 53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00. This byte sequence corresponds to the UTF-8 string "SQLite format 3" including the nul terminator character at the end.

1.2.2 Page Size

The two-byte value beginning at offset 16 determines the page size of the database. For SQLite versions 3.7.0.1 and earlier, this value is interpreted as a big-endian integer and must be a power of two between 512 and 32768, inclusive. Beginning with SQLite version 3.7.1, a page size of 65536 bytes is supported. The value 65536 will not fit in a two-byte integer, so to specify a 65536-byte page size, the value at offset 16 is 0x00 0x01. This value can be interpreted as a big-endian 1 and thought of is as a magic number to represent the 65536 page size. Or one can view the two-byte field as a little endian number and say that it represents the page size divided by 256. These two interpretations of the page-size field are equivalent.

1.2.3 File format version numbers

The file format write version and file format read version at offsets 18 and 19 are intended to allow for enhancements of the file format in future versions of SQLite. In current versions of SQLite, both of these values are 1 for rollback journalling modes and 2 for WAL journalling mode. If a version of SQLite coded to the current file format specification encounters a database file where the read version is 1 or 2 but the write version is greater than 2, then the database file must be treated as read-only. If a database file with a read version greater than 2 is encountered, then that database cannot be read or written.

1.2.4 Reserved bytes per page

SQLite has the ability to set aside a small number of extra bytes at the end of every page for use by extensions. These extra bytes are used, for example, by the SQLite Encryption Extension to store a nonce and/or cryptographic checksum associated with each page. The "reserved space" size in the 1-byte integer at offset 20 is the number of bytes of space at the end of each page to reserve for extensions. This value is usually 0. The value can be odd.

The "usable size" of a database page is the page size specify by the 2-byte integer at offset 16 in the header less the "reserved" space size recorded in the 1-byte integer at offset 20 in the header. The usable size of a page might be an odd number. However, the usable size is not allowed to be less than 480. In other words, if the page size is 512, then the reserved space size cannot exceed 32.

1.2.5 Payload fractions

The maximum and minimum embedded payload fractions and the leaf payload fraction values must be 64, 32, and 32. These values were originally intended to be tunable parameters that could be used to modify the storage format of the b-tree algorithm. However, that functionality is not supported and there are no current plans to add support in the future. Hence, these three bytes are fixed at the values specified.

1.2.6 File change counter

The file change counter is a 4-byte big-endian integer at offset 24 that is incremented whenever the database file is unlocked after having been modified. When two or more processes are reading the same database file, each process can detect database changes from other processes by monitoring the change counter. A process will normally want to flush its database page cache when another process modified the database, since the cache has become stale. The file change counter facilitates this.

In WAL mode, changes to the database are detected using the wal-index and so the change counter is not needed. Hence, the change counter might not be incremented on each transaction in WAL mode.

1.2.7 In-header database size

The 4-byte big-endian integer at offset 28 into the header stores the size of the database file in pages. If this in-header datasize size is not valid (see the next paragraph), then the database size is computed by looking at the actual size of the database file. Older versions of SQLite ignored the in-header database size and used the actual file size exclusively. Newer versions of SQLite use the in-header database size if it is available but fall back to the actual file size if the in-header database size is not valid.

The in-header database size is only considered to be valid if it is non-zero and if the 4-byte change counter at offset 24 exactly matches the 4-byte version-valid-for number at offset 92. The in-header database size is always valid when the database is only modified using recent versions of SQLite (versions 3.7.0 and later). If a legacy version of SQLite writes to the database, it will not know to update the in-header database size and so the in-header database size could be incorrect. But legacy versions of SQLite will also leave the version-valid-for number at offset 92 unchanged so it will not match the change-counter. Hence, invalid in-header database sizes can be detected (and ignored) by observing when the change-counter does not match the version-valid-for number.

1.2.8 Free page list

Unused pages in the database file are stored on a freelist. The 4-byte big-endian integer at offset 32 stores the page number of the first page of the freelist, or zero if the freelist is empty. The 4-byte big-endian integer at offset 36 stores stores the total number of pages on the freelist.

1.2.9 Schema cookie

The schema cookie is a 4-byte big-endian integer at offset 40 that is incremented whenever the database schema changes. A prepared statement is compiled against a specific version of the database schema. When the database schema changes, the statement must be reprepared. When a prepared statement runs, it first checks the schema cookie to ensure the value is the same as when the statement was prepared and if the schema cookie has changed, the statement either automatically reprepares and reruns or it aborts with an SQLITE_SCHEMA error.

1.2.10 Schema format number

The schema format number is a 4-byte big-endian integer at offset 44. The schema format number is similar to the file format read and write version numbers at offsets 18 and 19 except that the schema format number refers to the high-level SQL formatting rather than the low-level b-tree formatting. Four schema format numbers are currently defined:

1. Format 1 is understood by all versions of SQLite back to version 3.0.0.
2. Format 2 adds the ability of rows within the same table to have a varying number of columns, in order to support the ALTER TABLE ... ADD COLUMN functionality. Support for reading and writing format 2 was added in SQLite version 3.1.3 on 2005-02-19.
3. Format 3 adds the ability of extra columns added by ALTER TABLE ... ADD COLUMN to have non-NULL default values. This capability was added in SQLite version 3.1.4 on 2005-03-11.
4. Format 4 causes SQLite to respect the DESC keyword on index declarations. (The DESC keyword is ignored in indexes for formats 1, 2, and 3.) Format 4 also adds two new boolean record type values (serial types 8 and 9). Support for format 4 was added in SQLite 3.3.0 on 2006-01-10.

New database files created by SQLite use format 4 by default. The legacy_file_format pragma can be used to cause SQLite to create new database files using format 1. The format version number can be made to default to 1 instead of 4 by setting SQLITE_DEFAULT_FILE_FORMAT=1 at compile-time.

1.2.11 Suggested cache size

The 4-byte big-endian signed integer at offset 48 is the suggested cache size in pages for the database file. The value is a suggestion only and SQLite is under no obligation to honor it. The absolute value of the integer is used as the suggested size. The suggested cache size can be set using the default_cache_size pragma.

1.2.12 Incremental vacuum settings

The two 4-byte big-endian integers at offsets 52 and 64 are used to manage the auto_vacuum and incremental_vacuum modes. If the integer at offset 52 is zero then pointer-map (ptrmap) pages are omitted from the database file and neither auto_vacuum nor incremental_vacuum are supported. If the integer at offset 52 is non-zero then it is the page number of the largest root page in the database file, the database file will contain ptrmap pages, and the mode must be either auto_vacuum or incremental_vacuum. In this latter case, the integer at offset 64 is true for incremental_vacuum and false for auto_vacuum. If the integer at offset 52 is zero then the integer at offset 64 must also be zero.

1.2.13 Text encoding

The 4-byte big-endian integer at offset 56 determines the encoding used for all text strings stored in the database. A value of 1 means UTF-8. A value of 2 means UTF-16le. A value of 3 means UTF-16be. No other values are allowed. The sqlite3.h header file defines C-preprocessor macros SQLITE_UTF8 as 1, SQLITE_UTF16LE as 2, and SQLITE_UTF16BE as 3, to use in place of the numeric codes for the text encoding.

1.2.14 User version number

The 4-byte big-endian integer at offset 60 is the user version which is set and queried by the user_version pragma. The user version is not used by SQLite.

1.2.15 Application ID

The 4-byte big-endian integer at offset 68 is an "Application ID" that can be set by the PRAGMA application_id command in order to identify the database as belonging to or associated with a particular application. The application ID is intended for database files used as an application file-format. The application ID can be used by utilities such as file(1) to determine the specific file type rather than just reporting "SQLite3 Database". A list of assigned application IDs can be seen by consulting the magic.txt file in the SQLite source repository.

1.2.16 Write library version number and version-valid-for number

The 4-byte big-endian integer at offset 96 stores the SQLITE_VERSION_NUMBER value for the SQLite library that most recently modified the database file. The 4-byte big-endian integer at offset 92 is the value of the change counter when the version number was stored. The integer at offset 92 indicates which transaction the version number is valid for and is sometimes called the "version-valid-for number".

1.2.16 Header space reserved for expansion

All other bytes of the database file header are reserved for future expansion and must be set to zero.

1.3 The Lock-Byte Page

The lock-byte page is the single page of the database file that contains the bytes at offsets between 1073741824 and 1073742335, inclusive. A database file that is less than or equal to 1073741824 bytes in size contains no lock-byte page. A database file larger than 1073741824 contains exactly one lock-byte page.

The lock-byte page is set aside for use by the operating-system specific VFS implementation in implementing the database file locking primitives. SQLite does not use the lock-byte page. The SQLite core will never read or write the lock-byte page, though operating-system specific VFS implementations may choose to read or write bytes on the lock-byte page according to the needs and proclivities of the underlying system. The unix and win32 VFS implementations that come built into SQLite do not write to the lock-byte page, but third-party VFS implementations for other operating systems might.

1.4 The Freelist

A database file might contain one or more pages that are not in active use. Unused pages can come about, for example, when information is deleted from the database. Unused pages are stored on the freelist and are reused when additional pages are required.

The freelist is organized as a linked list of freelist trunk pages with each trunk pages containing page numbers for zero or more freelist leaf pages.

A freelist trunk page consists of an array of 4-byte big-endian integers. The size of the array is as many integers as will fit in the usable space of a page. The minimum usable space is 480 bytes so the array will always be at least 120 entries in length. The first integer on a freelist trunk page is the page number of the next freelist trunk page in the list or zero if this is the last freelist trunk page. The second integer on a freelist trunk page is the number of leaf page pointers to follow. Call the second integer on a freelist trunk page L. If L is greater than zero then integers with array indexes between 2 and L+1 inclusive contain page numbers for freelist leaf pages.

Freelist leaf pages contain no information. SQLite avoids reading or writing freelist leaf pages in order to reduce disk I/O.

A bug in SQLite versions prior to 3.6.0 caused the database to be reported as corrupt if any of the last 6 entries in the freelist trunk page array contained non-zero values. Newer versions of SQLite do not have this problem. However, newer versions of SQLite still avoid using the last six entries in the freelist trunk page array in order that database files created by newer versions of SQLite can be read by older versions of SQLite.

The number of freelist pages is stored as a 4-byte big-endian integer in the database header at an offset of 36 from the beginning of the file. The database header also stores the page number of the first freelist trunk page as a 4-byte big-endian integer at an offset of 32 from the beginning of the file.

1.5 B-tree Pages

The b-tree algorithm provides key/data storage with unique and ordered keys on page-oriented storage devices. For background information on b-trees, see Knuth, The Art Of Computer Programming, Volume 3 "Sorting and Searching", pages 471-479. Two kinds of b-trees are used by SQLite. The algorithm that Knuth calls "B*-Tree" stores all data in the leaves of the tree. SQLite calls this variety of b-tree a "table b-tree". The algorithm that Knuth calls simply "B-Tree" stores both the key and the data together in both leaves and in interior pages. In the SQLite implementation, the original B-Tree algorithm stores keys only, omitting the data entirely, and is called an "index b-tree".

A b-tree page is either an interior page or a leaf page. A leaf page contains keys and in the case of a table b-tree each key has associated data. An interior page contains K keys together with K+1 pointers to child b-tree pages. A "pointer" in an interior b-tree page is just the 31-bit integer page number of the child page.

Define the depth of a leaf b-tree to be 1 and the depth of any interior b-tree to be one more than the maximum depth of any of its children. In a well-formed database, all children of an interior b-tree have the same depth.

In an interior b-tree page, the pointers and keys logically alternate with a pointer on both ends. (The previous sentence is to be understood conceptually - the actual layout of the keys and pointers within the page is more complicated and will be described in the sequel.) All keys within the same page are unique and are logically organized in ascending order from left to right. (Again, this ordering is logical, not physical. The actual location of keys within the page is arbitrary.) For any key X, pointers to the left of a X refer to b-tree pages on which all keys are less than or equal to X. Pointers to the right of X refer to pages where all keys are greater than X.

Within an interior b-tree page, each key and the pointer to its immediate left are combined into a structure called a "cell". The right-most pointer is held separately. A leaf b-tree page has no pointers, but it still uses the cell structure to hold keys for index b-trees or keys and content for table b-trees. Data is also contained in the cell.

Every b-tree page has at most one parent b-tree page. A b-tree page without a parent is called a root page. A root b-tree page together with the closure of its children form a complete b-tree. It is possible (and in fact rather common) to have a complete b-tree that consists of a single page that is both a leaf and the root. Because there are pointers from parents to children, every page of a complete b-tree can be located if only the root page is known. Hence, b-trees are identified by their root page number.

A b-tree page is either a table b-tree page or an index b-tree page. All pages within each complete b-tree are of the same type: either table or index. There is one table b-trees in the database file for each rowid table in the database schema, including system tables such as sqlite_master. There is one index b-tree in the database file for each index in the schema, including implied indexes created by uniqueness constraints. There are no b-trees associated with virtual tables. Specific virtual table implementations might make use of shadow tables for storage, but those shadow tables will have separate entries in the database schema. WITHOUT ROWID tables use index b-trees rather than a table b-trees, so there is one index b-tree in the database file for each WITHOUT ROWID table. The b-tree corresponding to the sqlite_master table is always a table b-tree and always has a root page of 1. The sqlite_master table contains the root page number for every other table and index in the database file.

Each entry in a table b-tree consists of a 64-bit signed integer key and up to 2147483647 bytes of arbitrary data. (The key of a table b-tree corresponds to the rowid of the SQL table that the b-tree implements.) Interior table b-trees hold only keys and pointers to children. All data is contained in the table b-tree leaves.

Each entry in an index b-tree consists of an arbitrary key of up to 2147483647 bytes in length and no data.

Define the "payload" of a cell to be the arbitrary length section of the cell. For an index b-tree, the key is always arbitrary in length and hence the payload is the key. There are no arbitrary length elements in the cells of interior table b-tree pages and so those cells have no payload. Table b-tree leaf pages contain arbitrary length content and so for cells on those pages the payload is the content.

When the size of payload for a cell exceeds a certain threshold (to be defined later) then only the first few bytes of the payload are stored on the b-tree page and the balance is stored in a linked list of content overflow pages.

A b-tree page is divided into regions in the following order:

1. The 100-byte database file header (found on page 1 only)
2. The 8 or 12 byte b-tree page header
3. The cell pointer array
4. Unallocated space
5. The cell content area
6. The reserved region.

The 100-byte database file header is found only on page 1, which is always a table b-tree page. All other b-tree pages in the database file omit this 100-byte header.

The reserved region is an area of unused space at the end of every page (except the locking page) that extensions can use to hold per-page information. The size of the reserved region is determined by the one-byte unsigned integer found at an offset of 20 into the database file header. The size of the reserved region is usually zero.

The b-tree page header is 8 bytes in size for leaf pages and 12 bytes for interior pages. All multibyte values in the page header are big-endian. The b-tree page header is composed of the following fields:

B-tree Page Header Format

Offset	Size	Description
0	1	The one-byte flag at offset 0 indicating the b-tree page type. A value of 2 means the page is an interior index b-tree page. A value of 5 means the page is an interior table b-tree page. A value of 10 means the page is a leaf index b-tree page. A value of 13 means the page is a leaf table b-tree page. Any other value for the b-tree page type is an error.
1	2	The two-byte integer at offset 1 gives the start of the first freeblock on the page, or is zero if there are no freeblocks.
3	2	The two-byte integer at offset 3 gives the number of cells on the page.
5	2	The two-byte integer at offset 5 designates the start of the cell content area. A zero value for this integer is interpreted as 65536.
7	1	The one-byte integer at offset 7 gives the number of fragmented free bytes within the cell content area.
8	4	The four-byte page number at offset 8 is the right-most pointer. This value appears in the header of interior b-tree pages only and is omitted from all other pages.

The cell pointer array of a b-tree page immediately follows the b-tree page header. Let K be the number of cells on the btree. The cell pointer array consists of K 2-byte integer offsets to the cell contents. The cell pointers are arranged in key order with left-most cell (the cell with the smallest key) first and the right-most cell (the cell with the largest key) last.

Cell content is stored in the cell content region of the b-tree page. SQLite strives to place cells as far toward the end of the b-tree page as it can, in order to leave space for future growth of the cell pointer array. The area in between the last cell pointer array entry and the beginning of the first cell is the unallocated region.

If a page contains no cells (which is only possible for a root page of a table that contains no rows) then the offset to the cell content area will equal the page size minus the bytes of reserved space. If the database uses a 65536-byte page size and the reserved space is zero (the usual value for reserved space) then the cell content offset of an empty page wants to be 65536. However, that integer is too large to be stored in a 2-byte unsigned integer, so a value of 0 is used in its place.

A freeblock is a structure used to identify unallocated space within a b-tree page. Freeblocks are organized as a chain. The first 2 bytes of a freeblock are a big-endian integer which is the offset in the b-tree page of the next freeblock in the chain, or zero if the freeblock is the last on the chain. The third and fourth bytes of each freeblock form a big-endian integer which is the size of the freeblock in bytes, including the 4-byte header. Freeblocks are always connected in order of increasing offset. The second field of the b-tree page header is the offset of the first freeblock, or zero if there are no freeblocks on the page. In a well-formed b-tree page, there will always be at least one cell before the first freeblock.

A freeblock requires at least 4 bytes of space. If there is an isolated group of 1, 2, or 3 unused bytes within the cell content area, those bytes comprise a fragment. The total number of bytes in all fragments is stored in the fifth field of the b-tree page header. In a well-formed b-tree page, the total number of bytes in fragments may not exceed 60.

The total amount of free space on a b-tree page consists of the size of the unallocated region plus the total size of all freeblocks plus the number of fragmented free bytes. SQLite may from time to time reorganize a b-tree page so that there are no freeblocks or fragment bytes, all unused bytes are contained in the unallocated space region, and all cells are packed tightly at the end of the page. This is called "defragmenting" the b-tree page.

A variable-length integer or "varint" is a static Huffman encoding of 64-bit twos-complement integers that uses less space for small positive values. A varint is between 1 and 9 bytes in length. The varint consists of either zero or more bytes which have the high-order bit set followed by a single byte with the high-order bit clear, or nine bytes, whichever is shorter. The lower seven bits of each of the first eight bytes and all 8 bits of the ninth byte are used to reconstruct the 64-bit twos-complement integer. Varints are big-endian: bits taken from the earlier byte of the varint are the more significant than bits taken from the later bytes.

The format of a cell depends on which kind of b-tree page the cell appears on. The following table shows the elements of a cell, in order of appearance, for the various b-tree page types.

> Table B-Tree Leaf Cell (header 0x0d):
>
> - A varint which is the total number of bytes of payload, including any overflow
> - A varint which is the integer key, a.k.a. "rowid"
> - The initial portion of the payload that does not spill to overflow pages.
> - A 4-byte big-endian integer page number for the first page of the overflow page list - omitted if all payload fits on the b-tree page.
>
> Table B-Tree Interior Cell (header 0x05):
>
> - A 4-byte big-endian page number which is the left child pointer.
> - A varint which is the integer key
>
> Index B-Tree Leaf Cell (header 0x0a):
>
> - A varint which is the total number of bytes of key payload, including any overflow
> - The initial portion of the payload that does not spill to overflow pages.
> - A 4-byte big-endian integer page number for the first page of the overflow page list - omitted if all payload fits on the b-tree page.
>
> Index B-Tree Interior Cell (header 0x02):
>
> - A 4-byte big-endian page number which is the left child pointer.
> - A varint which is the total number of bytes of key payload, including any overflow
> - The initial portion of the payload that does not spill to overflow pages.
> - A 4-byte big-endian integer page number for the first page of the overflow page list - omitted if all payload fits on the b-tree page.

The information above can be recast into a table format as follows:

B-tree Cell Format

Datatype	Appears in...	Description		
Table Leaf (0x0d)	Table Interior (0x05)	Index Leaf (0x0a)	Index Interior (0x02)	
4-byte integer	✔	✔	Page number of left child	
varint	✔	✔	✔	Number of bytes of payload
varint	✔	✔	Rowid	
byte array	✔	✔	✔	Payload
4-byte integer	✔	✔	✔	Page number of first overflow page

The amount of payload that spills onto overflow pages also depends on the page type. For the following computations, let U be the usable size of a database page, the total page size less the reserved space at the end of each page. And let P be the payload size. In the following, symbol X represents the maximum amount of payload that can be stored directly on the b-tree page without spilling onto an overflow page and symbol M represents the minimum amount of payload that must be stored on the btree page before spilling is allowed.

> Table B-Tree Leaf Cell:
>
> Let X be U-35. If the payload size P is less than or equal to X then the entire payload is stored on the b-tree leaf page. Let M be ((U-12)*32/255)-23 and let K be M+((P-M)%(U-4)). If P is greater than X then the number of bytes stored on the table b-tree leaf page is K if K is less or equal to X or M otherwise. Note that number of bytes stored on the leaf page is never less than M.
>
> Table B-Tree Interior Cell:
>
> Interior pages of table b-trees have no payload and so there is never any payload to spill.
>
> Index B-Tree Leaf Or Interior Cell:
>
> Let X be ((U-12)*64/255)-23). If the payload size P is less than or equal to X then the entire payload is stored on the b-tree page. Let M be ((U-12)*32/255)-23 and let K be M+((P-M)%(U-4)). If P is greater than X then the number of bytes stored on the index b-tree page is K if K is less than or equal to X or M otherwise. Note that number of bytes stored on the index page is never less than M.

Here is an alternative description of the same computation:

- X is U-35 for table btree leaf pages or ((U-12)*64/255)-23 for index pages.
- M is always ((U-12)*32/255)-23.
- Let K be M+((P-M)%(U-4)).
- If P<=x then="" all="" p="" bytes="" of="" payload="" are="" stored="" directly="" on="" the="" btree="" page="" without="" overflow.="" <li="">If P>X and K<=x then="" the="" first="" k="" bytes="" of="" p="" are="" stored="" on="" btree="" page="" and="" remaining="" p-k="" overflow="" pages.="" <li="">If P>X and K>X then the first M bytes of P are stored on the btree page and the remaining P-M bytes are stored on overflow pages.</li=""></li="">

The overflow thresholds are designed to give a minimum fanout of 4 for index b-trees and to make sure enough of the payload is on the b-tree page that the record header can usually be accessed without consulting an overflow page. In hindsight, the designers of the SQLite b-tree logic realize that these thresholds could have been made much simpler. However, the computations cannot be changed without resulting in an incompatible file format. And the current computations work well, even if they are a little complex.

1.6 Cell Payload Overflow Pages

When the payload of a b-tree cell is too large for the b-tree page, the surplus is spilled onto overflow pages. Overflow pages form a linked list. The first four bytes of each overflow page are a big-endian integer which is the page number of the next page in the chain, or zero for the final page in the chain. The fifth byte through the last usable byte are used to hold overflow content.

1.7 Pointer Map or Ptrmap Pages

Pointer map or ptrmap pages are extra pages inserted into the database to make the operation of auto_vacuum and incremental_vacuum modes more efficient. Other page types in the database typically have pointers from parent to child. For example, an interior b-tree page contains pointers to its child b-tree pages and an overflow chain has a pointer from earlier to later links in the chain. A ptrmap page contains linkage information going in the opposite direction, from child to parent.

Ptrmap pages must exist in any database file which has a non-zero largest root b-tree page value at offset 52 in the database header. If the largest root b-tree page value is zero, then the database must not contain ptrmap pages.

In a database with ptrmap pages, the first ptrmap page is page 2. A ptrmap page consists of an array of 5-byte entries. Let J be the number of 5-byte entries that will fit in the usable space of a page. (In other words, J=U/5.) The first ptrmap page will contain back pointer information for pages 3 through J+2, inclusive. The second pointer map page will be on page J+3 and that ptrmap page will provide back pointer information for pages J+4 through 2*J+3 inclusive. And so forth for the entire database file.

In a database that uses ptrmap pages, all pages at locations identified by the computation in the previous paragraph must be ptrmap page and no other page may be a ptrmap page. Except, if the byte-lock page happens to fall on the same page number as a ptrmap page, then the ptrmap is moved to the following page for that one case.

Each 5-byte entry on a ptrmap page provides back-link information about one of the pages that immediately follow the pointer map. If page B is a ptrmap page then back-link information about page B+1 is provided by the first entry on the pointer map. Information about page B+2 is provided by the second entry. And so forth.

Each 5-byte ptrmap entry consists of one byte of "page type" information followed by a 4-byte big-endian page number. Five page types are recognized:

1. A b-tree root page. The page number should be zero.
2. A freelist page. The page number should be zero.
3. The first page of a cell payload overflow chain. The page number is the b-tree page that contains the cell whose content has overflowed.
4. A page in an overflow chain other than the first page. The page number is the prior page of the overflow chain.
5. A non-root b-tree page. The page number is the parent b-tree page.

In any database file that contains ptrmap pages, all b-tree root pages must come before any non-root b-tree page, cell payload overflow page, or freelist page. This restriction ensures that a root page will never be moved during an auto-vacuum or incremental-vacuum. The auto-vacuum logic does not know how to update the root_page field of the sqlite_master table and so it is necessary to prevent root pages from being moved during an auto-vacuum in order to preserve the integrity of the sqlite_master table. Root pages are moved to the beginning of the database file by the CREATE TABLE, CREATE INDEX, DROP TABLE, and DROP INDEX operations.

2.0 Schema Layer

The foregoing text describes low-level aspects of the SQLite file format. The b-tree mechanism provides a powerful and efficient means of accessing a large data set. This section will describe how the low-level b-tree layer is used to implement higher-level SQL capabilities.

2.1 Record Format

The data for a table b-tree leaf page and the key of an index b-tree page was characterized above as an arbitrary sequence of bytes. The prior discussion mentioned one key being less than another, but did not define what "less than" meant. The current section will address these omissions.

Payload, either table b-tree data or index b-tree keys, is always in the "record format". The record format defines a sequence of values corresponding to columns in a table or index. The record format specifies the number of columns, the datatype of each column, and the content of each column.

The record format makes extensive use of the variable-length integer or varint representation of 64-bit signed integers defined above.

A record contains a header and a body, in that order. The header begins with a single varint which determines the total number of bytes in the header. The varint value is the size of the header in bytes including the size varint itself. Following the size varint are one or more additional varints, one per column. These additional varints are called "serial type" numbers and determine the datatype of each column, according to the following chart:

Serial Type Codes Of The Record Format

Serial Type	Content Size	Meaning
0	0	Value is a NULL.
1	1	Value is an 8-bit twos-complement integer.
2	2	Value is a big-endian 16-bit twos-complement integer.
3	3	Value is a big-endian 24-bit twos-complement integer.
4	4	Value is a big-endian 32-bit twos-complement integer.
5	6	Value is a big-endian 48-bit twos-complement integer.
6	8	Value is a big-endian 64-bit twos-complement integer.
7	8	Value is a big-endian IEEE 754-2008 64-bit floating point number.
8	0	Value is the integer 0. (Only available for schema format 4 and higher.)
9	0	Value is the integer 1. (Only available for schema format 4 and higher.)
10,11	*Not used. Reserved for expansion.*	
N≥12 and even	(N-12)/2	Value is a BLOB that is (N-12)/2 bytes in length.
N≥13 and odd	(N-13)/2	Value is a string in the text encoding and (N-13)/2 bytes in length. The nul terminator is not stored.

Note that because of the way varints are defined, the header size varint and serial type varints will usually consist of a single byte. The serial type varints for large strings and BLOBs might extend to two or three byte varints, but that is the exception rather than the rule. The varint format is very efficient at coding the record header.

The values for each column in the record immediately follow the header. Note that for serial types 0, 8, 9, 12, and 13, the value is zero bytes in length. If all columns are of these types then the body section of the record is empty.

2.2 Record Sort Order

The order of keys in an index b-tree is determined by the sort order of the records that the keys represent. Record comparison progresses column by column. Columns of a record are examined from left to right. The first pair of columns that are not equal determines the relative order of the two records. The sort order of individual columns is as follows:

1. NULL values (serial type 0) sort first.
2. Numeric values (serial types 1 through 9) sort after NULLs and in numeric order.
3. Text values (odd serial types 13 and larger) sort after numeric values in the order determined by the columns collating function.
4. BLOB values (even serial types 12 and larger) sort last and in the order determined by memcmp().

A collating function for each column is necessary in order to compute the order of text fields. SQLite defines three built-in collating functions:

BINARY	The built-in BINARY collation compares strings byte by byte using the memcmp() function from the standard C library.
NOCASE	The NOCASE collation is like BINARY except that uppercase ASCII characters ('A' through 'Z') are folded into their lowercase equivalents prior to running the comparison. Note that only ASCII characters are case-folded. NOCASE does not implement a general purpose unicode caseless comparison.
RTRIM	RTRIM is like BINARY except that extra spaces at the end of either string do not change the result. In other words, strings will compare equal to one another as long as they differ only in the number of spaces at the end.

Additional application-specific collating functions can be added to SQLite using the sqlite3_create_collation() interface.

The default collating function for all strings is BINARY. Alternative collating functions for table columns can be specified in the CREATE TABLE statement using the COLLATE clause on the column definition. When a column is indexed, the same collating function specified in the CREATE TABLE statement is used for the column in the index, by default, though this can be overridden using a COLLATE clause in the CREATE INDEX statement.

2.3 Representation Of SQL Tables

Each ordinary SQL table in the database schema is represented on-disk by a table b-tree. Each entry in the table b-tree corresponds to a row of the SQL table. The rowid of the SQL table is the 64-bit signed integer key for each entry in the table b-tree.

The content of each SQL table row is stored in the database file by first combining the values in the various columns into a byte array in the record format, then storing that byte array as the payload in an entry in the table b-tree. The order of values in the record is the same as the order of columns in the SQL table definition. When an SQL table includes an INTEGER PRIMARY KEY column (which aliases the rowid) then that column appears in the record as a NULL value. SQLite will always use the table b-tree key rather than the NULL value when referencing the INTEGER PRIMARY KEY column.

If the affinity of a column is REAL and that column contains a value that can be converted to an integer without loss of information (if the value contains no fractional part and is not too large to be represented as an integer) then the column may be stored in the record as an integer. SQLite will convert the value back to floating point when extracting it from the record.

2.4 Representation of WITHOUT ROWID Tables

If an SQL table is created using the "WITHOUT ROWID" clause at the end of its CREATE TABLE statement, then that table is a WITHOUT ROWID table and uses a different on-disk representation. A WITHOUT ROWID table uses an index b-tree rather than a table b-tree for storage. The key for each entry in the WITHOUT ROWID b-tree is a record composed of the columns of the PRIMARY KEY followed by all remaining columns of the table. The primary key columns appear in the order they they were declared in the PRIMARY KEY clause and the remaining columns appear in the order they occur in the CREATE TABLE statement.

Hence, the content encoding for a WITHOUT ROWID table is the same as the content encoding for an ordinary rowid table, except that the order of the columns is rearranged so that PRIMARY KEY columns come first, and the content is used as the key in an index b-tree rather than as the data in a table b-tree. The special encoding rules for columns with REAL affinity apply to WITHOUT ROWID tables the same as they do with rowid tables.

2.5 Representation Of SQL Indices

Each SQL index, whether explicitly declared via a CREATE INDEX statement or implied by a UNIQUE or PRIMARY KEY constraint, corresponds to an index b-tree in the database file. Each entry in the index b-tree corresponds to a single row in the associated SQL table. The key to an index b-tree is a record composed of the columns that are being indexed followed by the key of the corresponding table row. For ordinary tables, the row key is the rowid, and for WITHOUT ROWID tables the row key is the PRIMARY KEY. Because every row in the table has a unique row key, all keys in an index are unique.

In a normal index, there is a one-to-one mapping between rows in a table and entries in each index associated with that table. However, in a partial index, the index b-tree only contains entries corresponding to table rows for which the WHERE clause expression on the CREATE INDEX statement is true. Corresponding rows in the index and table b-trees share the same rowid or primary key values and contain the same value for all indexed columns.

2.5.1 Suppression of redundant columns in WITHOUT ROWID secondary indexed

In an index on a WITHOUT ROWID table, if one or more of the columns of the table PRIMARY KEY are also columns of the index, then the indexed column is not repeated in the table-key suffix on the end of the index record. As an example, consider the following SQL:

```
CREATE TABLE ex25(a,b,c,d,e,PRIMARY KEY(d,c,a)) WITHOUT rowid;
CREATE INDEX ex25ce ON ex25(c,e);
CREATE INDEX ex25acde ON ex25(a,c,d,e);
```

Each row in the ex25ce index is a record with these columns: c, e, d, a. The first two columns are the columns being indexed, c and e. The remaining columns are the primary key of the corresponding table row. Normally, the primary key would be columns d, c, and a, but because column c already appears earlier in the index, it is omitted from the key suffix.

In the extreme case where the columns being indexed cover all columns of the PRIMARY KEY, the index will consist of only the columns being indexed. The ex25acde example above demonstrates this. Each entry in the ex25acde index consists of only the columns a, c, d, and e, in that order.

The suppression of redundant columns in the key suffix of an index entry only occurs in WITHOUT ROWID tables. In an ordinary rowid table, the index entry always ends with the rowid even if the INTEGER PRIMARY KEY column is one of the columns being indexed.

2.6 Storage Of The SQL Database Schema

Page 1 of a database file is the root page of a table b-tree that holds a special table named "sqlite_master" (or "sqlite_temp_master" in the case of a TEMP database) which stores the complete database schema. The structure of the sqlite_master table is as if it had been created using the following SQL:

```
CREATE TABLE sqlite_master(
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
);
```

The sqlite_master table contains one row for each table, index, view, and trigger (collectively "objects") in the database schema, except there is no entry for the sqlite_master table itself. The sqlite_master table contains entries for internal schema objects in addition to application- and programmer-defined objects.

The sqlite_master.type column will be one of the following text strings: 'table', 'index', 'view', or 'trigger' according to the type of object defined. The 'table' string is used for both ordinary and virtual tables.

The sqlite_master.name column will hold the name of the object. UNIQUE and PRIMARY KEY constraints on tables cause SQLite to create internal indexes with names of the form "sqlite_autoindex_TABLE_N" where TABLE is replaced by the name of the table that contains the constraint and N is an integer beginning with 1 and increasing by one with each constraint seen in the table definition. In a WITHOUT ROWID table, there is no sqlite_master entry for the PRIMARY KEY, but the "sqlite_autoindex_TABLE_N" name is set aside for the PRIMARY KEY as if the sqlite_master entry did exist. This will affect the numbering of subsequent UNIQUE constraints. The "sqlite_autoindex_TABLE_N" name is never allocated for an INTEGER PRIMARY KEY, either in rowid tables or WITHOUT ROWID tables.

The sqlite_master.tbl_name column holds the name of a table or view that the object is associated with. For a table or view, the tbl_name column is a copy of the name column. For an index, the tbl_name is the name of the table that is indexed. For a trigger, the tbl_name column stores the name of the table or view that causes the trigger to fire.

The sqlite_master.rootpage column stores the page number of the root b-tree page for tables and indexes. For rows that define views, triggers, and virtual tables, the rootpage column is 0 or NULL.

The sqlite_master.sql column stores SQL text that describes the object. This SQL text is a CREATE TABLE, CREATE VIRTUAL TABLE, CREATE INDEX, CREATE VIEW, or CREATE TRIGGER statement that if evaluated against the database file when it is the main database of a database connection would recreate the object. The text is usually a copy of the original statement used to create the object but with normalizations applied so that the text conforms to the following rules:

- The CREATE, TABLE, VIEW, TRIGGER, and INDEX keywords at the beginning of the statement are converted to all upper case letters.
- The TEMP or TEMPORARY keyword is removed if it occurs after the initial CREATE keyword.
- Any database name qualifier that occurs prior to the name of the object being created is removed.
- Leading spaces are removed.
- All spaces following the first two keywords are converted into a single space.

The text in the sqlite_master.sql column is a copy of the original CREATE statement text that created the object, except normalized as described above and as modified by subsequent ALTER TABLE statements. The sqlite_master.sql is NULL for the internal indexes that are automatically created by UNIQUE or PRIMARY KEY constraints.

2.6.1 Internal Schema Objects

In addition to the tables, indexes, views, and triggers created by the application and/or the developer using CREATE statements SQL, the sqlite*master table may contain zero or more entries for _internal schema objects* that are created by SQLite for its own internal use. The names of internal schema objects always begin with "sqlite*" and any table, index, view, or trigger whose name begins with "sqlite*" is an internal schema object. SQLite prohibits applications from creating objects whose names begin with "sqlite_".

Internal schema objects used by SQLite may include the following:

- Indices with names of the form "sqlite_autoindex_TABLE_N" that are used to implement UNIQUE and PRIMARY KEY constraints on ordinary tables.
- A table with the name "sqlite_sequence" that is used to keep track of the maximum historical INTEGER PRIMARY KEY for a table using AUTOINCREMENT.
- Tables with names of the form "sqlite_statN" where N is an integer. Such tables store database statistics gathered by the ANALYZE command and used by the query planner to help determine the best algorithm to use for each query.

New internal schema objects names, always beginning with "sqlite_", may be added to the SQLite file format in future releases.

2.6.2 The sqlite_sequence table

The sqlite_sequence table is an internal table used to help implement AUTOINCREMENT. The sqlite_sequence table is created automatically whenever any ordinary table with an AUTOINCREMENT integer primary key is created. Once created, the sqlite_sequence table exists in the sqlite_master table forever; it cannot be dropped. The schema for the sqlite_sequence table is:

```
CREATE TABLE sqlite_sequence(name,seq);
```

There is a single row in the sqlite_sequence table for each ordinary table that uses AUTOINCREMENT. The name of the table (as it appears in sqlite_master.name) is in the sqlite_sequence.main field and the largest INTEGER PRIMARY KEY ever used by that table is in the sqlite_sequence.seq field. New automatically generated integer primary keys for AUTOINCREMENT tables are guaranteed to be larger than the sqlite_sequence.seq field for that table. If the sqlite_sequence.seq field of an AUTOINCREMENT table is already at the largest integer value (9223372036854775807) then attempts to add new rows to that table with an automatically generated integer primary will fail with an SQLITE_FULL error. The sqlite_sequence.seq field is automatically updated if required when new entries are added to an AUTOINCREMENT table. The sqlite_sequence row for an AUTOINCREMENT table is automatically deleted when the table is dropped. If the sqlite_sequence row for an AUTOINCREMENT table does not exist when the AUTOINCREMENT table is updated, then a new sqlite_sequence row is created. If the sqlite_sequence.seq value for an AUTOINCREMENT table is manually set to something other than an integer and there is a subsequent attempt to insert the or update the AUTOINCREMENT table, then the behavior is undefined.

Application code is allowed to modify the sqlite_sequence table, to add new rows, to delete rows, or to modify existing rows. However, application code cannot create the sqlite_sequence table if it does not already exist. Application code can delete all entries from the sqlite_sequence table, but application code cannot drop the sqlite_sequence table.

2.6.3 The sqlite_stat1 table

The sqlite_stat1 is an internal table created by the ANALYZE command and used to hold supplemental information about tables and indexes that the query planner can use to help it find better ways of performing queries. Applications can update, delete from, insert into or drop the sqlite_stat1 table, but may not create or alter the sqlite_stat1 table. The schema of the sqlite_stat1 table is as follows:

```
CREATE TABLE sqlite_stat1(tbl,idx,stat);
```

There is normally one row per index, with the index identified by the name in the sqlite_stat1.idx column. The sqlite_stat1.tbl column is the name of the table to which the index belongs. In each such row, the sqlite_stat.stat column will be a string consisting of a list of integers followed by zero or more arguments. The first integer in this list is the approximate number of rows in the index. (The number of rows in the index is the same as the number of rows in the table, except for partial indexes.) The second integer is the approximate number of rows in the index that have the same value in the first column of the index. The third integer is the number number of rows in the index that have the same value for the first two columns. The N-th integer (for N>1) is the estimated average number of rows in the index which have the same value for the first N-1 columns. For a K-column index, there will be K+1 integers in the stat column. If the index is unique, then the last integer will be 1.

The list of integers in the stat column can optionally be followed by arguments, each of which is a sequence of non-space characters. All arguments are preceded by a single space. Unrecognized arguments are silently ignored.

If the "unordered" argument is present, then the query planner assumes that the index is unordered and will not use the index for a range query or for sorting.

The "sz=NNN" argument (where NNN represents a sequence of 1 or more digits) means that the average row size over all records of the table or index is NNN bytes per row. The SQLite query planner might use the estimated row size information provided by the "sz=NNN" token to help it choose smaller tables and indexes that require less disk I/O.

The presence of the "noskipscan" token on the sqlite_stat1.stat field of an index prevents that index from being used with the skip-scan optimization.

New text tokens may be added to the end of the stat column in future enhancements to SQLite. For compatibility, unrecognized tokens at the end of the stat column are silently ignored.

If the sqlite_stat1.idx column is NULL, then the sqlite_stat1.stat column contains a single integer which is the approximate number of rows in the table identified by sqlite_stat1.tbl.

2.6.4 The sqlite_stat2 table

The sqlite_stat2 is only created and is only used if SQLite is compiled with SQLITE_ENABLE_STAT2 and if the SQLite version number is between 3.6.18 and 3.7.8. The sqlite_stat2 table is neither read nor written by any version of SQLite before 3.6.18 nor after 3.7.8. The sqlite_stat2 table contains additional information about the distribution of keys within an index. The schema of the sqlite_stat2 table is as follows:

```
CREATE TABLE sqlite_stat2(tbl,idx,sampleno,sample);
```

The sqlite_stat2.idx column and the sqlite_stat2.tbl column in each row of the sqlite_stat2 table identify an index described by that row. There are usually 10 rows in the sqlite_stat2 table for each index.

The sqlite_stat2 entries for an index that have sqlite_stat2.sampleno between 0 and 9 inclusive are samples of the left-most key value in the index taken at evenly spaced points along the index. Let C be the number of rows in the index. Then the sampled rows are given by

> rownumber = (i*C2* + C)/20

The variable i in the previous expression varies between 0 and 9. Conceptually, the index space is divided into 10 uniform buckets and the samples are the middle row from each bucket.

The format for sqlite_stat2 is recorded here for legacy reference. Recent versions of SQLite no longer support sqlite_stat2 and the sqlite_stat2 table, it is exists, is simply ignored.

2.6.5 The sqlite_stat3 table

The sqlite_stat3 is only used if SQLite is compiled with SQLITE_ENABLE_STAT3 or SQLITE_ENABLE_STAT4 and if the SQLite version number is 3.7.9 or greater. The sqlite_stat3 table is neither read nor written by any version of SQLite before 3.7.9. If the SQLITE_ENABLE_STAT4 compile-time option is used and the SQLite version number is 3.8.1 or greater, then sqlite_stat3 might be read but not written. The sqlite_stat3 table contains additional information about the distribution of keys within an index, information that the query planner can use to devise better and faster query algorithms. The schema of the sqlite_stat3 table is as follows:

```
CREATE TABLE sqlite_stat3(tbl,idx,nEq,nLt,nDLt,sample);
```

There are usually multiple entries in the sqlite_stat3 table for each index. The sqlite_stat3.sample column holds the value of the left-most field of an index identified by sqlite_stat3.idx and sqlite_stat3.tbl. The sqlite_stat3.nEq column holds the approximate number of entries in the index whose left-most column exactly matches the sample. The sqlite_stat3.nLt holds the approximate number of entries in the index whose left-most column is less than the sample. The sqlite_stat3.nDLt column holds the approximate number of distinct left-most entries in the index that are less than the sample.

There can be an arbitrary number of sqlite_stat3 entries per index. The ANALYZE command will typically generate sqlite_stat3 tables that contain between 10 and 40 samples that are distributed across the key space and with large nEq values.

In a well-formed sqlite_stat3 table, the samples for any single index must appear in the same order that they occur in the index. In other words, if the entry with left-most column S1 is earlier in the index b-tree than the entry with left-most column S2, then in the sqlite_stat3 table, sample S1 must have a smaller rowid than sample S2.

2.6.6 The sqlite_stat4 table

The sqlite_stat4 is only created and is only used if SQLite is compiled with SQLITE_ENABLE_STAT4 and if the SQLite version number is 3.8.1 or greater. The sqlite_stat4 table is neither read nor written by any version of SQLite before 3.8.1. The sqlite_stat4 table contains additional information about the distribution of keys within an index or the distribution of keys in the primary key of a WITHOUT ROWID table. The query planner can sometimes use the additional information in the sqlite_stat4 table to devise better and faster query algorithms. The schema of the sqlite_stat4 table is as follows:

```
CREATE TABLE sqlite_stat4(tbl,idx,nEq,nLt,nDLt,sample);
```

There are typically between 10 to 40 entries in the sqlite_stat4 table for each index for which statistics are available, however these limits are not hard bounds. The meanings of the columns in the sqlite_stat4 table are as follows:

tbl:	The sqlite_stat4.tbl column holds name of the table that owns the index that the row describes
idx:	The sqlite_stat4.idx column holds name of the index that the row describes, or in the case of an sqlite_stat4 entry for a WITHOUT ROWID table, the name of the table itself.
sample:	The sqlite_stat4.sample column holds a BLOB in the record format that encodes the indexed columns followed by the rowid for a rowid table or by the columns of the primary key for a WITHOUT ROWID table. The sqlite_stat4.sample BLOB for the WITHOUT ROWID table itself contains just the columns of the primary key. Let the number of columns encoded by the sqlite_stat4.sample blob be N. For indexes on an ordinary rowid table, N will be one more than the number of columns indexed. For indexes on WITHOUT ROWID tables, N will be the number of columns indexed plus the number of columns in the primary key. For a WITHOUT ROWID table, N will be the number of columns in the primary key.
nEq:	The sqlite_stat4.nEq column holds a list of N integers where the K-th integer is the approximate number of entries in the index whose left-most K columns exactly match the K left-most columns of the sample.
nLt:	The sqlite_stat4.nLt column holds a list of N integers where the K-th integer is the approximate number of entries in the index whose K left-most columns are collectively less than the K left-most columns of the sample.
nDLt:	The sqlite_stat4.nDLt column holds a list of N integers where the K-th integer is the approximate number of entries in the index that are distinct in the first K columns and where the left-most K columns are collectively less than the left-most K columns of the sample.

The sqlite_stat4 is a generalization of the sqlite_stat3 table. The sqlite_stat3 table provides information about the left-most column of an index whereas the sqlite_stat4 table provides information about all columns of the index.

There can be an arbitrary number of sqlite_stat4 entries per index. The ANALYZE command will typically generate sqlite_stat4 tables that contain between 10 and 40 samples that are distributed across the key space and with large nEq values.

In a well-formed sqlite_stat4 table, the samples for any single index must appear in the same order that they occur in the index. In other words, if entry S1 is earlier in the index b-tree than entry S2, then in the sqlite_stat4 table, sample S1 must have a smaller rowid than sample S2.

3.0 The Rollback Journal

The rollback journal is a file associated with each SQLite database file that hold information used to restore the database file to its initial state during the course of a transaction. The rollback journal file is always located in the same directory as the database file and has the same name as the database file but with the string "`-journal`" appended. There can only be a single rollback journal associated with a give database and hence there can only be one write transaction open against a single database at one time.

If a transaction is aborted due to an application crash, an operating system crash, or a hardware power failure or crash, then the database may be left in an inconsistent state. The next time SQLite attempts to open the database file, the presence of the rollback journal file will be detected and the journal will be automatically played back to restore the database to its state at the start of the incomplete transaction.

A rollback journal is only considered to be valid if it exists and contains a valid header. Hence a transaction can be committed in one of three ways:

1. The rollback journal file can be deleted,
2. The rollback journal file can be truncated to zero length, or
3. The header of the rollback journal can be overwritten with invalid header text (for example, all zeros).

These three ways of committing a transaction correspond to the DELETE, TRUNCATE, and PERSIST settings, respectively, of the journal_mode pragma.

A valid rollback journal begins with a header in the following format:

Rollback Journal Header Format

Offset	Size	Description
0	8	Header string: 0xd9, 0xd5, 0x05, 0xf9, 0x20, 0xa1, 0x63, 0xd7
8	4	The "Page Count" - The number of pages in the next segment of the journal, or -1 to mean all content to the end of the file
12	4	A random nonce for the checksum
16	4	Initial size of the database in pages
20	4	Size of a disk sector assumed by the process that wrote this journal.
24	4	Size of pages in this journal.

A rollback journal header is padded with zeros out to the size of a single sector (as defined by the sector size integer at offset 20). The header is in a sector by itself so that if a power loss occurs while writing the sector, information that follows the header will be (hopefully) undamaged.

After the header and zero padding are zero or more page records. Each page record stores a copy of the content of a page from the database file before it was changed. The same page may not appear more than once within a single rollback journal. To rollback an incomplete transaction, a process has merely to read the rollback journal from beginning to end and write pages found in the journal back into the database file at the appropriate location.

Let the database page size (the value of the integer at offset 24 in the journal header) be N. Then the format of a page record is as follows:

Rollback Journal Page Record Format

Offset	Size	Description
0	4	The page number in the database file
4	N	Original content of the page prior to the start of the transaction
N+4	4	Checksum

The checksum is an unsigned 32-bit integer computed as follows:

1. Initialize the checksum to the checksum nonce value found in the journal header at offset 12.
2. Initialize index X to be N-200 (where N is the size of a database page in bytes.
3. Interpret the four bytes at offset X into the page as a 4-byte big-endian unsigned integer. Add the value of that integer to the checksum.
4. Subtrace 200 from X.
5. If X is greater than or equal to zero, go back to step 3.

The checksum value is used to guard against incomplete writes of a journal page record following a power failure. A different random nonce is used each time a transaction is started in order to minimize the risk that unwritten sectors might by chance contain data from the same page that was a part of prior journals. By changing the nonce for each transaction, stale data on disk will still generate an incorrect checksum and be detected with high probability. The checksum only uses a sparse sample of 32-bit words from the data record for performance reasons - design studies during the planning phases of SQLite 3.0.0 showed a significant performance hit in checksumming the entire page.

Let the page count value at offset 8 in the journal header be M. If M is greater than zero then after M page records the journal file may be zero padded out to the next multiple of the sector size and another journal header may be inserted. All journal headers within the same journal must contain the same database page size and sector size.

If M is -1 in the initial journal header, then the number of page records that follow is computed by computing how many page records will fit in the available space of the remainder of the journal file.

4.0 The Write-Ahead Log

Beginning with version 3.7.0, SQLite supports a new transaction control mechanism called "write-ahead log" or "WAL". When a database is in WAL mode, all connections to that database must use the WAL. A particular database will use either a rollback journal or a WAL, but not both at the same time. The WAL is always located in the same directory as the database file and has the same name as the database file but with the string " `-wal` " appended.

4.1 WAL File Format

A WAL file consists of a header followed by zero or more "frames". Each frame records the revised content of a single page from the database file. All changes to the database are recorded by writing frames into the WAL. Transactions commit when a frame is written that contains a commit marker. A single WAL can and usually does record multiple transactions. Periodically, the content of the WAL is transferred back into the database file in an operation called a "checkpoint".

A single WAL file can be reused multiple times. In other words, the WAL can fill up with frames and then be checkpointed and then new frames can overwrite the old ones. A WAL always grows from beginning toward the end. Checksums and counters attached to each frame are used to determine which frames within the WAL are valid and which are leftovers from prior checkpoints.

The WAL header is 32 bytes in size and consists of the following eight big-endian 32-bit unsigned integer values:

WAL Header Format

Offset	Size	Description
0	4	Magic number. 0x377f0682 or 0x377f0683
4	4	File format version. Currently 3007000.
8	4	Database page size. Example: 1024
12	4	Checkpoint sequence number
16	4	Salt-1: random integer incremented with each checkpoint
20	4	Salt-2: a different random number for each checkpoint
24	4	Checksum-1: First part of a checksum on the first 24 bytes of header
28	4	Checksum-2: Second part of the checksum on the first 24 bytes of header

Immediately following the wal-header are zero or more frames. Each frame consists of a 24-byte frame-header followed by a *page-size* bytes of page data. The frame-header is six big-endian 32-bit unsigned integer values, as follows:

WAL Frame Header Format

Offset	Size	Description
0	4	Page number
4	4	For commit records, the size of the database file in pages after the commit. For all other records, zero.
8	4	Salt-1 copied from the WAL header
12	4	Salt-2 copied from the WAL header
16	4	Checksum-1: Cumulative checksum up through and including this page
20	4	Checksum-2: Second half of the cumulative checksum.

A frame is considered valid if and only if the following conditions are true:

1. The salt-1 and salt-2 values in the frame-header match salt values in the wal-header
2. The checksum values in the final 8 bytes of the frame-header exactly match the checksum computed consecutively on the first 24 bytes of the WAL header and the first 8 bytes and the content of all frames up to and including the current frame.

4.2 Checksum Algorithm

The checksum is computed by interpreting the input as an even number of unsigned 32-bit integers: x(0) through x(N). The 32-bit integers are big-endian if the magic number in the first 4 bytes of the WAL header is 0x377f0683 and the integers are little-endian if the magic number is 0x377f0682. The checksum values are always stored in the frame header in a big-endian format regardless of which byte order is used to compute the checksum.

The checksum algorithm only works for content which is a multiple of 8 bytes in length. In other words, if the inputs are x(0) through x(N) then N must be odd. The checksum algorithm is as follows:

```
s0 = s1 = 0
for i from 0 to n-1 step 2:
   s0 += x(i) + s1;
   s1 += x(i+1) + s0;
endfor
# result in s0 and s1
```

The outputs s0 and s1 are both weighted checksums using Fibonacci weights in reverse order. (The largest Fibonacci weight occurs on the first element of the sequence being summed.) The s1 value spans all 32-bit integer terms of the sequence whereas s0 omits the final term.

4.3 Checkpoint Algorithm

On a checkpoint, the WAL is first flushed to persistent storage using the xSync method of the VFS. Then valid content of the WAL is transferred into the database file. Finally, the database is flushed to persistent storage using another xSync method call. The xSync operations serve as write barriers - all writes launched before the xSync must complete before any write that launches after the xSync begins.

After a checkpoint, new write transactions overwrite the WAL file from the beginning. At the start of the first new write transaction, the WAL header salt-1 value is incremented and the salt-2 value is randomized. These changes to the salts invalidate old frames in the WAL that have already been checkpointed but not yet overwritten, and prevent them from being checkpointed again.

4.4 Reader Algorithm

To read a page from the database (call it page number P), a reader first checks the WAL to see if it contains page P. If so, then the last valid instance of page P that is followed by a commit frame or is a commit frame itself becomes the value read. If the WAL contains no copies of page P that are valid and which are a commit frame or are followed by a commit frame, then page P is read from the database file.

To start a read transaction, the reader records the index of the last valid frame in the WAL. The reader uses this recorded "mxFrame" value for all subsequent read operations. New transactions can be appended to the WAL, but as long as the reader uses its original mxFrame value and ignores subsequently appended content, the reader will see a consistent snapshot of the database from a single point in time. This technique allows multiple concurrent readers to view different versions of the database content simultaneously.

The reader algorithm in the previous paragraphs works correctly, but because frames for page P can appear anywhere within the WAL, the reader has to scan the entire WAL looking for page P frames. If the WAL is large (multiple megabytes is typical) that scan can be slow, and read performance suffers. To overcome this problem, a separate data structure called the wal-index is maintained to expedite the search for frames of a particular page.

4.5 WAL-Index Format

Conceptually, the wal-index is shared memory, though the current VFS implementations use a mmapped file for the wal-index. The mmapped file is in the same directory as the database and has the same name as the database with a "`-shm`" suffix appended. Because the wal-index is shared memory, SQLite does not support journal_mode=WAL on a network filesystem when clients are on different machines. All users of the database must be able to share the same memory.

The purpose of the wal-index is to answer this question quickly:

> *Given a page number P and a maximum WAL frame index M, return the largest WAL frame index for page P that does not exceed M, or return NULL if there are no frames for page P that do not exceed M.*

The *M* value in the previous paragraph is the "mxFrame" value defined in section 4.4 that is read at the start of a transaction and which defines the maximum frame from the WAL that the reader will use.

The wal-index is transient. After a crash, the wal-index is reconstructed from the original WAL file. The VFS is required to either truncate or zero the header of the wal-index when the last connection to it closes. Because the wal-index is transient, it can use an architecture-specific format; it does not have to be cross-platform. Hence, unlike the database and WAL file formats which store all values as big endian, the wal-index stores multi-byte values in the native byte order of the host computer.

This document is concerned with the persistent state of the database file, and since the wal-index is a transient structure, no further information about the format of the wal-index will be provided here. Complete details on the format of the wal-index are contained within comments in SQLite source code.

1.0 Compilation Options For SQLite

For most purposes, SQLite can be built just fine using the default compilation options. However, if required, the compile-time options documented below can be used to omit SQLite features (resulting in a smaller compiled library size) or to change the default values of some parameters.

Every effort has been made to ensure that the various combinations of compilation options work harmoniously and produce a working library. Nevertheless, it is strongly recommended that the SQLite test-suite be executed to check for errors before using an SQLite library built with non-standard compilation options.

1.1 Platform Configuration

_HAVE_SQLITE_CONFIG_H

> If the *HAVE_SQLITE_CONFIG_H macro is defined then the SQLite source code will attempt to #include a file named "config.h". The "config.h" file usually contains other configuration options, especially "HAVE__INTERFACE*" type options generated by autoconf scripts.

HAVE_FDATASYNC

> If the HAVE_FDATASYNC compile-time option is true, then the default VFS for unix systems will attempt to use fdatasync() instead of fsync() where appropriate. If this flag is missing or false, then fsync() is always used.

HAVE_GMTIME_R

> If the HAVE_GMTIME_R option is true and if SQLITE_OMIT_DATETIME_FUNCS is true, then the CURRENT_TIME, CURRENT_DATE, and CURRENT_TIMESTAMP keywords will use the threadsafe "gmtime_r()" interface rather than "gmtime()". In the usual case where SQLITE_OMIT_DATETIME_FUNCS is not defined or is false, then the built-in date and time functions are used to implement the CURRENT_TIME, CURRENT_DATE, and CURRENT_TIMESTAMP keywords and neither gmtime_r() nor gmtime() is ever called.

HAVE_ISNAN

> If the HAVE_ISNAN option is true, then SQLite invokes the system library isnan() function to determine if a double-precision floating point value is a NaN. If HAVE_ISNAN is undefined or false, then SQLite substitutes its own home-grown implementation of isnan().

HAVE_LOCALTIME_R

> If the HAVE_LOCALTIME_R option is true, then SQLite uses the threadsafe localtime_r() library routine instead of localtime() to help implement the localtime modifier to the built-in date and time functions.

HAVE_LOCALTIME_S

> If the HAVE_LOCALTIME_S option is true, then SQLite uses the threadsafe localtime_s() library routine instead of localtime() to help implement the localtime modifier to the built-in date and time functions.

HAVE_MALLOC_USABLE_SIZE

> If the HAVE_MALLOC_USABLE_SIZE option is true, then SQLite tries uses the malloc_usable_size() interface to find the size of a memory allocation obtained from the standard-library malloc() or realloc() routines. This option is only applicable if the standard-library malloc() is used. On Apple systems, "zone malloc" is used instead, and so this option is not applicable. And, of course, if the application supplies its own malloc implementation using SQLITE_CONFIG_MALLOC then this option has no effect.
>
> If the HAVE_MALLOC_USABLE_SIZE option is omitted or is false, then SQLite uses a wrapper around system malloc() and realloc() that enlarges each allocation by 8 bytes and writes the size of the allocation in the initial 8 bytes, and then SQLite also implements its own home-grown version of malloc_usable_size() that consults that 8-byte prefix to find the allocation size. This approach works but it is suboptimal. Applications are encouraged to use HAVE_MALLOC_USABLE_SIZE whenever possible.

HAVE_STRCHRNUL

> If the HAVE_STRCHRNUL option is true, then SQLite uses the strchrnul() library function. If this option is missing or false, then SQLite substitutes its own home-grown implementation of strchrnul().

HAVE_USLEEP

> If the HAVE_USLEEP option is true, then the default unix VFS uses the usleep() system call to implement the xSleep method. If this option is undefined or false, then xSleep on unix is implemented using sleep() which means that sqlite3_sleep() will have a minimum wait interval of 1000 milliseconds regardless of its argument.

HAVE_UTIME

> If the HAVE_UTIME option is true, then the built-in but non-standard "unix-dotfile" VFS will use the utime() system call, instead of utimes(), to set the last access time on the lock file.

1.2 Options To Set Default Parameter Values

SQLITE*DEFAULT_AUTOMATIC_INDEX*=<0 or 1>_

> This macro determines the initial setting for PRAGMA automatic_index for newly opened database connections. For all versions of SQLite through 3.7.17, automatic indices are normally enabled for new database connections if this compile-time option is omitted. However, that might change in future releases of SQLite.
>
> See also: SQLITE_OMIT_AUTOMATIC_INDEX

SQLITE*DEFAULT_AUTOVACUUM*=<0 or 1 or 2>_

> This macro determines if SQLite creates databases with the auto_vacuum flag set by default to OFF (0), FULL (1), or INCREMENTAL (2). The default value is 0 meaning that databases are created with auto-vacuum turned off. In any case the compile-time default may be overridden by the PRAGMA auto_vacuum command.

SQLITE*DEFAULT_CACHE_SIZE*=<N>_

> This macro sets the default maximum size of the page-cache for each attached database. A positive value means that the limit is N page. If N is negative that means to limit the cache size to -N*1024 bytes. The suggested maximum cache size can be overridden by the PRAGMA cache_size command. The default value is -2000, which translates into a maximum of 2048000 bytes per cache.

SQLITE*DEFAULT_FILE_FORMAT*=<1 or 4>_

> The default schema format number used by SQLite when creating new database files is set by this macro. The schema formats are all very similar. The difference between formats 1 and 4 is that format 4 understands descending indices and has a tighter encoding for boolean values.
>
> All versions of SQLite since 3.3.0 (2006-01-10) can read and write any schema format between 1 and 4. But older versions of SQLite might not be able to read formats greater than 1. So that older versions of SQLite will be able to read and write database files created by newer versions of SQLite, the default schema format was set to 1 for SQLite versions through 3.7.9 (2011-11-01). Beginning with version 3.7.10, the default schema format is 4.
>
> The schema format number for a new database can be set at runtime using the PRAGMA legacy_file_format command.

SQLITE*DEFAULT_FILE_PERMISSIONS=_N*

> The default numeric file permissions for newly created database files under unix. If not specified, the default is 0644 which means that the files is globally readable but only writable by the creator.

SQLITE*DEFAULT_FOREIGN_KEYS*=<0 or 1>_

> This macro determines whether enforcement of foreign key constraints is enabled or disabled by default for new database connections. Each database connection can always turn enforcement of foreign key constraints on and off and run-time using the foreign_keys pragma. Enforcement of foreign key constraints is normally off by default, but if this compile-time parameter is set to 1, enforcement of foreign key constraints will be on by default.

SQLITE*DEFAULT_MMAP_SIZE=_N*

> This macro sets the default limit on the amount of memory that will be used for memory-mapped I/O for each open database file. If the *N* is zero, then memory mapped I/O is disabled by default. This compile-time limit and the SQLITE_MAX_MMAP_SIZE can be modified at start-time using the sqlite3_config(SQLITE_CONFIG_MMAP_SIZE) call, or at run-time using the mmap_size pragma.

SQLITE*DEFAULT_JOURNAL_SIZE_LIMIT*=<bytes>_

> This option sets the size limit on rollback journal files in persistent journal mode and exclusive locking mode and on the size of the write-ahead log file in WAL mode. When this compile-time option is omitted there is no upper bound on the size of the rollback journals or write-ahead logs. The journal file size limit can be changed at run-time using the journal_size_limit pragma.

SQLITE*DEFAULT_LOCKING_MODE*=<1 or 0>_

> If set to 1, then the default locking_mode is set to EXCLUSIVE. If omitted or set to 0 then the default locking_mode is NORMAL.

SQLITE*DEFAULT_MEMSTATUS*=<1 or 0>_

> This macro is used to determine whether or not the features enabled and disabled using the SQLITE_CONFIG_MEMSTATUS argument to sqlite3_config() are available by default. The default value is 1 (SQLITE_CONFIG_MEMSTATUS related features enabled).

SQLITE*DEFAULT_PAGE_SIZE*=<bytes>_

> This macro is used to set the default page-size used when a database is created. The value assigned must be a power of 2. The default value is 4096. The compile-time default may be overridden at runtime by the PRAGMA page_size command.

SQLITE*DEFAULT_SYNCHRONOUS*=<0-3>_

> This macro determines the default value of the PRAGMA synchronous setting. If not overridden at compile-time, the default setting is 2 (FULL).

SQLITE*DEFAULT_WAL_SYNCHRONOUS*=<0-3>_

> This macro determines the default value of the PRAGMA synchronous setting for database files that open in WAL mode. If not overridden at compile-time, this value is the same as SQLITE_DEFAULT_SYNCHRONOUS.
>
> If SQLITE_DEFAULT_WAL_SYNCHRONOUS differs from SQLITE_DEFAULT_SYNCHRONOUS, and if the application has not modified the synchronous setting for the database file using the PRAGMA synchronous statement, then the synchronous setting is changed to value defined by SQLITE_DEFAULT_WAL_SYNCHRONOUS when the database connection switches into WAL mode for the first time. If the SQLITE_DEFAULT_WAL_SYNCHRONOUS value is not overridden at compile-time, then it will always be the same as SQLITE_DEFAULT_SYNCHRONOUS and so no automatic synchronous setting changes will ever occur.

SQLITE*DEFAULT_WAL_AUTOCHECKPOINT*=<pages>_

> This macro sets the default page count for the WAL automatic checkpointing feature. If unspecified, the default page count is 1000.

SQLITE*DEFAULT_WORKER_THREADS*=_N_

> This macro sets the default value for the SQLITE_LIMIT_WORKER_THREADS parameter. The SQLITE_LIMIT_WORKER_THREADS parameter sets the maximum number of auxiliary threads that a single prepared statement will launch to assist it with a query. If not specified, the default maximum is 0. The value set here cannot be more than SQLITE_MAX_WORKER_THREADS.

SQLITE_EXTRA_DURABLE

> The SQLITE_EXTRA_DURABLE compile-time option that used to cause the default PRAGMA synchronous setting to be EXTRA, rather than FULL. This option is no longer supported. Use SQLITE_DEFAULT_SYNCHRONOUS=3 instead.

SQLITE*FTS3_MAX_EXPR_DEPTH=_N*

> This macro sets the maximum depth of the search tree that corresponds to the right-hand side of the MATCH operator in an FTS3 or FTS4 full-text index. The full-text search uses a recursive algorithm, so the depth of the tree is limited to prevent using too much stack space. The default limit is 12. This limit is sufficient for up to 4095 search terms on the right-hand side of the MATCH operator and it holds stack space usage to less than 2000 bytes.
>
> For ordinary FTS3/FTS4 queries, the search tree depth is approximately the base-2 logarithm of the number of terms in the right-hand side of the MATCH operator. However, for phrase queries and NEAR queries the search tree depth is linear in the number of right-hand side terms. So the default depth limit of 12 is sufficient for up to 4095 ordinary terms on a MATCH, it is only sufficient for 11 or 12 phrase or NEAR terms. Even so, the default is more than enough for most application.

SQLITE_LIKE_DOESNT_MATCH_BLOBS

> This compile-time option causes the LIKE operator to always return False if either operand is a BLOB. The default behavior of LIKE is that BLOB operands are cast to TEXT before the comparison is done.
>
> This compile-time option makes SQLite run more efficiently when processing queries that use the LIKE operator, at the expense of breaking backwards compatibility. However, the backwards compatibility break may be only a technicality. There was a long-standing bug in the LIKE processing logic (see https://www.sqlite.org/src/info/05f43be8fdda9f) that caused it to misbehavior for BLOB operands and nobody observed that bug in nearly 10 years of active use. So for more users, it is probably safe to enable this compile-time option and thereby save a little CPU time on LIKE queries.
>
> This compile-time option affects the SQL LIKE operator only and has no impact on the sqlite3_strlike() C-language interface.

SQLITE*MAX_MMAP_SIZE=_N*

> This macro sets a hard upper bound on the amount of address space that can be used by any single database for memory-mapped I/O. Setting this value to 0 completely disables memory-mapped I/O and causes logic associated with memory-mapped I/O to be omitted from the build. This option does change the default memory-mapped I/O address space size (set by SQLITE_DEFAULT_MMAP_SIZE or sqlite3_config(SQLITE_CONFIG_MMAP_SIZE) or the run-time memory-mapped I/O address space size (set by sqlite3_file_control(SQLITE_FCNTL_MMAP_SIZE) or PRAGMA mmap_size) as long as those other settings are less than the maximum value defined here.

SQLITE*MAX_SCHEMA_RETRY=_N*

> Whenever the database schema changes, prepared statements are automatically reprepared to accommodate the new schema. There is a race condition here in that if one thread is constantly changing the schema, another thread might spin on reparses and repreparations of a prepared statement and never get any real work done. This parameter prevents an infinite loop by forcing the spinning thread to give up after a fixed number of attempts at recompiling the prepared statement. The default setting is 50 which is more than adequate for most applications.

SQLITE*MAX_WORKER_THREADS=_N*

> Set an upper bound on the sqlite3_limit(db,SQLITE_LIMIT_WORKER_THREADS,N) setting that determines the maximum number of auxiliary threads that a single prepared statement will use to aid with CPU-intensive computations (mostly sorting). See also the SQLITE_DEFAULT_WORKER_THREADS options.

SQLITE*MINIMUM_FILE_DESCRIPTOR=_N*

> The unix VFS will never use a file descriptor less than *N*. The default value of *N* is 3.
>
> Avoiding the use of low-numbered file descriptors is a defense against accidental database corruption. If a database file was opened using file descriptor 2, for example, and then an assert() failed and invoked write(2,...), that would likely cause database corruption by overwriting part of the database file with the assertion error message. Using only higher-valued file descriptors avoids this potential problem. The protection against using low-numbered file descriptors can be disabled by setting this compile-time option to 0.

SQLITE*POWERSAFE_OVERWRITE*=<0 or 1>_

> This option changes the default assumption about powersafe overwrite for the underlying filesystems for the unix and windows VFSes. Setting SQLITE_POWERSAFE_OVERWRITE to 1 causes SQLite to assume that application-level writes cannot changes bytes outside the range of bytes written even if the write occurs just before a power loss. With SQLITE_POWERSAFE_OVERWRITE set to 0, SQLite assumes that other bytes in the same sector with a written byte might be changed or damaged by a power loss.

SQLITE_REVERSE_UNORDERED_SELECTS

> This option causes the PRAGMA reverse_unordered_selects setting to be enabled by default. When enabled, SELECT statements that lack an ORDER BY clause will run in reverse order.
>
> This option is useful for detecting when applications (incorrectly) assume that the order of rows in a SELECT without an ORDER BY clause will always be the same.

SQLITE*SORTER_PMASZ=_N*

> If multi-threaded processing is enabled via the PRAGMA threads setting, then sort operations will attempt to start helper threads when the amount of content to be sorted exceeds the minimum of the cache_size and PMA Size determined by the SQLITE_CONFIG_PMASZ start-time option. This compile-time option sets the default value for the SQLITE_CONFIG_PMASZ start-time option. The default value is 250.

SQLITE*STMTJRNL_SPILL=_N*

> The SQLITE_STMTJRNL_SPILL compile-time option determines the default setting of the SQLITE_CONFIG_STMTJRNL_SPILL start-time setting. That setting determines the size threshold above which statement journals are moved from memory to disk.

SQLITE_WIN32_MALLOC

> This option enables the use of the Windows Heap API functions for memory allocation instead of the standard library malloc() and free() routines.

YYSTACKDEPTH=<*max_depth*>

> This macro sets the maximum depth of the LALR(1) stack used by the SQL parser within SQLite. The default value is 100. A typical application will use less than about 20 levels of the stack. Developers whose applications contain SQL statements that need more than 100 LALR(1) stack entries should seriously consider refactoring their SQL as it is likely to be well beyond the ability of any human to comprehend.

1.3 Options To Set Size Limits

There are compile-time options that will set upper bounds on the sizes of various structures in SQLite. The compile-time options normally set a hard upper bound that can be changed at run-time on individual database connections using the sqlite3_limit() interface.

The compile-time options for setting upper bounds are documented separately. The following is a list of the available settings:

- SQLITE_MAX_ATTACHED
- SQLITE_MAX_COLUMN
- SQLITE_MAX_COMPOUND_SELECT
- SQLITE_MAX_EXPR_DEPTH
- SQLITE_MAX_FUNCTION_ARG
- SQLITE_MAX_LENGTH
- SQLITE_MAX_LIKE_PATTERN_LENGTH
- SQLITE_MAX_PAGE_COUNT
- SQLITE_MAX_SQL_LENGTH
- SQLITE_MAX_VARIABLE_NUMBER

1.4 Options To Control Operating Characteristics

SQLITE_4_BYTE_ALIGNED_MALLOC

> On most systems, the malloc() system call returns a buffer that is aligned to an 8-byte boundary. But on some systems (ex: windows) malloc() returns 4-byte aligned pointer. This compile-time option must be used on systems that return 4-byte aligned pointers from malloc().

SQLITE_CASE_SENSITIVE_LIKE

> If this option is present, then the built-in LIKE operator will be case sensitive. This same effect can be achieved at run-time using the case_sensitive_like pragma.

SQLITE_DIRECT_OVERFLOW_READ

> When this option is present, content contained in overflow pages of the database file is read directly from disk, bypassing the page cache, during read transactions. In applications that do a lot of reads of large BLOBs, this option might improve read performance.

SQLITE_HAVE_ISNAN

> If this option is present, then SQLite will use the isnan() function from the system math library. This is an alias for the HAVE_ISNAN configuration option.

SQLITE*OS_OTHER*=<0 or 1>_

> The option causes SQLite to omit its built-in operating system interfaces for Unix, Windows, and OS/2. The resulting library will have no default operating system interface. Applications must use sqlite3_vfs_register() to register an appropriate interface before using SQLite. Applications must also supply implementations for the sqlite3_os_init() and sqlite3_os_end() interfaces. The usual practice is for the supplied sqlite3_os_init() to invoke sqlite3_vfs_register(). SQLite will automatically invoke sqlite3_os_init() when it initializes.
>
> This option is typically used when building SQLite for an embedded platform with a custom operating system.

SQLITE_SECURE_DELETE

> This compile-time option changes the default setting of the secure_delete pragma. When this option is not used, secure_delete defaults to off. When this option is present, secure_delete defaults to on.
>
> The secure_delete setting causes deleted content to be overwritten with zeros. There is a small performance penalty since additional I/O must occur. On the other hand, secure_delete can prevent fragments of sensitive information from lingering in unused parts of the database file after it has been deleted. See the documentation on the secure_delete pragma for additional information.

SQLITE*THREADSAFE*=<0 or 1 or 2>_

> This option controls whether or not code is included in SQLite to enable it to operate safely in a multithreaded environment. The default is SQLITE_THREADSAFE=1 which is safe for use in a multithreaded environment. When compiled with SQLITE_THREADSAFE=0 all mutexing code is omitted and it is unsafe to use SQLite in a multithreaded program. When compiled with SQLITE_THREADSAFE=2, SQLite can be used in a multithreaded program so long as no two threads attempt to use the same database connection (or any prepared statements derived from that database connection) at the same time.
>
> To put it another way, SQLITE_THREADSAFE=1 sets the default threading mode to Serialized. SQLITE_THREADSAFE=2 sets the default threading mode to Multi-threaded. And SQLITE_THREADSAFE=0 sets the threading mode to Single-threaded.
>
> The value of SQLITE_THREADSAFE can be determined at run-time using the sqlite3_threadsafe() interface.
>
> When SQLite has been compiled with SQLITE_THREADSAFE=1 or SQLITE_THREADSAFE=2 then the threading mode can be altered at run-time using the sqlite3_config() interface together with one of these verbs:
>
> - SQLITE_CONFIG_SINGLETHREAD
> - SQLITE_CONFIG_MULTITHREAD
> - SQLITE_CONFIG_SERIALIZED
>
> The SQLITE_OPEN_NOMUTEX and SQLITE_OPEN_FULLMUTEX flags to sqlite3_open_v2() can also be used to adjust the threading mode of individual database connections at run-time.
>
> Note that when SQLite is compiled with SQLITE_THREADSAFE=0, the code to make SQLite threadsafe is omitted from the build. When this occurs, it is impossible to change the threading mode at start-time or run-time.
>
> See the threading mode documentation for additional information on aspects of using SQLite in a multithreaded environment.

SQLITE*TEMP_STORE*=<0 through 3>_

> This option controls whether temporary files are stored on disk or in memory. The meanings for various settings of this compile-time option are as follows:
>
> | SQLITE_TEMP_STORE | Meaning |
> |---|---|
> | 0 | Always use temporary files |
> | 1 | Use files by default but allow the PRAGMA temp_store command to override |
> | 2 | Use memory by default but allow the PRAGMA temp_store command to override |
> | 3 | Always use memory |
>
> The default setting is 1. Additional information can be found in tempfiles.html.

SQLITE*TRACE_SIZE_LIMIT=_N*

> If this macro is defined to a positive integer *N*, then the length of strings and BLOB that are expanded into parameters in the output of sqlite3_trace() is limited to *N* bytes.

SQLITE_USE_URI

> This option causes the URI filename process logic to be enabled by default.

1.5 Options To Enable Features Normally Turned Off

SQLITE_ALLOW_URI_AUTHORITY

> URI filenames normally throws an error if the authority section is not either empty or "localhost". However, if SQLite is compiled with the SQLITE_ALLOW_URI_AUTHORITY compile-time option, then the URI is converted into a Uniform Naming Convention (UNC) filename and passed down to the underlying operating system that way.
>
> Some future versions of SQLite may change to enable this feature by default.

SQLITE*ALLOW_COVERING_INDEX_SCAN*=<0 or 1>_

> This C-preprocess macro determines the default setting of the SQLITE_CONFIG_COVERING_INDEX_SCAN configuration setting. It defaults to 1 (on) which means that covering indices are used for full table scans where possible, in order to reduce I/O and improve performance. However, the use of a covering index for a full scan will cause results to appear in a different order from legacy, which could cause some (incorrectly-coded) legacy applications to break. Hence, the covering index scan option can be disabled at compile-time on systems that what to minimize their risk of exposing errors in legacy applications.

SQLITE*ENABLE_8_3_NAMES*=<1 or 2>_

> If this C-preprocessor macro is defined, then extra code is included that allows SQLite to function on a filesystem that only support 8+3 filenames. If the value of this macro is 1, then the default behavior is to continue to use long filenames and to only use 8+3 filenames if the database connection is opened using URI filenames with the "`8_3_names=1`" query parameter. If the value of this macro is 2, then the use of 8+3 filenames becomes the default but may be disabled on using the `8_3_names=0` query parameter. See

SQLITE_ENABLE_API_ARMOR

> When defined, this C-preprocessor macro activates extra code that attempts to detect misuse of the SQLite API, such as passing in NULL pointers to required parameters or using objects after they have been destroyed.

SQLITE_ENABLE_ATOMIC_WRITE

> If this C-preprocessor macro is defined and if the xDeviceCharacteristics method of sqlite3_io_methods object for a database file reports (via one of the SQLITE_IOCAP_ATOMIC bits) that the filesystem supports atomic writes and if a transaction involves a change to only a single page of the database file, then the transaction commits with just a single write request of a single page of the database and no rollback journal is created or written. On filesystems that support atomic writes, this optimization can result in significant speed improvements for small updates. However, few filesystems support this capability and the code paths that check for this capability slow down write performance on systems that lack atomic write capability, so this feature is disabled by default.

SQLITE_ENABLE_COLUMN_METADATA

> When this C-preprocessor macro is defined, SQLite includes some additional APIs that provide convenient access to meta-data about tables and queries. The APIs that are enabled by this option are:
>
> - sqlite3_column_database_name()
> - sqlite3_column_database_name16()
> - sqlite3_column_table_name()
> - sqlite3_column_table_name16()
> - sqlite3_column_origin_name()
> - sqlite3_column_origin_name16()

SQLITE_ENABLE_DBSTAT_VTAB

> This option enables the dbstat virtual table.

SQLITE_ENABLE_EXPLAIN_COMMENTS

> This option adds extra logic to SQLite that inserts comment text into the output of EXPLAIN. These extra comments use extra memory, thus making prepared statements larger and very slightly slower, and so they are turned off by default and in most application. But some applications, such as the command-line shell for SQLite, value clarity of EXPLAIN output over raw performance and so this compile-time option is available to them. The SQLITE_ENABLE_EXPLAIN_COMMENTS compile-time option is also enabled automatically if SQLITE_DEBUG is enabled.

SQLITE_ENABLE_FTS3

> When this option is defined in the amalgamation, version 3 of the full-text search engine is added to the build automatically.

SQLITE_ENABLE_FTS3_PARENTHESIS

> This option modifies the query pattern parser in FTS3 such that it supports operators AND and NOT (in addition to the usual OR and NEAR) and also allows query expressions to contain nested parenthesis.

SQLITE_ENABLE_FTS3_TOKENIZER

> This option enables the two-argument version of the fts3_tokenizer() interface. The second argument to fts3_tokenizer() is suppose to be a pointer to a function (encoded as a BLOB) that implements an application defined tokenizer. If hostile actors are able to run the two-argument version of fts3_tokenizer() with an arbitrary second argument, they could use crash or take control of the process.
>
> Because of security concerns, the two-argument fts3_tokenizer() feature was disabled beginning with Version 3.11.0 unless this compile-time option is used. Version 3.12.0 added the sqlite3_db_config(db,SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER,1,0) interface that activates the two-argument version of fts3_tokenizer() for a specific database connection at run-time.

SQLITE_ENABLE_FTS4

> When this option is defined in the amalgamation, versions 3 and 4 of the full-text search engine is added to the build automatically.

SQLITE_ENABLE_FTS5

> When this option is defined in the amalgamation, versions 5 of the full-text search engine (fts5) is added to the build automatically.

SQLITE_ENABLE_ICU

> This option causes the International Components for Unicode or "ICU" extension to SQLite to be added to the build.

SQLITE_ENABLE_IOTRACE

> When both the SQLite core and the Command Line Interface (CLI) are both compiled with this option, then the CLI provides an extra command named ".iotrace" that provides a low-level log of I/O activity. This option is experimental and may be discontinued in a future release.

SQLITE_ENABLE_JSON1

> When this option is defined in the amalgamation, the JSON SQL functions are added to the build automatically.

SQLITE_ENABLE_LOCKING_STYLE

> This option enables additional logic in the OS interface layer for Mac OS X. The additional logic attempts to determine the type of the underlying filesystem and choose and alternative locking strategy that works correctly for that filesystem type. Five locking strategies are available:
>
> - POSIX locking style. This is the default locking style and the style used by other (non Mac OS X) Unixes. Locks are obtained and released using the fcntl() system call.
> - AFP locking style. This locking style is used for network file systems that use the AFP (Apple Filing Protocol) protocol. Locks are obtained by calling the library function _AFPFSSetLock(). *Flock locking style. This is used for file-systems that do not support POSIX locking style. Locks are obtained and released using the flock() system call.* Dot-file locking style. This locking style is used when neither flock nor POSIX locking styles are supported by the file system. Database locks are obtained by creating and entry in the file-system at a well-known location relative to the database file (a "dot-file") and relinquished by deleting the same file.* No locking style. If none of the above can be supported, this locking style is used. No database locking mechanism is used. When this system is used it is not safe for a single database to be accessed by multiple clients.
>
> Additionally, five extra VFS implementations are provided as well as the default. By specifying one of the extra VFS implementations when calling sqlite3_open_v2(), an application may bypass the file-system detection logic and explicitly select one of the above locking styles. The five extra VFS implementations are called "unix-posix", "unix-afp", "unix-flock", "unix-dotfile" and "unix-none".

SQLITE_ENABLE_MEMORY_MANAGEMENT

> This option adds extra logic to SQLite that allows it to release unused memory upon request. This option must be enabled in order for the sqlite3_release_memory() interface to work. If this compile-time option is not used, the sqlite3_release_memory() interface is a no-op.

SQLITE_ENABLE_MEMSYS3

> This option includes code in SQLite that implements an alternative memory allocator. This alternative memory allocator is only engaged when the SQLITE_CONFIG_HEAP option to sqlite3_config() is used to supply a large chunk of memory from which all memory allocations are taken. The MEMSYS3 memory allocator uses a hybrid allocation algorithm patterned after dlmalloc(). Only one of SQLITE_ENABLE_MEMSYS3 and SQLITE_ENABLE_MEMSYS5 may be enabled at once.

SQLITE_ENABLE_MEMSYS5

> This option includes code in SQLite that implements an alternative memory allocator. This alternative memory allocator is only engaged when the SQLITE_CONFIG_HEAP option to sqlite3_config() is used to supply a large chunk of memory from which all memory allocations are taken. The MEMSYS5 module rounds all allocations up to the next power of two and uses a first-fit, buddy-allocator algorithm that provides strong guarantees against fragmentation and breakdown subject to certain operating constraints.

SQLITE_ENABLE_RBU

> Enable the code the implements the RBU extension.

SQLITE_ENABLE_RTREE

> This option causes SQLite to include support for the R*Tree index extension.

SQLITE_ENABLE_STMT_SCANSTATUS

> This option enables the sqlite3_stmt_scanstatus() interface. The sqlite3_stmt_scanstatus() interface is normally omitted from the build because it imposes a small performance penalty, even on statements that do not use the feature.

SQLITE_RTREE_INT_ONLY

> If this option is used together with SQLITE_ENABLE_RTREE then the R*Tree extension will only store 32-bit signed integer coordinates and all internal computations will be done using integers instead of floating point numbers.

SQLITE_ENABLE_SQLLOG

> This option enables extra code (especially the SQLITE_CONFIG_SQLLOG option to sqlite3_config()) that can be used to create logs of all SQLite processing performed by an application. These logs can be useful in doing off-line analysis of the behavior of an application, and especially for performance analysis. In order for the SQLITE_ENABLE_SQLLOG option to be useful, some extra code is required. The "test_sqllog.c" source code file in the SQLite source tree is a working example of the required extra code. On unix and windows systems, a developer can append the text of the "test_sqllog.c" source code file to the end of an "sqlite3.c" amalgamation, recompile the application using the -DSQLITE_ENABLE_SQLLOG option, then control logging using environment variables. See the header comment on the "test_sqllog.c" source file for additional detail.

SQLITE_ENABLE_STAT2

> This option used to cause the ANALYZE command to collect index histogram data in the **sqlite_stat2** table. But that functionality was superceded by SQLITE_ENABLE_STAT3 as of SQLite version 3.7.9. The SQLITE_ENABLE_STAT2 compile-time option is now a no-op.

SQLITE_ENABLE_STAT3

> This option adds additional logic to the ANALYZE command and to the query planner that can help SQLite to chose a better query plan under certain situations. The ANALYZE command is enhanced to collect histogram data from the left-most column of each index and store that data in the sqlite_stat3 table. The query planner will then use the histogram data to help it make better index choices. Note, however, that the use of histogram data in query planner violates the query planner stability guarantee which is important to some applications.

SQLITE_ENABLE_STAT4

> This option adds additional logic to the ANALYZE command and to the query planner that can help SQLite to chose a better query plan under certain situations. The ANALYZE command is enhanced to collect histogram data from all columns of every index and store that data in the sqlite_stat4 table. The query planner will then use the histogram data to help it make better index choices. The downside of this compile-time option is that it violates the query planner stability guarantee making it more difficult to ensure consistent performance in mass-produced applications.
>
> SQLITE_ENABLE_STAT4 is an enhancement of SQLITE_ENABLE_STAT3. STAT3 only recorded histogram data for the left-most column of each index whereas the STAT4 enhancement records histogram data from all columns of each index. The SQLITE_ENABLE_STAT3 compile-time option is a no-op and is ignored if the SQLITE_ENABLE_STAT4 compile-time option is used.

SQLITE_ENABLE_TREE_EXPLAIN

> This compile-time option is no longer used.

SQLITE_ENABLE_UPDATE_DELETE_LIMIT

> This option enables an optional ORDER BY and LIMIT clause on UPDATE and DELETE statements.
>
> If this option is defined, then it must also be defined when using the 'lemon' tool to generate a parse.c file. Because of this, this option may only be used when the library is built from source, not from the amalgamation or from the collection of pre-packaged C files provided for non-Unix like platforms on the website.

SQLITE_ENABLE_UNLOCK_NOTIFY

> This option enables the sqlite3_unlock_notify() interface and its associated functionality. See the documentation titled Using the SQLite Unlock Notification Feature for additional information.

SQLITE_SOUNDEX

> This option enables the soundex() SQL function.

SQLITE_USE_FCNTL_TRACE

> This option causes SQLite to issue extra SQLITE_FCNTL_TRACE file controls to provide supplementary information to the VFS. The "vfslog.c" extension makes use of this to provide enhanced logs of VFS activity.

YYTRACKMAXSTACKDEPTH

> This option causes the LALR(1) parser stack depth to be tracked and reported using the sqlite3_status(SQLITE_STATUS_PARSER_STACK,...) interface. SQLite's LALR(1) parser has a fixed stack depth (determined at compile-time using the YYSTACKDEPTH options). This option can be used to help determine if an application is getting close to exceeding the maximum LALR(1) stack depth.

1.6 Options To Disable Features Normally Turned On

SQLITE_DISABLE_LFS

> If this C-preprocessor macro is defined, large file support is disabled.

SQLITE_DISABLE_DIRSYNC

> If this C-preprocessor macro is defined, directory syncs are disabled. SQLite typically attempts to sync the parent directory when a file is deleted to ensure the directory entries are updated immediately on disk.

SQLITE_DISABLE_FTS3_UNICODE

> If this C-preprocessor macro is defined, the unicode61 tokenizer in FTS3 is omitted from the build and is unavailable to applications.

SQLITE_DISABLE_FTS4_DEFERRED

> If this C-preprocessor macro disables the "deferred token" optimization in FTS4. The "deferred token" optimization avoids loading massive posting lists for terms that are in most documents of the collection and instead simply scans for those tokens in the document source. FTS4 should get exactly the same answer both with and without this optimization.

1.7 Options To Omit Features

The following options can be used to reduce the size of the compiled library by omitting unused features. This is probably only useful in embedded systems where space is especially tight, as even with all features included the SQLite library is relatively small. Don't forget to tell your compiler to optimize for binary size! (the -Os option if using GCC). Telling your compiler to optimize for size usually has a much larger impact on library footprint than employing any of these compile-time options. You should also verify that debugging options are disabled.

The macros in this section do not require values. The following compilation switches all have the same effect: -DSQLITE_OMIT_ALTERTABLE -DSQLITE_OMIT_ALTERTABLE=1 -DSQLITE_OMIT_ALTERTABLE=0

If any of these options are defined, then the same set of SQLITE*OMIT options must also be defined when using the 'lemon' tool to generate the parse.c file and when compiling the 'mkkeywordhash' tool which generates the keywordhash.h file. Because of this, these options may only be used when the library is built from canonical source, not from the amalgamation. Some SQLITEOMIT* options might work, or appear to work, when used with the amalgamation. But this is not guaranteed. In general, always compile from canonical sources in order to take advantage of SQLITE*OMIT** options.

> ***Important Note:*** *The SQLITE_OMIT options may not work with the amalgamation. SQLITEOMIT* compile-time options usually work correctly only when SQLite is built from canonical source files._

Special versions of the SQLite amalgamation that do work with a predetermined set of SQLITE*OMIT** options can be generated. To do so, make a copy of the Makefile.linux-gcc makefile template in the canonical source code distribution. Change the name of your copy to simply "Makefile". Then edit "Makefile" to set up appropriate compile-time options. Then type:

```
make clean; make sqlite3.c
```

The resulting "sqlite3.c" amalgamation code file (and its associated header file "sqlite3.h") can then be moved to a non-unix platform for final compilation using a native compiler.

The SQLITE*OMIT options are unsupported. By this we mean that an SQLITEOMIT* option that omits code from the build in the current release might become a no-op in the next release. Or the other way around: an SQLITE*OMIT that is a no-op in the current release might cause code to be excluded in the next release. Also, not all SQLITEOMIT* options are tested. Some SQLITE*OMIT** options might cause SQLite to malfunction and/or provide incorrect answers.

> ***Important Note:*** *The SQLITE_OMIT** compile-time options are unsupported._

SQLITE_OMIT_ALTERTABLE

> When this option is defined, the ALTER TABLE command is not included in the library. Executing an ALTER TABLE statement causes a parse error.

SQLITE_OMIT_ANALYZE

> When this option is defined, the ANALYZE command is omitted from the build.

SQLITE_OMIT_ATTACH

> When this option is defined, the ATTACH and DETACH commands are omitted from the build.

SQLITE_OMIT_AUTHORIZATION

> Defining this option omits the authorization callback feature from the library. The sqlite3_set_authorizer() API function is not present in the library.

SQLITE_OMIT_AUTOINCREMENT

> This option is used to omit the AUTOINCREMENT functionality. When this is macro is defined, columns declared as "INTEGER PRIMARY KEY AUTOINCREMENT" behave in the same way as columns declared as "INTEGER PRIMARY KEY" when a NULL is inserted. The sqlite_sequence system table is neither created, nor respected if it already exists.

SQLITE_OMIT_AUTOINIT

> For backwards compatibility with older versions of SQLite that lack the sqlite3_initialize() interface, the sqlite3_initialize() interface is called automatically upon entry to certain key interfaces such as sqlite3_open(), sqlite3_vfs_register(), and sqlite3_mprintf(). The overhead of invoking sqlite3_initialize() automatically in this way may be omitted by building SQLite with the SQLITE_OMIT_AUTOINIT C-preprocessor macro. When built using SQLITE_OMIT_AUTOINIT, SQLite will not automatically initialize itself and the application is required to invoke sqlite3_initialize() directly prior to beginning use of the SQLite library.

SQLITE_OMIT_AUTOMATIC_INDEX

> This option is used to omit the automatic indexing functionality. See also: SQLITE_DEFAULT_AUTOMATIC_INDEX.

SQLITE_OMIT_AUTORESET

> By default, the sqlite3_step() interface will automatically invoke sqlite3_reset() to reset the prepared statement if necessary. This compile-time option changes that behavior so that sqlite3_step() will return SQLITE_MISUSE if it called again after returning anything other than SQLITE_ROW, SQLITE_BUSY, or SQLITE_LOCKED unless there was an intervening call to sqlite3_reset().
>
> In SQLite version 3.6.23.1 and earlier, sqlite3_step() used to always return SQLITE_MISUSE if it was invoked again after returning anything other than SQLITE_ROW without an intervening call to sqlite3_reset(). This caused problems on some poorly written smartphone applications which did not correctly handle the SQLITE_LOCKED and SQLITE_BUSY error returns. Rather than fix the many defective smartphone applications, the behavior of SQLite was changed in 3.6.23.2 to automatically reset the prepared statement. But that changed caused issues in other improperly implemented applications that were actually looking for an SQLITE_MISUSE return to terminate their query loops. (Anytime an application gets an SQLITE_MISUSE error code from SQLite, that means the application is misusing the SQLite interface and is thus incorrectly implemented.) The SQLITE_OMIT_AUTORESET interface was added to SQLite version 3.7.5 in an effort to get all of the (broken) applications to work again without having to actually fix the applications.

SQLITE_OMIT_AUTOVACUUM

> If this option is defined, the library cannot create or write to databases that support auto_vacuum. Executing a PRAGMA auto_vacuum statement is not an error (since unknown PRAGMAs are silently ignored), but does not return a value or modify the auto-vacuum flag in the database file. If a database that supports auto-vacuum is opened by a library compiled with this option, it is automatically opened in read-only mode.

SQLITE_OMIT_BETWEEN_OPTIMIZATION

> This option disables the use of indices with WHERE clause terms that employ the BETWEEN operator.

SQLITE_OMIT_BLOB_LITERAL

> When this option is defined, it is not possible to specify a blob in an SQL statement using the X'ABCD' syntax.

SQLITE_OMIT_BTREECOUNT

> When this option is defined, an optimization that accelerates counting all entries in a table (in other words, an optimization that helps "SELECT count(*) FROM table" run faster) is omitted.

SQLITE_OMIT_BUILTIN_TEST

> A standard SQLite build includes a small amount of logic controlled by the sqlite3_test_control() interface that is used to exercise parts of the SQLite core that are difficult to control and measure using the standard API. This option omits that built-in test logic.

SQLITE_OMIT_CAST

> This option causes SQLite to omit support for the CAST operator.

SQLITE_OMIT_CHECK

> This option causes SQLite to omit support for CHECK constraints. The parser will still accept CHECK constraints in SQL statements, they will just not be enforced.

SQLITE_OMIT_COMPILEOPTION_DIAGS

> This option is used to omit the compile-time option diagnostics available in SQLite, including the sqlite3_compileoption_used() and sqlite3_compileoption_get() C/C++ functions, the sqlite_compileoption_used() and sqlite_compileoption_get() SQL functions, and the compile_options pragma.

SQLITE_OMIT_COMPLETE

> This option causes the sqlite3_complete() and sqlite3_complete16() interfaces to be omitted.

SQLITE_OMIT_COMPOUND_SELECT

> This option is used to omit the compound SELECT functionality. SELECT statements that use the UNION, UNION ALL, INTERSECT or EXCEPT compound SELECT operators will cause a parse error.
>
> An INSERT statement with multiple values in the VALUES clause is implemented internally as a compound SELECT. Hence, this option also disables the ability to insert more than a single row using an INSERT INTO ... VALUES ... statement.

SQLITE_OMIT_CTE

> This option causes support for common table expressions to be omitted.

SQLITE_OMIT_DATETIME_FUNCS

> If this option is defined, SQLite's built-in date and time manipulation functions are omitted. Specifically, the SQL functions julianday(), date(), time(), datetime() and strftime() are not available. The default column values CURRENT_TIME, CURRENT_DATE and CURRENT_TIMESTAMP are still available.

SQLITE_OMIT_DECLTYPE

> This option causes SQLite to omit support for the sqlite3_column_decltype() and sqlite3_column_decltype16() interfaces.

SQLITE_OMIT_DEPRECATED

> This option causes SQLite to omit support for interfaces marked as deprecated. This includes sqlite3_aggregate_count(), sqlite3_expired(), sqlite3_transfer_bindings(), sqlite3_global_recover(), sqlite3_thread_cleanup() and sqlite3_memory_alarm() interfaces.

SQLITE_OMIT_DISKIO

> This option omits all support for writing to the disk and forces databases to exist in memory only. This option has not been maintained and probably does not work with newer versions of SQLite.

SQLITE_OMIT_EXPLAIN

> Defining this option causes the EXPLAIN command to be omitted from the library. Attempting to execute an EXPLAIN statement will cause a parse error.

SQLITE_OMIT_FLAG_PRAGMAS

> This option omits support for a subset of PRAGMA commands that query and set boolean properties.

SQLITE_OMIT_FLOATING_POINT

> This option is used to omit floating-point number support from the SQLite library. When specified, specifying a floating point number as a literal (i.e. "1.01") results in a parse error.
>
> In the future, this option may also disable other floating point functionality, for example the sqlite3_result_double(), sqlite3_bind_double(), sqlite3_value_double() and sqlite3_column_double() API functions.

SQLITE_OMIT_FOREIGN_KEY

> If this option is defined, then foreign key constraint syntax is not recognized.

SQLITE_OMIT_GET_TABLE

> This option causes support for sqlite3_get_table() and sqlite3_free_table() to be omitted.

SQLITE_OMIT_INCRBLOB

> This option causes support for incremental BLOB I/O to be omitted.

SQLITE_OMIT_INTEGRITY_CHECK

> This option omits support for the integrity_check pragma.

SQLITE_OMIT_LIKE_OPTIMIZATION

> This option disables the ability of SQLite to use indices to help resolve LIKE and GLOB operators in a WHERE clause.

SQLITE_OMIT_LOAD_EXTENSION

> This option omits the entire extension loading mechanism from SQLite, including sqlite3_enable_load_extension() and sqlite3_load_extension() interfaces.

SQLITE_OMIT_LOCALTIME

> This option omits the "localtime" modifier from the date and time functions. This option is sometimes useful when trying to compile the date and time functions on a platform that does not support the concept of local time.

SQLITE_OMIT_LOOKASIDE

> This option omits the lookaside memory allocator.

SQLITE_OMIT_MEMORYDB

> When this is defined, the library does not respect the special database name ":memory:" (normally used to create an in-memory database). If ":memory:" is passed to sqlite3_open(), sqlite3_open16(), or sqlite3_open_v2(), a file with this name will be opened or created.

SQLITE_OMIT_OR_OPTIMIZATION

> This option disables the ability of SQLite to use an index together with terms of a WHERE clause connected by the OR operator.

SQLITE_OMIT_PAGER_PRAGMAS

> Defining this option omits pragmas related to the pager subsystem from the build.

SQLITE_OMIT_PRAGMA

> This option is used to omit the PRAGMA command from the library. Note that it is useful to define the macros that omit specific pragmas in addition to this, as they may also remove supporting code in other sub-systems. This macro removes the PRAGMA command only.

SQLITE_OMIT_PROGRESS_CALLBACK

> This option may be defined to omit the capability to issue "progress" callbacks during long-running SQL statements. The sqlite3_progress_handler() API function is not present in the library.

SQLITE_OMIT_QUICKBALANCE

> This option omits an alternative, faster B-Tree balancing routine. Using this option makes SQLite slightly smaller at the expense of making it run slightly slower.

SQLITE_OMIT_REINDEX

> When this option is defined, the REINDEX command is not included in the library. Executing a REINDEX statement causes a parse error.

SQLITE_OMIT_SCHEMA_PRAGMAS

> Defining this option omits pragmas for querying the database schema from the build.

SQLITE_OMIT_SCHEMA_VERSION_PRAGMAS

> Defining this option omits pragmas for querying and modifying the database schema version and user version from the build. Specifically, the schema_version and user_version PRAGMAs are omitted.

SQLITE_OMIT_SHARED_CACHE

> This option builds SQLite without support for shared-cache mode. The sqlite3_enable_shared_cache() is omitted along with a fair amount of logic within the B-Tree subsystem associated with shared cache management.

SQLITE_OMIT_SUBQUERY

> If defined, support for sub-selects and the IN() operator are omitted.

SQLITE_OMIT_TCL_VARIABLE

> If this macro is defined, then the special "$<variable-name>" syntax used to automatically bind SQL variables to TCL variables is omitted.</variable-name>

SQLITE_OMIT_TEMPDB

> This option omits support for TEMP or TEMPORARY tables.

SQLITE_OMIT_TRACE

> This option omits support for the sqlite3_profile() and sqlite3_trace() interfaces and their associated logic.

SQLITE_OMIT_TRIGGER

> Defining this option omits support for TRIGGER objects. Neither the CREATE TRIGGER or DROP TRIGGER commands are available in this case, and attempting to execute either will result in a parse error. This option also disables enforcement of foreign key constraints, since the code that implements triggers and which is omitted by this option is also used to implement foreign key actions.

SQLITE_OMIT_TRUNCATE_OPTIMIZATION

> A default build of SQLite, if a DELETE statement has no WHERE clause and operates on a table with no triggers, an optimization occurs that causes the DELETE to occur by dropping and recreating the table. Dropping and recreating a table is usually much faster than deleting the table content row by row. This is the "truncate optimization".

SQLITE_OMIT_UTF16

> This macro is used to omit support for UTF16 text encoding. When this is defined all API functions that return or accept UTF16 encoded text are unavailable. These functions can be identified by the fact that they end with '16', for example sqlite3_prepare16(), sqlite3_column_text16() and sqlite3_bind_text16().

SQLITE_OMIT_VACUUM

> When this option is defined, the VACUUM command is not included in the library. Executing a VACUUM statement causes a parse error.

SQLITE_OMIT_VIEW

> Defining this option omits support for VIEW objects. Neither the CREATE VIEW nor the DROP VIEW commands are available in this case, and attempting to execute either will result in a parse error.
>
> WARNING: If this macro is defined, it will not be possible to open a database for which the schema contains VIEW objects.

SQLITE_OMIT_VIRTUALTABLE

> This option omits support for the Virtual Table mechanism in SQLite.

SQLITE_OMIT_WAL

> This option omits the "write-ahead log" (a.k.a. "WAL") capability.

SQLITE_OMIT_WSD

> This option builds a version of the SQLite library that contains no Writable Static Data (WSD). WSD is global variables and/or static variables. Some platforms do not support WSD, and this option is necessary in order for SQLite to work those platforms.
>
> Unlike other OMIT options which make the SQLite library smaller, this option actually increases the size of SQLite and makes it run a little slower. Only use this option if SQLite is being built for an embedded target that does not support WSD.

SQLITE_OMIT_XFER_OPT

> This option omits support for optimizations that help statements of the form "INSERT INTO ... SELECT ..." run faster.

SQLITE_ZERO_MALLOC

> This option omits both the default memory allocator and the debugging memory allocator from the build and substitutes a stub memory allocator that always fails. SQLite will not run with this stub memory allocator since it will be unable to allocate memory. But this stub can be replaced at start-time using sqlite3_config(SQLITE_CONFIG_MALLOC,...) or sqlite3_config(SQLITE_CONFIG_HEAP,...). So the net effect of this compile-time option is that it allows SQLite to be compiled and linked against a system library that does not support malloc(), free(), and/or realloc().

1.8 Analysis and Debugging Options

SQLITE_DEBUG

> The SQLite source code contains literally thousands of assert() statements used to verify internal assumptions and subroutine preconditions and postconditions. These assert() statements are normally turned off (they generate no code) since turning them on makes SQLite run approximately three times slower. But for testing and analysis, it is useful to turn the assert() statements on. The SQLITE_DEBUG compile-time option does this.
>
> SQLITE_DEBUG also enables some other debugging features, such as special PRAGMA statements that turn on tracing and listing features used for troubleshooting and analysis of the VDBE and code generator.

SQLITE_MEMDEBUG

> The SQLITE_MEMDEBUG option causes an instrumented debugging memory allocator to be used as the default memory allocator within SQLite. The instrumented memory allocator checks for misuse of dynamically allocated memory. Examples of misuse include using memory after it is freed, writing off the ends of a memory allocation, freeing memory not previously obtained from the memory allocator, or failing to initialize newly allocated memory.

1.9 Windows-Specific Options

SQLITE_WIN32_HEAP_CREATE

> This option forces the Win32 native memory allocator, when enabled, to create a private heap to hold all memory allocations.

SQLITE_WIN32_MALLOC_VALIDATE

> This option forces the Win32 native memory allocator, when enabled, to make strategic calls into the HeapValidate() function if assert() is also enabled.

Upgrading SQLite, Backwards Compatibility

Moving From SQLite 3.5.9 to 3.6.0

SQLite version 3.6.0 contains many changes. As is the custom with the SQLite project, most changes are fully backwards compatible. However, a few of the changes in version 3.6.0 are incompatible and might require modifications to application code and/or makefiles. This document is a briefing on the changes in SQLite 3.6.0 with special attention to the incompatible changes.

> **Key Points:**
>
> - The database file format is unchanged.
> - All incompatibilities are on obscure interfaces and hence should have zero impact on most applications.

1.0 Incompatible Changes

Incompatible changes are covered first since they are the most important to maintainers and programmers.

1.1 Overview Of Incompatible Changes

1. Changes to the sqlite3_vfs object

    i. The signature of the xAccess method has been modified to return an error code and to store its output into an integer pointed to by a parameter, rather than returning the output directly. This change allows the xAccess() method to report failures. In association with this signature change, a new extended error code SQLITE_IOERR_ACCESS has been added.

    ii. The xGetTempname method has been removed from sqlite3_vfs. In its place, the xOpen method is enhanced to open a temporary file of its own invention when the filename parameter is NULL.

    iii. Added the xGetLastError() method to sqlite3_vfs for returning filesystem-specific error messages and error codes back to SQLite.

2. The signature of the xCheckReservedLock method on sqlite3_io_methods has been modified so that it returns an error code and stores its boolean result into an integer pointed to by a parameter. In association with this change, a new extended error code SQLITE_IOERR_CHECKRESERVEDLOCK has been added.

3. When SQLite is ported to new operation systems (operating systems other than Unix, Windows, and OS/2 for which ports are provided together with the core) two new functions, sqlite3_os_init() and sqlite3_os_end(), must be provided as part of the port.

4. The way in which the IN and NOT IN operators handle NULL values in their right-hand expressions has been brought into compliance with the SQL standard and with other SQL database engines.

5. The column names for the result sets of SELECT statements have been tweaked in some cases to work more like other SQL database engines.

6. Changes to compile-time options:

    i. The SQLITE_MUTEX_APPDEF compile-time parameter is no longer recognized. As a replacement, alternative mutex implementations may be created at runtime using sqlite3_config() with the SQLITE_CONFIG_MUTEX operator and the sqlite3_mutex_methods object.

    ii. Compile-time options OS*UNIX, OS_WIN, OS_OS2, OS_OTHER, and TEMP_STORE have been renamed to include an "SQLITE*" prefix in order to help avoid namespace collisions with application software. The new names of these options are respectively: SQLITE_OS_UNIX, SQLITE_OS_WIN, SQLITE_OS_OS2, SQLITE_OS_OTHER, and SQLITE_TEMP_STORE.

1.2 Changes To The VFS Layer

SQLite version 3.5.0 introduced a new OS interface layer that provided an abstraction of the underlying operating system. This was an important innovation and has proven to be helpful in porting and maintaining SQLite. However, the developers have discovered some minor flaws in the original "virtual file system" design introduced in version 3.5.0 and so SQLite 3.6.0 includes some small incompatible changes to address these flaws.

> **Key Point:** The incompatible changes in the SQLite operating-system interface for version 3.6.0 only affect the rare applications that make use of the virtual file system interface or that supply an application-defined mutex implementation or that make use of other obscure compile-time options. The changes introduced by SQLite version 3.6.0 will have zero impact on the vast majority of SQLite applications that use the built-in interfaces to Unix, Windows, and OS/2 and that use the standard build configuration.

1.3 Changes In The Way The IN Operator Handles NULLs

All versions of SQLite up to and including version 3.5.9 have mishandled NULL values on the right-hand side of IN and NOT IN operators. Specifically, SQLite has previously ignored NULLs on the right-hand side of IN and NOT IN.

Suppose we have a table X1 defined as follows:

```
CREATE TABLE x1(x INTEGER);
INSERT INTO x1 VALUES(1);
INSERT INTO x1 VALUES(2);
INSERT INTO x1 VALUES(NULL);
```

Given the definition of X1 above, the following expressions have historically evaluated to FALSE in SQLite, though the correct answer is actually NULL:

```
3 IN (1,2,NULL)
3 IN (SELECT * FROM x1)
```

Similarly, the following expressions have historically evaluated to TRUE when in fact NULL is also the correct answer here:

```
3 NOT IN (1,2,NULL)
3 NOT IN (SELECT * FROM x1)
```

The historical behavior of SQLite is incorrect according to the SQL:1999 standard and it is inconsistent with the behavior of MySQL and PostgreSQL. Version 3.6.0 changes the behavior of the IN and NOT IN operators to conform to the standard and to give the same results as other SQL database engines.

> **Key Point:** The change to the way NULL values are handled by the IN and NOT IN operators is technically a bug fix, not a design change. However, maintainers should check to ensure that applications do not depend on the older, buggy behavior prior to upgrading to version 3.6.0.

1.4 Changes To Column Naming Rules

The column names reported by join subqueries have been modified slightly in order to work more like other database engines. Consider the following query:

```
CREATE TABLE t1(a);
CREATE TABLE t2(x);
SELECT * FROM (SELECT t1.a FROM t1 JOIN t2 ORDER BY t2.x LIMIT 1) ORDER BY 1;
```

In version 3.5.9 the query above would return a single column named "t1.a". In version 3.6.0 the column name is just "a".

SQLite has never made any promises about the names of columns in the result set of SELECT statement unless the column contains an AS clause. So this change to column name is technically not an incompatibility. SQLite is merely changing from one undefined behavior to another. Nevertheless, many applications depend on the unspecified column naming behavior of SQLite and so this change is discussed under the incompatible changes subheading.

1.5 Changes To Compile-Time Options

Compile-time options to SQLite are controlled by C-preprocessor macros. SQLite version 3.6.0 changes the names of some of these macros so that all C-preprocessor macros that are specific to SQLite begin with the "SQLITE_" prefix. This is done to reduce the risk of name collisions with other software modules.

> **Key Point:** Changes to compile-time options have the potential to affect makefiles in projects that do customized builds of SQLite. These changes should have zero impact on application code and for most projects which use a standard, default build of SQLite.

2.0 Fully Backwards-Compatible Enhancements

In addition to the incompatible changes listed above, SQLite version 3.6.0 adds the following backwards compatible changes and enhancements:

1. The new sqlite3_config() interface allows an application to customize the behavior of SQLite at run-time. Customizations possible using sqlite3_config() include the following:
    1. Specify an alternative mutex implementation using the SQLITE_CONFIG_MUTEX verb with the sqlite3_mutex_methods object.
    2. Specify an alternative malloc implementation using the SQLITE_CONFIG_MALLOC verb with the sqlite3_mem_methods object.
    3. Partially or fully disable the use of mutexes using SQLITE_CONFIG_SINGLETHREAD, SQLITE_CONFIG_MULTITHREAD and SQLITE_CONFIG_SERIALIZED.
2. A new flag SQLITE_OPEN_NOMUTEX is made available to the sqlite3_open_v2() interface.
3. The new sqlite3_status() interface allows an application to query the performance status of SQLite at runtime.
4. The sqlite3_memory_used() and sqlite3_memory_highwater() interfaces are deprecated. The equivalent functionality is now available through sqlite3_status().

5. The sqlite3_initialize() interface can be called to explicitly initialize the SQLite subsystem. The sqlite3_initialize() interface is called automatically when invoking certain interfaces so the use of sqlite3_initialize() is not required, but it is recommended.

6. The sqlite3_shutdown() interface causes SQLite release any system resources (memory allocations, mutexes, open file handles) that it might have been allocated by sqlite3_initialize().

7. The sqlite3_next_stmt() interface allows an application to discover all prepared statements associated with a database connection.

8. Added the page_count PRAGMA for returning the size of the underlying database file in pages.

9. Added a new R*Tree index extension.

Moving From SQLite 3.4.2 to 3.5.0

SQLite version 3.5.0 introduces a new OS interface layer that is incompatible with all prior versions of SQLite. In addition, a few existing interfaces have been generalized to work across all database connections within a process rather than just all connections within a thread. The purpose of this article is to describe the changes to 3.5.0 in detail so that users of prior versions of SQLite can judge what, if any, effort will be required to upgrade to newer versions.

1.0 Overview Of Changes

A quick enumeration of the changes in SQLite version 3.5.0 is provide here. Subsequent sections will describe these changes in more detail.

1. The OS interface layer has been completely reworked:
    1. i. The undocumented **sqlite3_os_switch()** interface has been removed.
    2. ii. The **SQLITE_ENABLE_REDEF_IO** compile-time flag no longer functions. I/O procedures are now always redefinable.
    3. iii. Three new objects are defined for specifying I/O procedures: sqlite3_vfs, sqlite3_file, and sqlite3_io_methods.
    4. iv. Three new interfaces are used to create alternative OS interfaces: sqlite3_vfs_register(), sqlite3_vfs_unregister(), and sqlite3_vfs_find().
    5. v. A new interface has been added to provided additional control over the creation of new database connections: sqlite3_open_v2(). The legacy interfaces of sqlite3_open() and sqlite3_open16() continue to be fully supported.
2. The optional shared cache and memory management features that were introduced in version 3.3.0 can now be used across multiple threads within the same process. Formerly, these extensions only applied to database connections operating within a single thread.
    1. i. The sqlite3_enable_shared_cache() interface now applies to all threads within a process, not to just the one thread in which it was run.
    2. ii. The sqlite3_soft_heap_limit() interface now applies to all threads within a process, not to just the one thread in which it was run.
    3. iii. The sqlite3_release_memory() interface will now attempt to reduce the memory usages across all database connections in all threads, not just connections in the thread where the interface is called.
    4. iv. The sqlite3_thread_cleanup() interface has become a no-op.
3. Restrictions on the use of the same database connection by multiple threads have been

dropped. It is now safe for multiple threads to use the same database connection at the same time.

4. There is now a compile-time option that allows an application to define alternative malloc()/free() implementations without having to modify any core SQLite code.
5. There is now a compile-time option that allows an application to define alternative mutex implementations without having to modify any core SQLite code.

Of these changes, only 1a and 2a through 2c are incompatibilities in any formal sense. But users who have previously made custom modifications to the SQLite source (for example to add a custom OS layer for embedded hardware) might find that these changes have a larger impact. On the other hand, an important goal of these changes is to make it much easier to customize SQLite for use on different operating systems.

2.0 The OS Interface Layer

If your system defines a custom OS interface for SQLite or if you were using the undocumented **sqlite3_os_switch()** interface, then you will need to make modifications in order to upgrade to SQLite version 3.5.0. This may seem painful at first glance. But as you look more closely, you will probably discover that your changes are made smaller and easier to understand and manage by the new SQLite interface. It is likely that your changes will now also work seamlessly with the SQLite amalgamation. You will no longer need to make any changes to the code SQLite source code. All of your changes can be effected by application code and you can link against a standard, unmodified version of the SQLite amalgamation. Furthermore, the OS interface layer, which was formerly undocumented, is now an officially support interface for SQLite. So you have some assurance that this will be a one-time change and that your new backend will continue to work in future versions of SQLite.

2.1 The Virtual File System Object

The new OS interface for SQLite is built around an object named sqlite3_vfs. The "vfs" stands for "Virtual File System". The sqlite3_vfs object is basically a structure containing pointers to functions that implement the primitive disk I/O operations that SQLite needs to perform in order to read and write databases. In this article, we will often refer a sqlite3_vfs objects as a "VFS".

SQLite is able to use multiple VFSes at the same time. Each individual database connection is associated with just one VFS. But if you have multiple database connections, each connection can be associated with a different VFS.

There is always a default VFS. The legacy interfaces sqlite3_open() and sqlite3_open16() always use the default VFS. The new interface for creating database connections, sqlite3_open_v2(), allows you to specify which VFS you want to use by name.

2.1.1 Registering New VFS Objects

Standard builds of SQLite for Unix or Windows come with a single VFS named "unix" or "win32", as appropriate. This one VFS is also the default. So if you are using the legacy open functions, everything will continue to operate as it has before. The change is that an application now has the flexibility of adding new VFS modules to implement a customized OS layer. The sqlite3_vfs_register() API can be used to tell SQLite about one or more application-defined VFS modules:

```
int sqlite3_vfs_register(sqlite3_vfs*, int makeDflt);
```

Applications can call sqlite3_vfs_register at any time, though of course a VFS needs to be registered before it can be used. The first argument is a pointer to a customized VFS object that the application has prepared. The second argument is true to make the new VFS the default VFS so that it will be used by the legacy sqlite3_open() and sqlite3_open16() APIs. If the new VFS is not the default, then you will probably have to use the new sqlite3_open_v2() API to use it. Note, however, that if a new VFS is the only VFS known to SQLite (if SQLite was compiled without its usual default VFS or if the precompiled default VFS was removed using sqlite3_vfs_unregister()) then the new VFS automatic becomes the default VFS regardless of the makeDflt argument to sqlite3_vfs_register().

Standard builds include the default "unix" or "win32" VFSes. But if you use the -DOS_OTHER=1 compile-time option, then SQLite is built without a default VFS. In that case, the application must register at least one VFS prior to calling sqlite3_open(). This is the approach that embedded applications should use. Rather than modifying the SQLite source to insert an alternative OS layer as was done in prior releases of SQLite, instead compile an unmodified SQLite source file (preferably the amalgamation) with the -DOS_OTHER=1 option, then invoke sqlite3_vfs_register() to define the interface to the underlying filesystem prior to creating any database connections.

2.1.2 Additional Control Over VFS Objects

The sqlite3_vfs_unregister() API is used to remove an existing VFS from the system.

```
int sqlite3_vfs_unregister(sqlite3_vfs*);
```

The sqlite3_vfs_find() API is used to locate a particular VFS by name. Its prototype is as follows:

```
sqlite3_vfs *sqlite3_vfs_find(const char *zVfsName);
```

The argument is the symbolic name for the desired VFS. If the argument is a NULL pointer, then the default VFS is returned. The function returns a pointer to the sqlite3_vfs object that implements the VFS. Or it returns a NULL pointer if no object could be found that matched the search criteria.

2.1.3 Modifications Of Existing VFSes

Once a VFS has been registered, it should never be modified. If a change in behavior is required, a new VFS should be registered. The application could, perhaps, use sqlite3_vfs_find() to locate the old VFS, make a copy of the old VFS into a new sqlite3_vfs object, make the desired modifications to the new VFS, unregister the old VFS, the register the new VFS in its place. Existing database connections would continue to use the old VFS even after it is unregistered, but new database connections would use the new VFS.

2.1.4 The VFS Object

A VFS object is an instance of the following structure:

```
typedef struct sqlite3_vfs sqlite3_vfs;
struct sqlite3_vfs {
  int iVersion;            /* Structure version number */
  int szOsFile;            /* Size of subclassed sqlite3_file */
  int mxPathname;          /* Maximum file pathname length */
  sqlite3_vfs *pNext;      /* Next registered VFS */
  const char *zName;       /* Name of this virtual file system */
  void *pAppData;          /* Pointer to application-specific data */
  int (*xOpen)(sqlite3_vfs*, const char *zName, sqlite3_file*,
               int flags, int *pOutFlags);
  int (*xDelete)(sqlite3_vfs*, const char *zName, int syncDir);
  int (*xAccess)(sqlite3_vfs*, const char *zName, int flags);
  int (*xGetTempName)(sqlite3_vfs*, char *zOut);
  int (*xFullPathname)(sqlite3_vfs*, const char *zName, char *zOut);
  void *(*xDlOpen)(sqlite3_vfs*, const char *zFilename);
  void (*xDlError)(sqlite3_vfs*, int nByte, char *zErrMsg);
  void *(*xDlSym)(sqlite3_vfs*,void*, const char *zSymbol);
  void (*xDlClose)(sqlite3_vfs*, void*);
  int (*xRandomness)(sqlite3_vfs*, int nByte, char *zOut);
  int (*xSleep)(sqlite3_vfs*, int microseconds);
  int (*xCurrentTime)(sqlite3_vfs*, double*);
  /* New fields may be appended in figure versions.  The iVersion
  ** value will increment whenever this happens. */
};
```

To create a new VFS, an application fills in an instance of this structure with appropriate values and then calls sqlite3_vfs_register().

The iVersion field of sqlite3_vfs should be 1 for SQLite version 3.5.0. This number may increase in future versions of SQLite if we have to modify the VFS object in some way. We hope that this never happens, but the provision is made in case it does.

The szOsFile field is the size in bytes of the structure that defines an open file: the sqlite3_file object. This object will be described more fully below. The point here is that each VFS implementation can define its own sqlite3_file object containing whatever information the VFS implementation needs to store about an open file. SQLite needs to know how big this object is, however, in order to preallocate enough space to hold it.

The mxPathname field is the maximum length of a file pathname that this VFS can use. SQLite sometimes has to preallocate buffers of this size, so it should be as small as reasonably possible. Some filesystems permit huge pathnames, but in practice pathnames rarely extend beyond 100 bytes or so. You do not have to put the longest pathname that the underlying filesystem can handle here. You only have to put the longest pathname that you want SQLite to be able to handle. A few hundred is a good value in most cases.

The pNext field is used internally by SQLite. Specifically, SQLite uses this field to form a linked list of registered VFSes.

The zName field is the symbolic name of the VFS. This is the name that the sqlite3_vfs_find() compares against when it is looking for a VFS.

The pAppData pointer is unused by the SQLite core. The pointer is available to store auxiliary information that a VFS information might want to carry around.

The remaining fields of the sqlite3_vfs object all store pointer to functions that implement primitive operations. We call these "methods". The first methods, xOpen, is used to open files on the underlying storage media. The result is an sqlite3_file object. There are additional methods, defined by the sqlite3_file object itself that are used to read and write and close the file. The additional methods are detailed below. The filename is in UTF-8. SQLite will guarantee that the zFilename string passed to xOpen() is a full pathname as generated by xFullPathname() and that the string will be valid and unchanged until xClose() is called. So the sqlite3_file can store a pointer to the filename if it needs to remember the filename for some reason. The flags argument to xOpen() is a copy of the flags argument to sqlite3_open_v2(). If sqlite3_open() or sqlite3_open16() is used, then flags is SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE. If xOpen() opens a file read-only then it sets *pOutFlags to include SQLITE_OPEN_READONLY. Other bits in* pOutFlags may be set. SQLite will also add one of the following flags to the xOpen() call, depending on the object being opened:

- SQLITE_OPEN_MAIN_DB
- SQLITE_OPEN_MAIN_JOURNAL
- SQLITE_OPEN_TEMP_DB
- SQLITE_OPEN_TEMP_JOURNAL
- SQLITE_OPEN_TRANSIENT_DB
- SQLITE_OPEN_SUBJOURNAL

- SQLITE_OPEN_MASTER_JOURNAL

The file I/O implementation can use the object type flags to changes the way it deals with files. For example, an application that does not care about crash recovery or rollback, might make the open of a journal file a no-op. Writes to this journal are also a no-op. Any attempt to read the journal returns SQLITE_IOERR. Or the implementation might recognize the a database file will be doing page-aligned sector reads and writes in a random order and set up its I/O subsystem accordingly. SQLite might also add one of the following flags to the xOpen method:

- SQLITE_OPEN_DELETEONCLOSE
- SQLITE_OPEN_EXCLUSIVE

The SQLITE_OPEN_DELETEONCLOSE flag means the file should be deleted when it is closed. This will always be set for TEMP databases and journals and for subjournals. The SQLITE_OPEN_EXCLUSIVE flag means the file should be opened for exclusive access. This flag is set for all files except for the main database file. The sqlite3_file structure passed as the third argument to xOpen is allocated by the caller. xOpen just fills it in. The caller allocates a minimum of szOsFile bytes for the sqlite3_file structure.

The differences between an SQLITE_OPEN_TEMP_DB database and an SQLITE_OPEN_TRANSIENT_DB database is this: The SQLITE_OPEN_TEMP_DB is used for explicitly declared and named TEMP tables (using the CREATE TEMP TABLE syntax) or for named tables in a temporary database that is created by opening a database with a filename that is an empty string. An SQLITE_OPEN_TRANSIENT_DB holds a database table that SQLite creates automatically in order to evaluate a subquery or ORDER BY or GROUP BY clause. Both TEMP_DB and TRANSIENT_DB databases are private and are deleted automatically. TEMP_DB databases last for the duration of the database connection. TRANSIENT_DB databases last only for the duration of a single SQL statement.

The xDelete method is used delete a file. The name of the file is given in the second parameter. The filename will be in UTF-8. The VFS must convert the filename into whatever character representation the underlying operating system expects. If the syncDir parameter is true, then the xDelete method should not return until the change to the directory contents for the directory containing the deleted file have been synced to disk in order to ensure that the file does not "reappear" if a power failure occurs soon after.

The xAccess method is used to check for access permissions on a file. The filename will be UTF-8 encoded. The flags argument will be SQLITE_ACCESS_EXISTS to check for the existence of the file, SQLITE_ACCESS_READWRITE to check to see if the file is both readable and writable, or SQLITE_ACCESS_READ to check to see if the file is at least readable. The "file" named by the second parameter might be a directory or folder name.

The xGetTempName method computes the name of a temporary file that SQLite can use. The name should be written into the buffer given by the second parameter. SQLite will size that buffer to hold at least mxPathname bytes. The generated filename should be in UTF-8. To avoid security problems, the generated temporary filename should contain enough randomness to prevent an attacker from guessing the temporary filename in advance.

The xFullPathname method is used to convert a relative pathname into a full pathname. The resulting full pathname is written into the buffer provided by the third parameter. SQLite will size the output buffer to at least mxPathname bytes. Both the input and output names should be in UTF-8.

The xDlOpen, xDlError, xDlSym, and xDlClose methods are all used for accessing shared libraries at run-time. These methods may be omitted (and their pointers set to zero) if the library is compiled with SQLITE_OMIT_LOAD_EXTENSION or if the sqlite3_enable_load_extension() interface is never used to enable dynamic extension loading. The xDlOpen method opens a shared library or DLL and returns a pointer to a handle. NULL is returned if the open fails. If the open fails, the xDlError method can be used to obtain a text error message. The message is written into the zErrMsg buffer of the third parameter which is at least nByte bytes in length. The xDlSym returns a pointer to a symbol in the shared library. The name of the symbol is given by the second parameter. UTF-8 encoding is assumed. If the symbol is not found a NULL pointer is returned. The xDlClose routine closes the shared library.

The xRandomness method is used exactly once to initialize the pseudo-random number generator (PRNG) inside of SQLite. Only the xRandomness method on the default VFS is used. The xRandomness methods on other VFSes are never accessed by SQLite. The xRandomness routine requests that nByte bytes of randomness be written into zOut. The routine returns the actual number of bytes of randomness obtained. The quality of the randomness so obtained will determine the quality of the randomness generated by built-in SQLite functions such as random() and randomblob(). SQLite also uses its PRNG to generate temporary file names.. On some platforms (ex: Windows) SQLite assumes that temporary file names are unique without actually testing for collisions, so it is important to have good-quality randomness even if the random() and randomblob() functions are never used.

The xSleep method is used to suspend the calling thread for at least the number of microseconds given. This method is used to implement the sqlite3_sleep() and sqlite3_busy_timeout() APIs. In the case of sqlite3_sleep() the xSleep method of the default VFS is always used. If the underlying system does not have a microsecond resolution sleep capability, then the sleep time should be rounded up. xSleep returns this rounded-up value.

The xCurrentTime method finds the current time and date and writes the result as double-precision floating point value into pointer provided by the second parameter. The time and date is in coordinated universal time (UTC) and is a fractional Julian day number.

2.1.5 The Open File Object

The result of opening a file is an instance of an sqlite3_file object. The sqlite3_file object is an abstract base class defined as follows:

```
typedef struct sqlite3_file sqlite3_file;
struct sqlite3_file {
  const struct sqlite3_io_methods *pMethods;
};
```

Each VFS implementation will subclass the sqlite3_file by adding additional fields at the end to hold whatever information the VFS needs to know about an open file. It does not matter what information is stored as long as the total size of the structure does not exceed the szOsFile value recorded in the sqlite3_vfs object.

The sqlite3_io_methods object is a structure that contains pointers to methods for reading, writing, and otherwise dealing with files. This object is defined as follows:

```
typedef struct sqlite3_io_methods sqlite3_io_methods;
struct sqlite3_io_methods {
  int iVersion;
  int (*xClose)(sqlite3_file*);
  int (*xRead)(sqlite3_file*, void*, int iAmt, sqlite3_int64 iOfst);
  int (*xWrite)(sqlite3_file*, const void*, int iAmt, sqlite3_int64 iOfst);
  int (*xTruncate)(sqlite3_file*, sqlite3_int64 size);
  int (*xSync)(sqlite3_file*, int flags);
  int (*xFileSize)(sqlite3_file*, sqlite3_int64 *pSize);
  int (*xLock)(sqlite3_file*, int);
  int (*xUnlock)(sqlite3_file*, int);
  int (*xCheckReservedLock)(sqlite3_file*);
  int (*xFileControl)(sqlite3_file*, int op, void *pArg);
  int (*xSectorSize)(sqlite3_file*);
  int (*xDeviceCharacteristics)(sqlite3_file*);
  /* Additional methods may be added in future releases */
};
```

The iVersion field of sqlite3_io_methods is provided as insurance against future enhancements. The iVersion value should always be 1 for SQLite version 3.5.

The xClose method closes the file. The space for the sqlite3_file structure is deallocated by the caller. But if the sqlite3_file contains pointers to other allocated memory or resources, those allocations should be released by the xClose method.

The xRead method reads iAmt bytes from the file beginning at a byte offset to iOfst. The data read is stored in the pointer of the second parameter. xRead returns the SQLITE_OK on success, SQLITE_IOERR_SHORT_READ if it was not able to read the full number of bytes because it reached end-of-file, or SQLITE_IOERR_READ for any other error.

The xWrite method writes iAmt bytes of data from the second parameter into the file beginning at an offset of iOfst bytes. If the size of the file is less than iOfst bytes prior to the write, then xWrite should ensure that the file is extended with zeros up to iOfst bytes prior to beginning its write. xWrite continues to extends the file as necessary so that the size of the file is at least iAmt+iOfst bytes at the conclusion of the xWrite call. The xWrite method returns SQLITE_OK on success. If the write cannot complete because the underlying storage medium is full, then SQLITE_FULL is returned. SQLITE_IOERR_WRITE should be returned for any other error.

The xTruncate method truncates a file to be nByte bytes in length. If the file is already nByte bytes or less in length then this method is a no-op. The xTruncate method returns SQLITE_OK on success and SQLITE_IOERR_TRUNCATE if anything goes wrong.

The xSync method is used to force previously written data out of operating system cache and into non-volatile memory. The second parameter is usually SQLITE_SYNC_NORMAL. If the second parameter is SQLITE_SYNC_FULL then the xSync method should make sure that data has also been flushed through the disk controllers cache. The SQLITE_SYNC_FULL parameter is the equivalent of the F_FULLSYNC ioctl() on Mac OS X. The xSync method returns SQLITE_OK on success and SQLITE_IOERR_FSYNC if anything goes wrong.

The xFileSize() method determines the current size of the file in bytes and writes that value into *pSize. It returns SQLITE_OK on success and SQLITE_IOERR_FSTAT if something goes wrong.

The xLock and xUnlock methods are used to set and clear file locks. SQLite supports five levels of file locks, in order:

- SQLITE_LOCK_NONE
- SQLITE_LOCK_SHARED
- SQLITE_LOCK_RESERVED
- SQLITE_LOCK_PENDING
- SQLITE_LOCK_EXCLUSIVE

The underlying implementation can support some subset of these locking levels as long as it meets the other requirements of this paragraph. The locking level is specified as the second argument to both xLock and xUnlock. The xLock method increases the locking level to the specified locking level or higher. The xUnlock method decreases the locking level to no lower than the level specified. SQLITE_LOCK_NONE means that the file is unlocked. SQLITE_LOCK_SHARED gives permission to read the file. Multiple database connections can hold SQLITE_LOCK_SHARED at the same time. SQLITE_LOCK_RESERVED is like SQLITE_LOCK_SHARED in that its is permission to read the file. But only a single connection can hold a reserved lock at any point in time. The SQLITE_LOCK_PENDING is also permission to read the file. Other connections can continue to read the file as well, but no other connection is allowed to escalate a lock from none to shared. SQLITE_LOCK_EXCLUSIVE is permission to write on the file. Only a single connection can hold an exclusive lock and no other connection can hold any lock (other than "none") while one connection is hold an exclusive lock. The xLock returns SQLITE_OK on success, SQLITE_BUSY if it is unable to obtain the lock, or SQLITE_IOERR_RDLOCK if something else goes wrong. The xUnlock method returns SQLITE_OK on success and SQLITE_IOERR_UNLOCK for problems.

The xCheckReservedLock method checks to see if another connection or another process is currently holding a reserved, pending, or exclusive lock on the file. It returns true or false.

The xFileControl() method is a generic interface that allows custom VFS implementations to directly control an open file using the (new and experimental) sqlite3_file_control() interface. The second "op" argument is an integer opcode. The third argument is a generic pointer which is intended to be a pointer to a structure that may contain arguments or space in which to write return values. Potential uses for xFileControl() might be functions to enable blocking locks with timeouts, to change the locking strategy (for example to use dot-file locks), to inquire about the status of a lock, or to break stale locks. The SQLite core reserves opcodes less than 100 for its own use. A list of opcodes less than 100 is available. Applications that define a custom xFileControl method should use opcodes greater than 100 to avoid conflicts.

The xSectorSize returns the "sector size" of the underlying non-volatile media. A "sector" is defined as the smallest unit of storage that can be written without disturbing adjacent storage. On a disk drive the "sector size" has until recently been 512 bytes, though there is a push to increase this value to 4KiB. SQLite needs to know the sector size so that it can write a full sector at a time, and thus avoid corrupting adjacent storage space if a power lose occurs in the middle of a write.

The xDeviceCharacteristics method returns an integer bit vector that defines any special properties that the underlying storage medium might have that SQLite can use to increase performance. The allowed return is the bit-wise OR of the following values:

- SQLITE_IOCAP_ATOMIC
- SQLITE_IOCAP_ATOMIC512
- SQLITE_IOCAP_ATOMIC1K
- SQLITE_IOCAP_ATOMIC2K
- SQLITE_IOCAP_ATOMIC4K
- SQLITE_IOCAP_ATOMIC8K
- SQLITE_IOCAP_ATOMIC16K
- SQLITE_IOCAP_ATOMIC32K
- SQLITE_IOCAP_ATOMIC64K

- SQLITE_IOCAP_SAFE_APPEND
- SQLITE_IOCAP_SEQUENTIAL

The SQLITE_IOCAP_ATOMIC bit means that all writes to this device are atomic in the sense that either the entire write occurs or none of it occurs. The other SQLITE*IOCAP_ATOMIC_nnn* values indicate that writes of aligned blocks of the indicated size are atomic. SQLITE_IOCAP_SAFE_APPEND means that when extending a file with new data, the new data is written first and then the file size is updated. So if a power failure occurs, there is no chance that the file might have been extended with randomness. The SQLITE_IOCAP_SEQUENTIAL bit means that all writes occur in the order that they are issued and are not reordered by the underlying file system.

2.1.6 Checklist For Constructing A New VFS

The preceding paragraphs contain a lot of information. To ease the task of constructing a new VFS for SQLite we offer the following implementation checklist:

1. Define an appropriate subclass of the sqlite3_file object.
2. Implement the methods required by the sqlite3_io_methods object.
3. Create a static and constant sqlite3_io_methods object containing pointers to the methods from the previous step.
4. Implement the xOpen method that opens a file and populates an sqlite3_file object, including setting pMethods to point to the sqlite3_io_methods object from the previous step.
5. Implement the other methods required by sqlite3_vfs.
6. Define a static (but not constant) sqlite3_vfs structure that contains pointers to the xOpen method and the other methods and which contains the appropriate values for iVersion, szOsFile, mxPathname, zName, and pAppData.
7. Implement a procedure that calls sqlite3_vfs_register() and passes it a pointer to the sqlite3_vfs structure from the previous step. This procedure is probably the only exported symbol in the source file that implements your VFS.

Within your application, call the procedure implemented in the last step above as part of your initialization process before any database connections are opened.

3.0 The Memory Allocation Subsystem

Beginning with version 3.5, SQLite obtains all of the heap memory it needs using the routines sqlite3_malloc(), sqlite3_free(), and sqlite3_realloc(). These routines have existed in prior versions of SQLite, but SQLite has previously bypassed these routines and used its own memory allocator. This all changes in version 3.5.0.

The SQLite source tree actually contains multiple versions of the memory allocator. The default high-speed version found in the "mem1.c" source file is used for most builds. But if the SQLITE_MEMDEBUG flag is enabled, a separate memory allocator the "mem2.c" source file is used instead. The mem2.c allocator implements lots of hooks to do error checking and to simulate memory allocation failures for testing purposes. Both of these allocators use the malloc()/free() implementation in the standard C library.

Applications are not required to use either of these standard memory allocators. If SQLite is compiled with SQLITE_OMIT_MEMORY_ALLOCATION then no implementation for the sqlite3_malloc(), sqlite3_realloc(), and sqlite3_free() functions is provided. Instead, the application that links against SQLite must provide its own implementation of these functions. The application provided memory allocator is not required to use the malloc()/free() implementation in the standard C library. An embedded application might provide an alternative memory allocator that uses memory for a fixed memory pool set aside for the exclusive use of SQLite, for example.

Applications that implement their own memory allocator must provide implementation for the usual three allocation functions sqlite3_malloc(), sqlite3_realloc(), and sqlite3_free(). And they must also implement a fourth function:

```
int sqlite3_memory_alarm(
  void(*xCallback)(void *pArg, sqlite3_int64 used, int N),
  void *pArg,
  sqlite3_int64 iThreshold
);
```

The sqlite3_memory_alarm routine is used to register a callback on memory allocation events. This routine registers or clears a callbacks that fires when the amount of memory allocated exceeds iThreshold. Only a single callback can be registered at a time. Each call to sqlite3_memory_alarm() overwrites the previous callback. The callback is disabled by setting xCallback to a NULL pointer.

The parameters to the callback are the pArg value, the amount of memory currently in use, and the size of the allocation that provoked the callback. The callback will presumably invoke sqlite3_free() to free up memory space. The callback may invoke sqlite3_malloc() or sqlite3_realloc() but if it does, no additional callbacks will be invoked by the recursive calls.

The sqlite3_soft_heap_limit() interface works by registering a memory alarm at the soft heap limit and invoking sqlite3_release_memory() in the alarm callback. Application programs should not attempt to use the sqlite3_memory_alarm() interface because doing so will interfere with the sqlite3_soft_heap_limit() module. This interface is exposed only so that applications can provide their own alternative implementation when the SQLite core is compiled with SQLITE_OMIT_MEMORY_ALLOCATION.

The built-in memory allocators in SQLite also provide the following additional interfaces:

```
sqlite3_int64 sqlite3_memory_used(void);
sqlite3_int64 sqlite3_memory_highwater(int resetFlag);
```

These interfaces can be used by an application to monitor how much memory SQLite is using. The sqlite3_memory_used() routine returns the number of bytes of memory currently in use and the sqlite3_memory_highwater() returns the maximum instantaneous memory usage. Neither routine includes the overhead associated with the memory allocator. These routines are provided for use by the application. SQLite never invokes them itself. So if the application is providing its own memory allocation subsystem, it can omit these interfaces if desired.

4.0 The Mutex Subsystem

SQLite has always been threadsafe in the sense that it is safe to use different SQLite database connections in different threads at the same time. The constraint was that the same database connection could not be used in two separate threads at once. SQLite version 3.5.0 relaxes this constraint.

In order to allow multiple threads to use the same database connection at the same time, SQLite must make extensive use of mutexes. And for this reason a new mutex subsystem as been added. The mutex subsystem as the following interface:

```
sqlite3_mutex *sqlite3_mutex_alloc(int);
void sqlite3_mutex_free(sqlite3_mutex*);
void sqlite3_mutex_enter(sqlite3_mutex*);
int sqlite3_mutex_try(sqlite3_mutex*);
void sqlite3_mutex_leave(sqlite3_mutex*);
```

Though these routines exist for the use of the SQLite core, application code is free to use these routines as well, if desired. A mutex is an sqlite3_mutex object. The sqlite3_mutex_alloc() routine allocates a new mutex object and returns a pointer to it. The argument to sqlite3_mutex_alloc() should be SQLITE_MUTEX_FAST or SQLITE_MUTEX_RECURSIVE for non-recursive and recursive mutexes, respectively. If the underlying system does not provide non-recursive mutexes, then a recursive mutex can be substituted in that case. The argument to sqlite3_mutex_alloc() can also be a constant designating one of several static mutexes:

- SQLITE_MUTEX_STATIC_MASTER
- SQLITE_MUTEX_STATIC_MEM
- SQLITE_MUTEX_STATIC_MEM2
- SQLITE_MUTEX_STATIC_PRNG

- SQLITE_MUTEX_STATIC_LRU

These static mutexes are reserved for use internally by SQLite and should not be used by the application. The static mutexes are all non-recursive.

The sqlite3_mutex_free() routine should be used to deallocate a non-static mutex. If a static mutex is passed to this routine then the behavior is undefined.

The sqlite3_mutex_enter() attempts to enter the mutex and blocks if another threads is already there. sqlite3_mutex_try() attempts to enter and returns SQLITE_OK on success or SQLITE_BUSY if another thread is already there. sqlite3_mutex_leave() exits a mutex. The mutex is held until the number of exits matches the number of entrances. If sqlite3_mutex_leave() is called on a mutex that the thread is not currently holding, then the behavior is undefined. If any routine is called for a deallocated mutex, then the behavior is undefined.

The SQLite source code provides multiple implementations of these APIs, suitable for varying environments. If SQLite is compiled with the SQLITE_THREADSAFE=0 flag then a no-op mutex implementation that is fast but does no real mutual exclusion is provided. That implementation is suitable for use in single-threaded applications or applications that only use SQLite in a single thread. Other real mutex implementations are provided based on the underlying operating system.

Embedded applications may wish to provide their own mutex implementation. If SQLite is compiled with the -DSQLITE_MUTEX_APPDEF=1 compile-time flag then the SQLite core provides no mutex subsystem and a mutex subsystem that matches the interface described above must be provided by the application that links against SQLite.

5.0 Other Interface Changes

Version 3.5.0 of SQLite changes the behavior of a few APIs in ways that are technically incompatible. However, these APIs are seldom used and even when they are used it is difficult to imagine a scenario where the change might break something. The changes actually makes these interface much more useful and powerful.

Prior to version 3.5.0, the sqlite3_enable_shared_cache() API would enable and disable the shared cache feature for all connections within a single thread - the same thread from which the sqlite3_enable_shared_cache() routine was called. Database connections that used the shared cache were restricted to running in the same thread in which they were opened. Beginning with version 3.5.0, the sqlite3_enable_shared_cache() applies to all database connections in all threads within the process. Now database connections running in separate threads can share a cache. And database connections that use shared cache can migrate from one thread to another.

Prior to version 3.5.0 the sqlite3_soft_heap_limit() set an upper bound on heap memory usage for all database connections within a single thread. Each thread could have its own heap limit. Beginning in version 3.5.0, there is a single heap limit for the entire process. This seems more restrictive (one limit as opposed to many) but in practice it is what most users want.

Prior to version 3.5.0 the sqlite3_release_memory() function would try to reclaim memory from all database connections in the same thread as the sqlite3_release_memory() call. Beginning with version 3.5.0, the sqlite3_release_memory() function will attempt to reclaim memory from all database connections in all threads.

6.0 Summary

The transition from SQLite version 3.4.2 to 3.5.0 is a major change. Every source code file in the SQLite core had to be modified, some extensively. And the change introduced some minor incompatibilities in the C interface. But we feel that the benefits of the transition from 3.4.2 to 3.5.0 far outweigh the pain of porting. The new VFS layer is now well-defined and stable and should simplify future customizations. The VFS layer, and the separable memory allocator and mutex subsystems allow a standard SQLite source code amalgamation to be used in an embedded project without change, greatly simplifying configuration management. And the resulting system is much more tolerant of highly threaded designs.

Release History

This page provides a high-level summary of changes to SQLite. For more detail, see the Fossil checkin logs at http://www.sqlite.org/src/timeline and http://www.sqlite.org/src/timeline?t=release. See the chronology a succinct listing of releases.

2016-03-29 (3.12.0)

Potentially Disruptive Change:

- The SQLITE_DEFAULT_PAGE_SIZE is increased from 1024 to 4096. The SQLITE_DEFAULT_CACHE_SIZE is changed from 2000 to -2000 so the same amount of cache memory is used by default. See the application note on the version 3.12.0 page size change for further information.

  Performance enhancements:

- Enhancements to the Lemon parser generator so that it creates a smaller and faster SQL parser.

- Only create master journal files if two or more attached databases are all modified, do not have PRAGMA synchronous set to OFF, and do not have the journal_mode set to OFF, MEMORY, or WAL.
- Only create statement journal files when their size exceeds a threshold. Otherwise the journal is held in memory and no I/O occurs. The threshold can be configured at compile-time using SQLITE_STMTJRNL_SPILL or at start-time using sqlite3_config(SQLITE_CONFIG_STMTJRNL_SPILL).
- The query planner is able to optimize IN operators on virtual tables even if the xBestIndex method does not set the sqlite3_index_constraint_usage.omit flag of the virtual table column to the left of the IN operator.
- The query planner now does a better job of optimizing virtual table accesses in a 3-way or higher join where constraints on the virtual table are split across two or more other tables of the join.
- More efficient handling of application-defined SQL functions, especially in cases where the application defines hundreds or thousands of custom functions.
- The query planner considers the LIMIT clause when estimating the cost of ORDER BY.
- The configure script (on unix) automatically detects pread() and pwrite() and sets compile-time options to use those OS interfaces if they are available.
- Reduce the amount of memory needed to hold the schema.

- Other miscellaneous micro-optimizations for improved performance and reduced memory usage.

New Features:

- Added the SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER option to sqlite3_db_config() which allows the two-argument version of the fts3_tokenizer() SQL function to be enabled or disabled at run-time.
- Added the sqlite3rbu_bp_progress() interface to the RBU extension.
- The PRAGMA defer_foreign_keys=ON statement now also disables RESTRICT actions on foreign key.
- Added the sqlite3_system_errno() interface.
- Added the SQLITE_DEFAULT_SYNCHRONOUS and SQLITE_DEFAULT_WAL_SYNCHRONOUS compile-time options. The SQLITE_DEFAULT_SYNCHRONOUS compile-time option replaces the SQLITE_EXTRA_DURABLE option, which is no longer supported.
- Enhanced the ".stats" command in the command-line shell to show more information about I/O performance obtained from /proc, when available.

Bug fixes:

- Make sure the sqlite3_set_auxdata() values from multiple triggers within a single statement do not interfere with one another. Ticket dc9b1c91.
- Fix the code generator for expressions of the form "x IN (SELECT...)" where the SELECT statement on the RHS is a correlated subquery. Ticket 5e3c886796e5512e.
- Fix a harmless TSAN warning associated with the sqlite3_db_readonly() interface.

Hashes:

- SQLITE_SOURCE_ID: "2016-03-29 10:14:15 e9bb4cf40f4971974a74468ef922bdee481c988b"
- SHA1 for sqlite3.c: cba2be96d27cb51978cd4a200397a4ad178986eb

2016-03-03 (3.11.1)

- Improvements to the Makefiles and build scripts used by VisualStudio.
- Fix an FTS5 issue in which the 'optimize' command could cause index corruption.
- Fix a buffer overread that might occur if FTS5 is used to query a corrupt database file.
- Increase the maximum "scope" value for the spellfix1 extension from 6 to 30.
- SQLITE_SOURCE_ID: "2016-03-03 16:17:53 f047920ce16971e573bc6ec9a48b118c9de2b3a7"
- SHA1 for sqlite3.c: 3da832fd2af36eaedb05d61a8f4c2bb9f3d54265

2016-02-15 (3.11.0)

General improvements:

- Enhanced WAL mode so that it works efficiently with transactions that are larger than the cache_size.
- Added the FTS5 detail option.
- Added the "EXTRA" option to PRAGMA synchronous that does a sync of the containing directory when a rollback journal is unlinked in DELETE mode, for better durability. The SQLITE_EXTRA_DURABLE compile-time option enables PRAGMA synchronous=EXTRA by default.
- Enhanced the query planner so that it is able to use a covering index as part of the OR optimization.
- Avoid recomputing NOT NULL and CHECK constraints on unchanged columns in UPDATE statement.
- Many micro-optimizations, resulting in a library that is faster than the previous release.

Enhancements to the command-line shell:

- By default, the shell is now in "auto-explain" mode. The output of EXPLAIN commands is automatically formatted.
- Added the ".vfslist" dot-command.
- The SQLITE_ENABLE_EXPLAIN_COMMENTS compile-time option is now turned on by default in the standard builds.

Enhancements to the TCL Interface:

- If a database connection is opened with the "-uri 1" option, then URI filenames are honored by the "backup" and "restore" commands.
- Added the "-sourceid" option to the "sqlite3" command.

Makefile improvements:

- Improved pthreads detection in configure scripts.
- Add the ability to do MSVC Windows builds from the amalgamation tarball.

Bug fixes

- Fix an issue with incorrect sharing of VDBE temporary registers between co-routines that could cause incorrect query results in obscure cases. Ticket d06a25c84454a.
- Fix a problem in the sqlite3_result_subtype() interface that could cause problems for the json1 extension under obscure circumstances. Fix for ticket f45ac567eaa9f9.
- Escape control characters in JSON strings. Fix for ticket ad2559db380abf8.

- Reenable the xCurrentTime and xGetLastError methods in the built-in unix VFSes as long as SQLITE_OMIT_DEPRECATED is not defined.

  Backwards Compatibility:

- Because of continuing security concerns, the two-argument version of of the seldom-used and little-known fts3_tokenizer() function is disabled unless SQLite is compiled with the SQLITE_ENABLE_FTS3_TOKENIZER.

  Hashes:

- SQLITE_SOURCE_ID: "2016-02-15 17:29:24 3d862f207e3adc00f78066799ac5a8c282430a5f"
- SHA1 for sqlite3.c: df01436c5fcfe72d1a95bc172158219796e1a90b

2016-01-20 (3.10.2)

Critical bug fix:

- Version 3.10.0 introduced a case-folding bug in the LIKE operator which is fixed by this patch release. Ticket 80369eddd5c94.

  Other miscellaneous bug fixes:

- Fix a use-after-free that can occur when SQLite is compiled with -DSQLITE_HAS_CODEC.
- Fix the build so that it works with -DSQLITE_OMIT_WAL.
- Fix the configure script for the amalgamation so that the --readline option works again on Raspberry PIs.

  Hashes:

- SQLITE_SOURCE_ID: "2016-01-20 15:27:19 17efb4209f97fb4971656086b138599a91a75ff9"
- SHA1 for sqlite3.c: f7088b19d97cd7a1c805ee95c696abd54f01de4f

2016-01-14 (3.10.1)

New feature:

- Add the SQLITE_FCNTL_JOURNAL_POINTER file control.

  Bug fix:

- Fix a 16-month-old bug in the query planner that could generate incorrect results when a scalar subquery attempts to use the block sorting optimization. Ticket cb3aa0641d9a4.

  Hashes:

- SQLITE_SOURCE_ID: "2016-01-13 21:41:56 254419c36766225ca542ae873ed38255e3fb8588"
- SHA1 for sqlite3.c: 1398ba8e4043550a533cdd0834bfdad1c9eab0f4

2016-01-06 (3.10.0)

General improvements:

- Added support for LIKE, GLOB, and REGEXP operators on virtual tables.
- Added the colUsed field to sqlite3_index_info for use by the sqlite3_module.xBestIndex method.
- Enhance the PRAGMA cache_spill statement to accept a 32-bit integer parameter which is the threshold below which cache spilling is prohibited.
- On unix, if a symlink to a database file is opened, then the corresponding journal files are based on the actual filename, not the symlink name.
- Added the "--transaction" option to sqldiff.
- Added the sqlite3_db_cacheflush() interface.
- Added the sqlite3_strlike() interface.
- When using memory-mapped I/O map the database file read-only so that stray pointers and/or array overruns in the application cannot accidently modify the database file.
- Added the *experimental* sqlite3_snapshot_get(), sqlite3_snapshot_open(), and sqlite3_snapshot_free() interfaces. These are subject to change or removal in a subsequent release.
- Enhance the 'utc' modifier in the date and time functions so that it is a no-op if the date/time is known to already be in UTC. (This is not a compatibility break since the behavior has long been documented as "undefined" in that case.)
- Added the json_group_array() and json_group_object() SQL functions in the json extension.
- Added the SQLITE_LIKE_DOESNT_MATCH_BLOBS compile-time option.
- Many small performance optimizations.

  Portability enhancements:

- Work around a sign-exension bug in the optimizer of the HP C compiler on HP/UX. (details)

  Enhancements to the command-line shell:

- Added the ".changes ON|OFF" and ".vfsinfo" dot-commands.
- Translate between MBCS and UTF8 when running in cmd.exe on Windows.

Enhancements to makefiles:

- Added the --enable-editline and --enable-static-shell options to the various autoconf-generated configure scripts.
- Omit all use of "awk" in the makefiles, to make building easier for MSVC users.

Important fixes:

- Fix inconsistent integer to floating-point comparison operations that could result in a corrupt index if the index is created on a table column that contains both large integers and floating point values of similar magnitude. Ticket 38a97a87a6.
- Fix an infinite-loop in the query planner that could occur on malformed common table expressions.
- Various bug fixes in the sqldiff tool.

Hashes:

- SQLITE_SOURCE_ID: "2016-01-06 11:01:07 fd0a50f0797d154fefff724624f00548b5320566"
- SHA1 for sqlite3.c: b92ca988ebb6df02ac0c8f866dbf3256740408ac

2015-11-02 (3.9.2)

- Fix the schema parser so that it interprets certain (obscure and ill-formed) CREATE TABLE statements the same as legacy. Fix for ticket ac661962a2aeab3c331
- Fix a query planner problem that could result in an incorrect answer due to the use of automatic indexing in subqueries in the FROM clause of a correlated scalar subqueries. Fix for ticket 8a2adec1.
- SQLITE_SOURCE_ID: "2015-11-02 18:31:45 bda77dda9697c463c3d0704014d51627fceee328"
- SHA1 for sqlite3.c: 1c4013876f50bbaa3e6f0f98e0147c76287684c1

2015-10-16 (3.9.1)

- Fix the json1 extension so that it does not recognize ASCII form-feed as a whitespace character, in order to comply with RFC-7159. Fix for ticket 57eec374ae1d0a1d
- Add a few #ifdef and build script changes to address compilation issues that appeared after the 3.9.0 release.

- SQLITE_SOURCE_ID: ""2015-10-16 17:31:12 767c1727fec4ce11b83f25b3f1bfcfe68a2c8b02"
- SHA1 for sqlite3.c: 5e6d1873a32d82c2cf8581f143649940cac8ae49

2015-10-14 (3.9.0)

Policy Changes:

- The version numbering conventions for SQLite are revised to use the emerging standard of semantic versioning.

  New Features And Enhancements:

- Added the json1 extension module in the source tree, and in the amalgamation. Enable support using the SQLITE_ENABLE_JSON1 compile-time option.

- Added Full Text Search version 5 (FTS5) to the amalgamation, enabled using SQLITE_ENABLE_FTS5. FTS5 will be considered "experimental" (subject to incompatible changes) for at least one more release cycle.
- The CREATE VIEW statement now accepts an optional list of column names following the view name.
- Added support for indexes on expressions.
- Added support for table-valued functions in the FROM clause of a SELECT statement.
- Added support for eponymous virtual tables.
- A VIEW may now reference undefined tables and functions when initially created. Missing tables and functions are reported when the VIEW is used in a query.
- Added the sqlite3_value_subtype() and sqlite3_result_subtype() interfaced (used by the json1 extension).
- The query planner is now able to use partial indexes that contain AND-connected terms in the WHERE clause.
- The sqlite3_analyzer.exe utility is updated to report the depth of each btree and to show the average fanout for indexes and WITHOUT ROWID tables.
- Enhanced the dbstat virtual table so that it can be used as a table-valued function where the argument is the schema to be analyzed.

  Other changes:

- The sqlite3_memory_alarm() interface, which has been deprecated and undocumented for 8 years, is changed into a no-op.

  Important fixes:

- Fixed a critical bug in the SQLite Encryption Extension that could cause the database to become unreadable and unrecoverable if a VACUUM command changed the size of the encryption nonce.

- Added a memory barrier in the implementation of sqlite3_initialize() to help ensure that it is thread-safe.
- Fix the OR optimization so that it always ignores subplans that do not use an index.
- Do not apply the WHERE-clause pushdown optimization on terms that originate in the ON or USING clause of a LEFT JOIN. Fix for ticket c2a19d81652f40568c.
- SQLITE_SOURCE_ID: "2015-10-14 12:29:53 a721fc0d89495518fe5612e2e3bbc60befd2e90d"
- SHA1 for sqlite3.c: c03e47e152ddb9c342b84ffb39448bf4a2bd4288

2015-07-29 (3.8.11.1)

- Restore an undocumented side-effect of PRAGMA cache_size: force the database schema to be parsed if the database has not been previously accessed.
- Fix a long-standing problem in sqlite3_changes() for WITHOUT ROWID tables that was reported a few hours after the 3.8.11 release.
- SQLITE_SOURCE_ID: "2015-07-29 20:00:57 cf538e2783e468bbc25e7cb2a9ee64d3e0e80b2f"
- SHA1 for sqlite3.c: 3be71d99121fe5b17f057011025bcf84e7cc6c84

2015-07-27 (3.8.11)

- Added the experimental RBU extension. Note that this extension is experimental and subject to change in incompatible ways.
- Added the experimental FTS5 extension. Note that this extension is experimental and subject to change in incompatible ways.
- Added the sqlite3_value_dup() and sqlite3_value_free() interfaces.
- Enhance the spellfix1 extension to support ON CONFLICT clauses.
- The IS operator is now able to drive indexes.
- Enhance the query planner to permit automatic indexing on FROM-clause subqueries that are implemented by co-routine.
- Disallow the use of "rowid" in common table expressions.
- Added the PRAGMA cell_size_check command for better and earlier detection of database file corruption.
- Added the matchinfo 'b' flag to the matchinfo() function in FTS3.
- Improved fuzz-testing of database files, with fixes for problems found.
- Add the fuzzcheck test program and automatically run this program using both SQL and database test cases on "make test".

- Added the SQLITE_MUTEX_STATIC_VFS1 static mutex and use it in the Windows VFS.
- The sqlite3_profile() callback is invoked (by sqlite3_reset() or sqlite3_finalize()) for statements that did not run to completion.
- Enhance the page cache so that it can preallocate a block of memory to use for the initial set page cache lines. Set the default preallocation to 100 pages. Yields about a 5% performance increase on common workloads.
- Miscellaneous micro-optimizations result in 22.3% more work for the same number of CPU cycles relative to the previous release. SQLite now runs twice as fast as version 3.8.0 and three times as fast as version 3.3.9. (Measured using cachegrind on the speedtest1.c workload on Ubuntu 14.04 x64 with gcc 4.8.2 and -Os. Your performance may vary.)
- Added the sqlite3_result_zeroblob64() and sqlite3_bind_zeroblob64() interfaces.

Important bug fixes:

- Fix CREATE TABLE AS so that columns of type TEXT never end up holding an INT value. Ticket f2ad7de056ab1dc9200

- Fix CREATE TABLE AS so that it does not leave NULL entries in the sqlite_master table if the SELECT statement on the right-hand side aborts with an error. Ticket 873cae2b6e25b
- Fix the skip-scan optimization so that it works correctly when the OR optimization is used on WITHOUT ROWID tables. Ticket 8fd39115d8f46
- Fix the sqlite3_memory_used() and sqlite3_memory_highwater() interfaces so that they actually do provide a 64-bit answer.

Hashes:

- SQLITE_SOURCE_ID: "2015-07-27 13:49:41 b8e92227a469de677a66da62e4361f099c0b79d0"

- SHA1 for sqlite3.c: 719f6891abcd9c459b5460b191d731cd12a3643e

2015-05-20 (3.8.10.2)

- Fix an index corruption issue introduced by version 3.8.7. An index with a TEXT key can be corrupted by an INSERT into the corresponding table if the table has two nested triggers that convert the key value to INTEGER and back to TEXT again. Ticket 34cd55d68e0
- SQLITE_SOURCE_ID: "2015-05-20 18:17:19 2ef4f3a5b1d1d0c4338f8243d40a2452cc1f7fe4"
- SHA1 for sqlite3.c: 638abb77965332c956dbbd2c8e4248e84da4eb63

2015-05-09 (3.8.10.1)

- Make sqlite3_compileoption_used() responsive to the SQLITE_ENABLE_DBSTAT_VTAB compile-time option.
- Fix a harmless warning in the command-line shell on some versions of MSVC.
- Fix minor issues with the dbstat virtual table.
- SQLITE_SOURCE_ID: "2015-05-09 12:14:55 05b4b1f2a937c06c90db70c09890038f6c98ec40"
- SHA1 for sqlite3.c: 85e4e1c08c7df28ef61bb9759a0d466e0eefbaa2

2015-05-07 (3.8.10)

- Added the sqldiff.exe utility program for computing the differences between two SQLite database files.
- Added the matchinfo y flag to the matchinfo() function of FTS3.
- Performance improvements for ORDER BY, VACUUM, CREATE INDEX, PRAGMA integrity_check, and PRAGMA quick_check.
- Fix many obscure problems discovered while SQL fuzzing.
- Identify all methods for important objects in the interface documentation. (example)
- Made the American Fuzzy Lop fuzzer a standard part of SQLite's testing strategy.
- Add the ".binary" and ".limits" commands to the command-line shell.
- Make the dbstat virtual table part of standard builds when compiled with the SQLITE_ENABLE_DBSTAT_VTAB option.
- SQLITE_SOURCE_ID: "2015-05-07 11:53:08 cf975957b9ae671f34bb65f049acf351e650d437"
- SHA1 for sqlite3.c: 0b34f0de356a3f21b9dfc761f3b7821b6353c570

2015-04-08 (3.8.9)

- Add VxWorks-7 as an officially supported and tested platform.
- Added the sqlite3_status64() interface.
- Fix memory size tracking so that it works even if SQLite uses more than 2GiB of memory.
- Added the PRAGMA index_xinfo command.
- Fix a potential 32-bit integer overflow problem in the sqlite3_blob_read() and sqlite3_blob_write() interfaces.
- Ensure that prepared statements automatically reset on extended error codes of SQLITE_BUSY and SQLITE_LOCKED even when compiled using SQLITE_OMIT_AUTORESET.
- Correct miscounts in the sqlite3_analyzer.exe utility related to WITHOUT ROWID tables.

- Added the ".dbinfo" command to the command-line shell.
- Improve the performance of fts3/4 queries that use the OR operator and at least one auxiliary fts function.
- Fix a bug in the fts3 snippet() function causing it to omit leading separator characters from snippets that begin with the first token in a column.
- SQLITE_SOURCE_ID: "2015-04-08 12:16:33 8a8ffc862e96f57aa698f93de10dee28e69f6e09"
- SHA1 for sqlite3.c: 49f1c3ae347e1327b5aaa6c7f76126bdf09c6f42

2015-02-25 (3.8.8.3)

- Fix a bug (ticket 2326c258d02ead33) that can lead to incorrect results if the qualifying constraint of a partial index appears in the ON clause of a LEFT JOIN.
- Added the ability to link against the "linenoise" command-line editing library in unix builds of the command-line shell.
- SQLITE_SOURCE_ID: "2015-02-25 13:29:11 9d6c1880fb75660bbabd693175579529785f8a6b"
- SHA1 for sqlite3.c: 74ee38c8c6fd175ec85a47276dfcefe8a262827a

2015-01-30 (3.8.8.2)

- Enhance sqlite3_wal_checkpoint_v2(TRUNCATE) interface so that it truncates the WAL file even if there is no checkpoint work to be done.
- SQLITE_SOURCE_ID: "2015-01-30 14:30:45 7757fc721220e136620a89c9d28247f28bbbc098"
- SHA1 for sqlite3.c: 85ce79948116aa9a087ec345c9d2ce2c1d3cd8af

2015-01-20 (3.8.8.1)

- Fix a bug in the sorting logic, present since version 3.8.4, that can cause output to appear in the wrong order on queries that contains an ORDER BY clause, a LIMIT clause, and that have approximately 60 or more columns in the result set. Ticket f97c4637102a3ae72b79.
- SQLITE_SOURCE_ID: "2015-01-20 16:51:25 f73337e3e289915a76ca96e7a05a1a8d4e890d55"
- SHA1 for sqlite3.c: 33987fb50dcc09f1429a653d6b47672f5a96f19e

2015-01-16 (3.8.8)

New Features:

- Added the PRAGMA data_version command that can be used to determine if a database file has been modified by another process.
- Added the SQLITE_CHECKPOINT_TRUNCATE option to the sqlite3_wal_checkpoint_v2() interface, with corresponding enhancements to PRAGMA wal_checkpoint.
- Added the sqlite3_stmt_scanstatus() interface, available only when compiled with SQLITE_ENABLE_STMT_SCANSTATUS.
- The sqlite3_table_column_metadata() is enhanced to work correctly on WITHOUT ROWID tables and to check for the existence of a a table if the column name parameter is NULL. The interface is now also included in the build by default, without requiring the SQLITE_ENABLE_COLUMN_METADATA compile-time option.
- Added the SQLITE_ENABLE_API_ARMOR compile-time option.
- Added the SQLITE_REVERSE_UNORDERED_SELECTS compile-time option.
- Added the SQLITE_SORTER_PMASZ compile-time option and SQLITE_CONFIG_PMASZ start-time option.
- Added the SQLITE_CONFIG_PCACHE_HDRSZ option to sqlite3_config() which makes it easier for applications to determine the appropriate amount of memory for use with SQLITE_CONFIG_PAGECACHE.
- The number of rows in a VALUES clause is no longer limited by SQLITE_LIMIT_COMPOUND_SELECT.
- Added the eval.c loadable extension that implements an eval() SQL function that will recursively evaluate SQL.

Performance Enhancements:

- Reduce the number of memcpy() operations involved in balancing a b-tree, for 3.2% overall performance boost.

- Improvements to cost estimates for the skip-scan optimization.
- The automatic indexing optimization is now capable of generating a partial index if that is appropriate.

Bug fixes:

- Ensure durability following a power loss with "PRAGMA journal_mode=TRUNCATE" by calling fsync() right after truncating the journal file.

- The query planner now recognizes that any column in the right-hand table of a LEFT JOIN can be NULL, even if that column has a NOT NULL constraint. Avoid trying to optimize out NULL tests in those cases. Fix for ticket 6f2222d550f5b0ee7ed.
- Make sure ORDER BY puts rows in ascending order even if the DISTINCT operator is implemented using a descending index. Fix for ticket c5ea805691bfc4204b1cb9e.
- Fix data races that might occur under stress when running with many threads in shared

cache mode where some of the threads are opening and closing connections.

- Fix obscure crash bugs found by american fuzzy lop. Ticket a59ae93ee990a55.
- Work around a GCC optimizer bug (for gcc 4.2.1 on MacOS 10.7) that caused the R-Tree extension to compute incorrect results when compiled with -O3.

Other changes:

- Disable the use of the strchrnul() C-library routine unless it is specifically enabled using the -DHAVE_STRCHRNULL compile-time option.

- Improvements to the effectiveness and accuracy of the likelihood(), likely(), and unlikely() SQL hint functions.
- SQLITE_SOURCE_ID: "2015-01-16 12:08:06 7d68a42face3ab14ed88407d4331872f5b243fdf"
- SHA1 for sqlite3.c: 91aea4cc722371d58aae3d22e94d2a4165276905

2014-12-09 (3.8.7.4)

- Bug fix: Add in a mutex that was omitted from the previous release.
- SQLITE_SOURCE_ID: "2014-12-09 01:34:36 f66f7a17b78ba617acde90fc810107f34f1a1f2e"
- SHA1 for sqlite3.c: 0a56693a3c24aa3217098afab1b6fecccdedfd23

2014-12-05 (3.8.7.3)

- Bug fix: Ensure the cached KeyInfo objects (an internal abstraction not visible to the application) do not go stale when operating in shared cache mode and frequently closing and reopening some database connections while leaving other database connections on the same shared cache open continuously. Ticket e4a18565a36884b00edf.
- Bug fix: Recognize that any column in the right-hand table of a LEFT JOIN can be NULL even if the column has a NOT NULL constraint. Do not apply optimizations that assume the column is never NULL. Ticket 6f2222d550f5b0ee7ed.
- SQLITE_SOURCE_ID: "2014-12-05 22:29:24 647e77e853e81a5effeb4c33477910400a67ba86"
- SHA1 for sqlite3.c: 3ad2f5ba3a4a3e3e51a1dac9fda9224b359f0261

2014-11-18 (3.8.7.2)

- Enhance the ROLLBACK command so that pending queries are allowed to continue as long as the schema is unchanged. Formerly, a ROLLBACK would cause all pending queries to fail with an SQLITE_ABORT or SQLITE_ABORT_ROLLBACK error. That error is still returned if the ROLLBACK modifies the schema.

- Bug fix: Make sure that NULL results from OP_Column are fully and completely NULL and do not have the MEM_Ephem bit set. Ticket 094d39a4c95ee4.
- Bug fix: The %c format in sqlite3_mprintf() is able to handle precisions greater than 70.
- Bug fix: Do not automatically remove the DISTINCT keyword from a SELECT that forms the right-hand side of an IN operator since it is necessary if the SELECT also contains a LIMIT. Ticket db87229497.
- SQLITE_SOURCE_ID: "2014-11-18 20:57:56 2ab564bf9655b7c7b97ab85cafc8a48329b27f93"
- SHA1 for sqlite3.c: b2a68d5783f48dba6a8cb50d8bf69b238c5ec53a

2014-10-29 (3.8.7.1)

- In PRAGMA journal_mode=TRUNCATE mode, call fsync() immediately after truncating the journal file to ensure that the transaction is durable across a power loss.
- Fix an assertion fault that can occur when updating the NULL value of a field at the end of a table that was added using ALTER TABLE ADD COLUMN.
- Do not attempt to use the strchrnul() function from the standard C library unless the HAVE_STRCHRNULL compile-time option is set.
- Fix a couple of problems associated with running an UPDATE or DELETE on a VIEW with a rowid in the WHERE clause.
- SQLITE_SOURCE_ID: "2014-10-29 13:59:56 3b7b72c4685aa5cf5e675c2c47ebec10d9704221"
- SHA1 for sqlite3.c: 2d25bd1a73dc40f538f3a81c28e6efa5999bdf0c

2014-10-17 (3.8.7)

Performance Enhancements:

- Many micro-optimizations result in 20.3% more work for the same number of CPU cycles relative to the previous release. The cumulative performance increase since version 3.8.0 is 61%. (Measured using cachegrind on the speedtest1.c workload on Ubuntu 13.10 x64 with gcc 4.8.1 and -Os. Your performance may vary.)
- The sorter can use auxiliary helper threads to increase real-time response. This feature is off by default and may be enabled using the PRAGMA threads command or the SQLITE_DEFAULT_WORKER_THREADS compile-time option.
- Enhance the skip-scan optimization so that it is able to skip index terms that occur in the middle of the index, not just as the left-hand side of the index.
- Improved optimization of CAST operators.
- Various improvements in how the query planner uses sqlite_stat4 information to estimate plan costs.

New Features:

- Added new interfaces with 64-bit length parameters: sqlite3_malloc64(), sqlite3_realloc64(), sqlite3_bind_blob64(), sqlite3_result_blob64(), sqlite3_bind_text64(), and sqlite3_result_text64().
- Added the new interface sqlite3_msize() that returns the size of a memory allocation obtained from sqlite3_malloc64() and its variants.
- Added the SQLITE_LIMIT_WORKER_THREADS option to sqlite3_limit() and PRAGMA threads command for configuring the number of available worker threads.
- The spellfix1 extension allows the application to optionally specify the rowid for each INSERT.
- Added the User Authentication extension.

Bug Fixes:

- Fix a bug in the partial index implementation that might result in an incorrect answer if a partial index is used in a subquery or in a view. Ticket 98d973b8f5.
- Fix a query planner bug that might cause a table to be scanned in the wrong direction (thus reversing the order of output) when a DESC index is used to implement the ORDER BY clause on a query that has an identical GROUP BY clause. Ticket ba7cbfaedc7e6.
- Fix a bug in sqlite3_trace() that was causing it to sometimes fail to print an SQL statement if that statement needed to be re-prepared. Ticket 11d5aa455e0d98f3c1e6a08
- Fix a faulty assert() statement. Ticket 369d57fb8e5ccdff06f1

Test, Debug, and Analysis Changes:

- Show ASCII-art abstract syntax tree diagrams using the ".selecttrace" and ".wheretrace" commands in the command-line shell when compiled with SQLITE_DEBUG, SQLITE_ENABLE_SELECTTRACE, and SQLITE_ENABLE_WHERETRACE. Also provide the sqlite3TreeViewExpr() and sqlite3TreeViewSelect() entry points that can be invoked from with the debugger to show the parse tree when stopped at a breakpoint.
- Drop support for SQLITE_ENABLE_TREE_EXPLAIN. The SELECTTRACE mechanism provides more useful diagnostics information.
- New options to the command-line shell for configuring auxiliary memory usage: --pagecache, --lookaside, and --scratch.
- SQLITE_SOURCE_ID: "2014-10-17 11:24:17 e4ab094f8afce0817f4074e823fabe59fc29ebb4"
- SHA1 for sqlite3.c: 56dcf5e931a9e1fa12fc2d600cd91d3bf9b639cd

2014-08-15 (3.8.6)

- Added support for hexadecimal integer literals in the SQL parser. (Ex: 0x123abc)
- Enhanced the PRAGMA integrity_check command to detect UNIQUE and NOT NULL constraint violations.
- Increase the maximum value of SQLITE_MAX_ATTACHED from 62 to 125.
- Increase the timeout in WAL mode before issuing an SQLITE_PROTOCOL error from 1 second to 10 seconds.
- Added the likely(X) SQL function.
- The unicode61 tokenizer is now included in FTS4 by default.
- Trigger automatic reprepares on all prepared statements when ANALYZE is run.
- Added a new loadable extension source code file to the source tree: fileio.c
- Add extension functions readfile(X) and writefile(X,Y) (using code copy/pasted from fileio.c in the previous bullet) to the command-line shell.
- Added the .fullschema dot-command to the command-line shell.

Performance Enhancements:

- Deactivate the DISTINCT keyword on subqueries on the right-hand side of the IN operator.

- Add the capability of evaluating an IN operator as a sequence of comparisons as an alternative to using a table lookup. Use the sequence of comparisons implementation in circumstances where it is likely to be faster, such as when the right-hand side of the IN operator is small and/or changes frequently.
- The query planner now uses sqlite_stat4 information (created by ANALYZE) to help determine if the skip-scan optimization is appropriate.
- Ensure that the query planner never tries to use a self-made transient index in place of a schema-defined index.
- Other minor tweaks to improve the quality of VDBE code.

Bug Fixes:

- Fix a bug in CREATE UNIQUE INDEX, introduced when WITHOUT ROWID support added in version 3.8.2, that allows a non-unique NOT NULL column to be given a UNIQUE index. Ticket 9a6daf340df99ba93c

- Fix a bug in R-Tree extension, introduced in the previous release, that can cause an incorrect results for queries that use the rowid of the R-Tree on the left-hand side of an IN operator. Ticket d2889096e7bdeac6.
- Fix the sqlite3_stmt_busy() interface so that it gives the correct answer for ROLLBACK statements that have been stepped but never reset.
- Fix a bug in that would cause a null pointer to be dereferenced if a column with a

DEFAULT that is an aggregate function tried to usee its DEFAULT. Ticket 3a88d85f36704eebe1

- CSV output from the command-line shell now always uses CRNL for the row separator and avoids inserting CR in front of NLs contained in data.
- Fix a column affinity problem with the IN operator. Ticket 9a8b09f8e6.
- Fix the ANALYZE command so that it adds correct samples for WITHOUT ROWID tables in the sqlite_stat4 table. Ticket b2fa5424e6fcb15.
- SQLITE_SOURCE_ID: "2014-08-15 11:46:33 9491ba7d738528f168657adb43a198238abde19e"
- SHA1 for sqlite3.c: 72c64f05cd9babb9c0f9b3c82536d83be7804b1c

2014-06-04 (3.8.5)

- Added support for partial sorting by index.
- Enhance the query planner so that it always prefers an index that uses a superset of WHERE clause terms relative to some other index.
- Improvements to the automerge command of FTS4 to better control the index size for a full-text index that is subject to a large number of updates.
- Added the sqlite3_rtree_query_callback() interface to R-Tree extension
- Added new URI query parameters "nolock" and "immutable".
- Use less memory by not remembering CHECK constraints on read-only database connections.
- Enable the OR optimization for WITHOUT ROWID tables.
- Render expressions of the form "x IN (?)" (with a single value in the list on the right-hand side of the IN operator) as if they where "x==?", Similarly optimize "x NOT IN (?)"
- Add the ".system" and ".once" commands to the command-line shell.
- Added the SQLITE_IOCAP_IMMUTABLE bit to the set of bits that can be returned by the xDeviceCharacteristics method of a VFS.
- Added the SQLITE_TESTCTRL_BYTEORDER test control.

Bug Fixes:

- OFFSET clause ignored on queries without a FROM clause. Ticket 07d6a0453d
- Assertion fault on queries involving expressions of the form "x IN (?)". Ticket e39d032577.
- Incorrect column datatype reported. Ticket a8a0d2996a
- Duplicate row returned on a query against a table with more than 16 indices, each on a separate column, and all used via OR-connected constraints. Ticket 10fb063b11
- Partial index causes assertion fault on UPDATE OR REPLACE. Ticket 2ea3e9fe63
- Crash when calling undocumented SQL function sqlite_rename_parent() with NULL parameters. Ticket 264b970c43

- ORDER BY ignored if the query has an identical GROUP BY. Ticket b75a9ca6b0
- The group_concat(x,'') SQL function returns NULL instead of an empty string when all inputs are empty strings. Ticket 55746f9e65
- Fix a bug in the VDBE code generator that caused crashes when doing an INSERT INTO ... SELECT statement where the number of columns being inserted is larger than the number of columns in the destination table. Ticket e9654505cfd
- Fix a problem in CSV import in the command-line shell where if the leftmost field of the first row in the CSV file was both zero bytes in size and unquoted no data would be imported.
- Fix a problem in FTS4 where the left-most column that contained the notindexed column name as a prefix was not indexed rather than the column whose name matched exactly.
- Fix the sqlite3_db_readonly() interface so that it returns true if the database is read-only due to the file format write version number being too large.
- SQLITE_SOURCE_ID: "2014-06-04 14:06:34 b1ed4f2a34ba66c29b130f8d13e9092758019212"
- SHA1 for sqlite3.c: 7bc194957238c61b1a47f301270286be5bc5208c

2014-04-03 (3.8.4.3)

- Add a one-character fix for a problem that might cause incorrect query results on a query that mixes DISTINCT, GROUP BY in a subquery, and ORDER BY. Ticket 98825a79ce14.
- SQLITE_SOURCE_ID: "2014-04-03 16:53:12 a611fa96c4a848614efe899130359c9f6fb889c3"
- SHA1 for sqlite3.c: 310a1faeb9332a3cd8d1f53b4a2e055abf537bdc

2014-03-26 (3.8.4.2)

- Fix a potential buffer overread that could result when trying to search a corrupt database file.
- SQLITE_SOURCE_ID: "2014-03-26 18:51:19 02ea166372bdb2ef9d8dfbb05e78a97609673a8e"
- SHA1 for sqlite3.c: 4685ca86c2ea0649ed9f59a500013e90b3fe6d03

2014-03-11 (3.8.4.1)

- Work around a C-preprocessor macro conflict that breaks the build for some configurations with Microsoft Visual Studio.
- When computing the cost of the skip-scan optimization, take into account the fact that

multiple seeks are required.
- SQLITE_SOURCE_ID: "2014-03-11 15:27:36 018d317b1257ce68a92908b05c9c7cf1494050d0"
- SHA1 for sqlite3.c: d5cd1535053a50aa8633725e3595740b33709ac5

2014-03-10 (3.8.4)

- Code optimization and refactoring for improved performance.
- Add the ".clone" and ".save" commands to the command-line shell.
- Update the banner on the command-line shell to alert novice users when they are using an ephemeral in-memory database.
- Fix editline support in the command-line shell.
- Add support for coverage testing of VDBE programs using the SQLITE_TESTCTRL_VDBE_COVERAGE verb of sqlite3_test_control().
- Update the _FILE_OFFSET_BITS macro so that builds work again on QNX.
- Change the datatype of SrcList.nSrc from type u8 to type int to work around an issue in the C compiler on AIX.
- Get extension loading working on Cygwin.
- Bug fix: Fix the char() SQL function so that it returns an empty string rather than an "out of memory" error when called with zero arguments.
- Bug fix: DISTINCT now recognizes that a zeroblob and a blob of all 0x00 bytes are the same thing. Ticket [fccbde530a]
- Bug fix: Compute the correct answer for queries that contain an IS NOT NULL term in the WHERE clause and also contain an OR term in the WHERE clause and are compiled with SQLITE_ENABLE_STAT4. Ticket [4c86b126f2]
- Bug fix: Make sure "rowid" columns are correctly resolved in joins between normal tables and WITHOUT ROWID tables. Ticket [c34d0557f7]
- Bug fix: Make sure the same temporary registers are not used in concurrent co-routines used to implement compound SELECT statements containing ORDER BY clauses, as such use can lead to incorrect answers. Ticket [8c63ff0eca]
- Bug fix: Ensure that "ORDER BY random()" clauses do not get optimized out. Ticket [65bdeb9739]
- Bug fix: Repair a name-resolution error that can occur in sub-select statements contained within a TRIGGER. Ticket [4ef7e3cfca]
- Bug fix: Fix column default values expressions of the form "DEFAULT(-(-9223372036854775808))" so that they work correctly, initializing the column to a floating point value approximately equal to +9223372036854775808.0.
- SQLITE_SOURCE_ID: "2014-03-10 12:20:37 530a1ee7dc2435f80960ce4710a3c2d2bfaaccc5"
- SHA1 for sqlite3.c: b0c22e5f15f5ba2afd017ecd990ea507918afe1c

2014-02-11 (3.8.3.1)

- Fix a bug (ticket 4c86b126f2) that causes rows to go missing on some queries with OR clauses and IS NOT NULL operators in the WHERE clause, when the SQLITE_ENABLE_STAT3 or SQLITE_ENABLE_STAT4 compile-time options are used.
- Fix a harmless compiler warning that was causing problems for VS2013.
- SQLITE_SOURCE_ID: "2014-02-11 14:52:19 ea3317a4803d71d88183b29f1d3086f46d68a00e"
- SHA1 for sqlite3.c: 990004ef2d0eec6a339e4caa562423897fe02bf0

2014-02-03 (3.8.3)

- Added support for common table expressions and the WITH clause.
- Added the printf() SQL function.
- Added SQLITE_DETERMINISTIC as an optional bit in the 4th argument to the sqlite3_create_function() and related interfaces, providing applications with the ability to create new functions that can be factored out of inner loops when they have constant arguments.
- Add SQLITE_READONLY_DBMOVED error code, returned at the beginning of a transaction, to indicate that the underlying database file has been renamed or moved out from under SQLite.
- Allow arbitrary expressions, including function calls and subqueries, in the filename argument to ATTACH.
- Allow a VALUES clause to be used anywhere a SELECT statement is valid.
- Reseed the PRNG used by sqlite3_randomness(N,P) when invoked with N==0. Automatically reseed after a fork() on unix.
- Enhance the spellfix1 virtual table so that it can search efficiently by rowid.
- Performance enhancements.
- Improvements to the comments in the VDBE byte-code display when running EXPLAIN.
- Add the "%token_class" directive to LEMON parser generator and use it to simplify the grammar.
- Change the LEMON source code to avoid calling C-library functions that OpenBSD considers dangerous. (Ex: sprintf).
- Bug fix: In the command-line shell CSV import feature, do not end a field when an escaped double-quote occurs at the end of a CRLN line.
- SQLITE_SOURCE_ID: "2014-02-03 13:52:03 e816dd924619db5f766de6df74ea2194f3e3b538"
- SHA1 for sqlite3.c: 98a07da78f71b0275e8d9c510486877adc31dbee

2013-12-06 (3.8.2)

- Changed the defined behavior for the CAST expression when floating point values greater than +9223372036854775807 are cast into into integers so that the result is the largest possible integer, +9223372036854775807, instead of the smallest possible integer, -9223372036854775808. After this change, CAST(9223372036854775809.0 as INT) yields +9223372036854775807 instead of -9223372036854775808. **<big>←</big> Potentially Incompatible Change!**
- Added support for WITHOUT ROWID tables.
- Added the skip-scan optimization to the query planner.
- Extended the virtual table interface, and in particular the sqlite3_index_info object to allow a virtual table to report its estimate on the number of rows that will be returned by a query.
- Update the R-Tree extension to make use of the enhanced virtual table interface.
- Add the SQLITE_ENABLE_EXPLAIN_COMMENTS compile-time option.
- Enhanced the comments that are inserted into EXPLAIN output when the SQLITE_ENABLE_EXPLAIN_COMMENTS compile-time option is enabled.
- Performance enhancements in the VDBE, especially to the OP_Column opcode.
- Factor constant subexpressions in inner loops out to the initialization code in prepared statements.
- Enhanced the ".explain" output formatting of the command-line shell so that loops are indented to better show the structure of the program.
- Enhanced the ".timer" feature of the command-line shell so that it shows wall-clock time in addition to system and user times.
- SQLITE_SOURCE_ID: "2013-12-06 14:53:30 27392118af4c38c5203a04b8013e1afdb1cebd0d"
- SHA1 for sqlite3.c: 6422c7d69866f5ea3db0968f67ee596e7114544e

2013-10-17 (3.8.1)

- Added the unlikely() and likelihood() SQL functions to be used as hints to the query planner.
- Enhancements to the query planner:
  - Take into account the fact WHERE clause terms that cannot be used with indices still probably reduce the number of output rows.
  - Estimate the sizes of table and index rows and use the smallest applicable B-Tree for full scans and "count(*)" operations.
- Added the soft_heap_limit pragma.
- Added support for SQLITE_ENABLE_STAT4
- Added support for "sz=NNN" parameters at the end of sqlite_stat1.stat fields used to specify the average length in bytes for table and index rows.
- Avoid running foreign-key constraint checks on an UPDATE if none of the modified

columns are associated with foreign keys.

- Added the SQLITE_MINIMUM_FILE_DESCRIPTOR compile-time option
- Added the win32-longpath VFS on windows, permitting filenames up to 32K characters in length.
- The Date And Time Functions are enhanced so that the current time (ex: julianday('now')) is always the same for multiple function invocations within the same sqlite3_step() call.
- Add the "totype.c" extension, implementing the tointeger() and toreal() SQL functions.
- FTS4 queries are better able to make use of docid<$limit constraints="" to="" limit="" the="" amount="" of="" i="" o="" required.="" <li="">Added the hidden fts4aux languageid column to the fts4aux virtual table.</li="">
- The VACUUM command packs the database about 1% tighter.
- The sqlite3_analyzer utility program is updated to provide better descriptions and to compute a more accurate estimate for "Non-sequential pages"
- Refactor the implementation of PRAGMA statements to improve parsing performance.
- The directory used to hold temporary files on unix can now be set using the SQLITE_TMPDIR environment variable, which takes precedence over the TMPDIR environment variable. The sqlite3_temp_directory global variable still has higher precedence than both environment variables, however.
- Added the PRAGMA stats statement.
- **Bug fix:** Return the correct answer for "SELECT count(*) FROM table" even if there is a partial index on the table. Ticket a5c8ed66ca.
- SQLITE_SOURCE_ID: "2013-10-17 12:57:35 c78be6d786c19073b3a6730dfe3fb1be54f5657a"
- SHA1 for sqlite3.c: 0a54d76566728c2ba96292a49b138e4f69a7c391

2013-09-03 (3.8.0.2)

- Fix a bug in the optimization that attempts to omit unused LEFT JOINs
- SQLITE_SOURCE_ID: "2013-09-03 17:11:13 7dd4968f235d6e1ca9547cda9cf3bd570e1609ef"
- SHA1 for sqlite3.c: 6cf0c7b46975a87a0dc3fba69c229a7de61b0c21

2013-08-29 (3.8.0.1)

- Fix an off-by-one error that caused quoted empty string at the end of a CRNL-terminated line of CSV input to be misread by the command-line shell.
- Fix a query planner bug involving a LEFT JOIN with a BETWEEN or LIKE/GLOB constraint and then another INNER JOIN to the right that involves an OR constraint.
- Fix a query planner bug that could result in a segfault when querying tables with a

UNIQUE or PRIMARY KEY constraint with more than four columns.
- SQLITE_SOURCE_ID: "2013-08-29 17:35:01 352362bc01660edfbda08179d60f09e2038a2f49"
- SHA1 for sqlite3.c: 99906bf63e6cef63d6f3d7f8526ac4a70e76559e

2013-08-26 (3.8.0)

- Add support for partial indexes
- Cut-over to the next generation query planner for faster and better query plans.
- The EXPLAIN QUERY PLAN output no longer shows an estimate of the number of rows generated by each loop in a join.
- Added the FTS4 notindexed option, allowing non-indexed columns in an FTS4 table.
- Added the SQLITE_STMTSTATUS_VM_STEP option to sqlite3_stmt_status().
- Added the cache_spill pragma.
- Added the query_only pragma.
- Added the defer_foreign_keys pragma and the sqlite3_db_status(db, SQLITE_DBSTATUS_DEFERRED_FKS,...) C-language interface.
- Added the "percentile()" function as a loadable extension in the ext/misc subdirectory of the source tree.
- Added the SQLITE_ALLOW_URI_AUTHORITY compile-time option.
- Add the sqlite3_cancel_auto_extension(X) interface.
- A running SELECT statement that lacks a FROM clause (or any other statement that never reads or writes from any database file) will not prevent a read transaction from closing.
- Add the SQLITE_DEFAULT_AUTOMATIC_INDEX compile-time option. Setting this option to 0 disables automatic indices by default.
- Issue an SQLITE_WARNING_AUTOINDEX warning on the SQLITE_CONFIG_LOG whenever the query planner uses an automatic index.
- Added the SQLITE_FTS3_MAX_EXPR_DEPTH compile-time option.
- Added an optional 5th parameter defining the collating sequence to the next_char() extension SQL function.
- The SQLITE_BUSY_SNAPSHOT extended error code is returned in WAL mode when a read transaction cannot be upgraded to a write transaction because the read is on an older snapshot.
- Enhancements to the sqlite3_analyzer utility program to provide size information separately for each individual index of a table, in addition to the aggregate size.
- Allow read transactions to be freely opened and closed by SQL statements run from within the implementation of application-defined SQL functions if the function is called by a SELECT statement that does not access any database table.
- Disable the use of posix_fallocate() on all (unix) systems unless the

HAVE_POSIX_FALLOCATE compile-time option is used.
- Update the ".import" command in the command-line shell to support multi-line fields and correct RFC-4180 quoting and to issue warning and/or error messages if the input text is not strictly RFC-4180 compliant.
- Bug fix: In the unicode61 tokenizer of FTS4, treat all private code points as identifier symbols.
- Bug fix: Bare identifiers in ORDER BY clauses bind more tightly to output column names, but identifiers in expressions bind more tightly to input column names. Identifiers in GROUP BY clauses always prefer output column names, however.
- Bug fixes: Multiple problems in the legacy query optimizer were fixed by the move to NGQP.
- SQLITE_SOURCE_ID: "2013-08-26 04:50:08 f64cd21e2e23ed7cff48f7dafa5e76adde9321c2"
- SHA1 for sqlite3.c: b7347f4b4c2a840e6ba12040093d606bd16ea21e

2013-05-20 (3.7.17)

- Add support for memory-mapped I/O.
- Add the sqlite3_strglob() convenience interface.
- Assigned the integer at offset 68 in the database header as the Application ID for when SQLite is used as an application file-format. Added the PRAGMA application_id command to query and set the Application ID.
- Report rollback recovery in the error log as SQLITE_NOTICE_RECOVER_ROLLBACK. Change the error log code for WAL recover from SQLITE_OK to SQLITE_NOTICE_RECOVER_WAL.
- Report the risky uses of unlinked database files and database filename aliasing as SQLITE_WARNING messages in the error log.
- Added the SQLITE_TRACE_SIZE_LIMIT compile-time option.
- Increase the default value of SQLITE_MAX_SCHEMA_RETRY to 50 and make sure that it is honored in every place that a schema change might force a statement retry.
- Add a new test harness called "mptester" used to verify correct operation when multiple processes are using the same database file at the same time.
- Enhance the extension loading mechanism to be more flexible (while still maintaining backwards compatibility) in two ways:
  1. If the default entry point "sqlite3_extension_init" is not present in the loadable extension, also try an entry point "sqlite3_X_init" where "X" is based on the shared library filename. This allows every extension to have a different entry point, which allows them to be statically linked with no code changes.
  2. The shared library filename passed to sqlite3_load_extension() may omit the filename suffix, and an appropriate architecture-dependent suffix (".so", ".dylib", or

".dll") will be added automatically.

- Added many new loadable extensions to the source tree, including amatch, closure, fuzzer, ieee754, nextchar, regexp, spellfix, and wholenumber. See header comments on each extension source file for further information about what that extension does.
- Enhance FTS3 to avoid using excess stack space when there are a huge number of terms on the right-hand side of the MATCH operator. A side-effect of this change is that the MATCH operator can only accommodate 12 NEAR operators at a time.
- Enhance the fts4aux virtual table so that it can be a TEMP table.
- Added the fts3tokenize virtual table to the full-text search logic.
- Query planner enhancement: Use the transitive property of constraints to move constraints into the outer loops of a join whenever possible, thereby reducing the amount of work that needs to occur in inner loops.
- Discontinue the use of posix_fallocate() on unix, as it does not work on all filesystems.
- Improved tracing and debugging facilities in the Windows VFS.
- Bug fix: Fix a potential **database corruption bug** in shared cache mode when one database connection is closed while another is in the middle of a write transaction. Ticket e636a050b7
- Bug fix: Only consider AS names from the result set as candidates for resolving identifiers in the WHERE clause if there are no other matches. In the ORDER BY clause, AS names take priority over any column names. Ticket 2500cdb9be05
- Bug fix: Do not allow a virtual table to cancel the ORDER BY clause unless all outer loops are guaranteed to return no more than one row result. Ticket ba82a4a41eac1.
- Bug fix: Do not suppress the ORDER BY clause on a virtual table query if an IN constraint is used. Ticket f69b96e3076e.
- Bug fix: The command-line shell gives an exit code of 0 when terminated using the ".quit" command.
- Bug fix: Make sure PRAGMA statements appear in sqlite3_trace() output.
- Bug fix: When a compound query that uses an ORDER BY clause with a COLLATE operator, make sure that the sorting occurs according to the specified collation and that the comparisons associate with the compound query use the native collation. Ticket 6709574d2a8d8.
- Bug fix: Makes sure the authorizer callback gets a valid pointer to the string "ROWID" for the column-name parameter when doing an UPDATE that changes the rowid. Ticket 0eb70d77cb05bb2272
- Bug fix: Do not move WHERE clause terms inside OR expressions that are contained within an ON clause of a LEFT JOIN. Ticket f2369304e4
- Bug fix: Make sure an error is always reported when attempting to preform an operation that requires a collating sequence that is missing. Ticket 0fc59f908b
- SQLITE_SOURCE_ID: "2013-05-20 00:56:22 118a3b35693b134d56ebd780123b7fd6f1497668"

- SHA1 for sqlite3.c: 246987605d0503c700a08b9ee99a6b5d67454aab

2013-04-12 (3.7.16.2)

- Fix a bug (present since version 3.7.13) that could result in database corruption on windows if two or more processes try to access the same database file at the same time and immediately after third process crashed in the middle of committing to that same file. See ticket 7ff3120e4f for further information.
- SQLITE_SOURCE_ID: "2013-04-12 11:52:43 cbea02d93865ce0e06789db95fd9168ebac970c7"
- SHA1 for sqlite3.c: d466b54789dff4fb0238b9232e74896deaefab94

2013-03-29 (3.7.16.1)

- Fix for a bug in the ORDER BY optimizer that was introduced in version 3.7.15 which would sometimes optimize out the sorting step when in fact the sort was required. Ticket a179fe7465
- Fix a long-standing bug in the CAST expression that would recognize UTF16 characters as digits even if their most-significant-byte was not zero. Ticket 689137afb6da41.
- Fix a bug in the NEAR operator of FTS3 when applied to subfields. Ticket 38b1ae018f.
- Fix a long-standing bug in the storage engine that would (very rarely) cause a spurious report of an SQLITE_CORRUPT error but which was otherwise harmless. Ticket 6bfb98dfc0c.
- The SQLITE_OMIT_MERGE_SORT option has been removed. The merge sorter is now a required component of SQLite.
- Fixed lots of spelling errors in the source-code comments
- SQLITE_SOURCE_ID: "2013-03-29 13:44:34 527231bc67285f01fb18d4451b28f61da3c4e39d"
- SHA1 for sqlite3.c: 7a91ceceac9bcf47ceb8219126276e5518f7ff5a

2013-03-18 (3.7.16)

- Added the PRAGMA foreign_key_check command.
- Added new extended error codes for all SQLITE_CONSTRAINT errors
- Added the SQLITE_READONLY_ROLLBACK extended error code for when a database cannot be opened because it needs rollback recovery but is read-only.
- Added SQL functions unicode(A) and char(X1,...,XN).
- Performance improvements for PRAGMA incremental_vacuum, especially in cases where the number of free pages is greater than what will fit on a single trunk page of the freelist.

- Improved optimization of queries containing aggregate min() or max().
- Enhance virtual tables so that they can potentially use an index when the WHERE clause contains the IN operator.
- Allow indices to be used for sorting even if prior terms of the index are constrained by IN operators in the WHERE clause.
- Enhance the PRAGMA table_info command so that the "pk" column is an increasing integer to show the order of columns in the primary key.
- Enhance the query optimizer to exploit transitive join constraints.
- Performance improvements in the query optimizer.
- Allow the error message from PRAGMA integrity_check to be longer than 20000 bytes.
- Improved name resolution for deeply nested queries.
- Added the test_regexp.c module as a demonstration of how to implement the REGEXP operator.
- Improved error messages in the RTREE extension.
- Enhance the command-line shell so that a non-zero argument to the ".exit" command causes the shell to exit immediately without cleanly shutting down the database connection.
- Improved error messages for invalid boolean arguments to dot-commands in the command-line shell.
- Improved error messages for "foreign key mismatch" showing the names of the two tables involved.
- Remove all uses of umask() in the unix VFS.
- Added the PRAGMA vdbe_addoptrace and PRAGMA vdbe_debug commands.
- Change to use strncmp() or the equivalent instead of memcmp() when comparing non-zero-terminated strings.
- Update cygwin interfaces to omit deprecated API calls.
- Enhance the spellfix1 extension so that the edit distance cost table can be changed at runtime by inserting a string like 'edit_cost_table=TABLE' into the "command" field.
- Bug fix: repair a long-standing problem that could cause incorrect query results in a 3-way or larger join that compared INTEGER fields against TEXT fields in two or more places. Ticket fc7bd6358f
- Bug fix: Issue an error message if the 16-bit reference counter on a view overflows due to an overly complex query.
- Bug fix: Avoid leaking memory on LIMIT and OFFSET clauses in deeply nested UNION ALL queries.
- Bug fix: Make sure the schema is up-to-date prior to running pragmas table_info, index_list, index_info, and foreign_key_list.
- SQLITE_SOURCE_ID: "2013-03-18 11:39:23 66d5f2b76750f3520eb7a495f6247206758f5b90"
- SHA1 for sqlite3.c: 7308ab891ca1b2ebc596025cfe4dc36f1ee89cf6

2013-01-09 (3.7.15.2)

- Fix a bug, introduced in version 3.7.15, that causes an ORDER BY clause to be optimized out of a three-way join when the ORDER BY is actually required. Ticket 598f5f7596b055
- SQLITE_SOURCE_ID: "2013-01-09 11:53:05 c0e09560d26f0a6456be9dd3447f5311eb4f238f"
- SHA1 for sqlite3.c: 5741f47d1bc38aa0a8c38f09e60a5fe0031f272d

2012-12-19 (3.7.15.1)

- Fix a bug, introduced in version 3.7.15, that causes a segfault if the AS name of a result column of a SELECT statement is used as a logical term in the WHERE clause. Ticket a7b7803e8d1e869.
- SQLITE_SOURCE_ID: "2012-12-19 20:39:10 6b85b767d0ff7975146156a99ad673f2c1a23318"
- SHA1 for sqlite3.c: bbbaa68061e925bd4d7d18d7e1270935c5f7e39a

2012-12-12 (3.7.15)

- Added the sqlite3_errstr() interface.
- Avoid invoking the sqlite3_trace() callback multiple times when a statement is automatically reprepared due to SQLITE_SCHEMA errors.
- Added support for Windows Phone 8 platforms
- Enhance IN operator processing to make use of indices with numeric affinities.
- Do full-table scans using covering indices when possible, under the theory that an index will be smaller and hence can be scanned with less I/O.
- Enhance the query optimizer so that ORDER BY clauses are more aggressively optimized, especially in joins where various terms of the ORDER BY clause come from separate tables of the join.
- Add the ability to implement FROM clause subqueries as coroutines rather that manifesting the subquery into a temporary table.
- Enhancements the command-line shell:
  - Added the ".print" command
  - Negative numbers in the ".width" command cause right-alignment
  - Add the ".wheretrace" command when compiled with SQLITE_DEBUG
- Added the busy_timeout pragma.
- Added the instr() SQL function.
- Added the SQLITE_FCNTL_BUSYHANDLER file control, used to allow VFS implementations to get access to the busy handler callback.
- The xDelete method in the built-in VFSes now return

SQLITE_IOERR_DELETE_NOENT if the file to be deleted does not exist.

- Enhanced support for QNX.
- Work around an optimizer bug in the MSVC compiler when targeting ARM.
- Bug fix: Avoid various concurrency problems in shared cache mode.
- Bug fix: Avoid a deadlock or crash if the backup API, shared cache, and the SQLite Encryption Extension are all used at once.
- Bug fix: SQL functions created using the TCL interface honor the "nullvalue" setting.
- Bug fix: Fix a 32-bit overflow problem on CREATE INDEX for databases larger than 16GB.
- Bug fix: Avoid segfault when using the COLLATE operator inside of a CHECK constraint or view in shared cache mode.
- SQLITE_SOURCE_ID: "2012-12-12 13:36:53 cd0b37c52658bfdf992b1e3dc467bae1835a94ae"
- SHA1 for sqlite3.c: 2b413611f5e3e3b6ef5f618f2a9209cdf25cbcff"

2012-10-04 (3.7.14.1)

- Fix a bug (ticket [d02e1406a58ea02d]]) that causes a segfault on a LEFT JOIN that includes an OR in the ON clause.
- Work around a bug in the optimizer in the VisualStudio-2012 compiler that causes invalid code to be generated when compiling SQLite on ARM.
- Fix the TCL interface so that the "nullvalue" setting is honored for TCL implementations of SQL functions.
- SQLITE_SOURCE_ID: "2012-10-04 19:37:12 091570e46d04e84b67228e0bdbcd6e1fb60c6bdb"
- SHA1 for sqlite3.c: 62aaecaacab3a4bf4a8fe4aec1cfdc1571fe9a44

2012-09-03 (3.7.14)

- Drop built-in support for OS/2. If you need to upgrade an OS/2 application to use this or a later version of SQLite, then add an application-defined VFS using the sqlite3_vfs_register() interface. The code removed in this release can serve as a baseline for the application-defined VFS.
- Ensure that floating point values are preserved exactly when reconstructing a database from the output of the ".dump" command of the command-line shell.
- Added the sqlite3_close_v2() interface.
- Updated the command-line shell so that it can be built using SQLITE_OMIT_FLOATING_POINT and SQLITE_OMIT_AUTOINIT.
- Improvements to the windows makefiles and build processes.
- Enhancements to PRAGMA integrity_check and PRAGMA quick_check so that they can

optionally check just a single attached database instead of all attached databases.
- Enhancements to WAL mode processing that ensure that at least one valid read-mark is available at all times, so that read-only processes can always read the database.
- Performance enhancements in the sorter used by ORDER BY and CREATE INDEX.
- Added the SQLITE_DISABLE_FTS4_DEFERRED compile-time option.
- Better handling of aggregate queries where the aggregate functions are contained within subqueries.
- Enhance the query planner so that it will try to use a covering index on queries that make use of or optimization.
- SQLITE_SOURCE_ID: "2012-09-03 15:42:36 c0d89d4a9752922f9e367362366efde4f1b06f2a"
- SHA1 for sqlite3.c: 5fdf596b29bb426001f28b488ff356ae14d5a5a6

2012-06-11 (3.7.13)

- In-memory databases that are specified using URI filenames are allowed to use shared cache, so that the same in-memory database can be accessed from multiple database connections.
- Recognize and use the mode=memory query parameter in URI filenames.
- Avoid resetting the schema of shared cache connections when any one connection closes. Instead, wait for the last connection to close before resetting the schema.
- In the RTREE extension, when rounding 64-bit floating point numbers to 32-bit for storage, always round in a direction that causes the bounding box to get larger.
- Adjust the unix driver to avoid unnecessary calls to fchown().
- Add interfaces sqlite3_quota_ferror() and sqlite3_quota_file_available() to the test_quota.c module.
- The sqlite3_create_module() and sqlite3_create_module_v2() interfaces return SQLITE_MISUSE on any attempt to overload or replace a virtual table module. The destructor is always called in this case, in accordance with historical and current documentation.
- SQLITE_SOURCE_ID: "2012-06-11 02:05:22 f5b5a13f7394dc143aa136f1d4faba6839eaa6dc"
- SHA1 for sqlite3.c: ff0a771d6252545740ba9685e312b0e3bb6a641b

2012-05-22 (3.7.12.1)

- Fix a bug (ticket c2ad16f997) in the 3.7.12 release that can cause a segfault for certain obscure nested aggregate queries.
- Fix various other minor test script problems.
- SQLITE_SOURCE_ID: "2012-05-22 02:45:53

6d326d44fd1d626aae0e8456e5fa2049f1ce0789"
- SHA1 for sqlite3.c: d494e8d81607f0515d4f386156fb0fd86d5ba7df

2012-05-14 (3.7.12)

- Add the SQLITE_DBSTATUS_CACHE_WRITE option for sqlite3_db_status().
- Optimize the typeof() and length() SQL functions so that they avoid unnecessary reading of database content from disk.
- Add the FTS4 "merge" command, the FTS4 "automerge" command, and the FTS4 "integrity-check" command.
- Report the name of specific CHECK constraints that fail.
- In the command-line shell, use popen() instead of fopen() if the first character of the argument to the ".output" command is "|".
- Make use of OVERLAPPED in the windows VFS to avoid some system calls and thereby obtain a performance improvement.
- More aggressive optimization of the AND operator when one side or the other is always false.
- Improved performance of queries with many OR-connected terms in the WHERE clause that can all be indexed.
- Add the SQLITE_RTREE_INT_ONLY compile-time option to force the R*Tree Extension Module to use integer instead of floating point values for both storage and computation.
- Enhance the PRAGMA integrity_check command to use much less memory when processing multi-gigabyte databases.
- New interfaces added to the test_quota.c add-on module.
- Added the ".trace" dot-command to the command-line shell.
- Allow virtual table constructors to be invoked recursively.
- Improved optimization of ORDER BY clauses on compound queries.
- Improved optimization of aggregate subqueries contained within an aggregate query.
- Bug fix: Fix the RELEASE command so that it does not cancel pending queries. This repairs a problem introduced in 3.7.11.
- Bug fix: Do not discard the DISTINCT as superfluous unless a subset of the result set is subject to a UNIQUE constraint *and* it none of the columns in that subset can be NULL. Ticket 385a5b56b9.
- Bug fix: Do not optimize away an ORDER BY clause that has the same terms as a UNIQUE index unless those terms are also NOT NULL. Ticket 2a5629202f.
- SQLITE_SOURCE_ID: "2012-05-14 01:41:23 8654aa9540fe9fd210899d83d17f3f407096c004"
- SHA1 for sqlite3.c: 57e2104a0f7b3f528e7f6b7a8e553e2357ccd2e1

2012-03-20 (3.7.11)

- Enhance the INSERT syntax to allow multiple rows to be inserted via the VALUES clause.
- Enhance the CREATE VIRTUAL TABLE command to support the IF NOT EXISTS clause.
- Added the sqlite3_stricmp() interface as a counterpart to sqlite3_strnicmp().
- Added the sqlite3_db_readonly() interface.
- Added the SQLITE_FCNTL_PRAGMA file control, giving VFS implementations the ability to add new PRAGMA statements or to override built-in PRAGMAs.
- Queries of the form: "SELECT max(x), y FROM table" returns the value of y on the same row that contains the maximum x value.
- Added support for the FTS4 languageid option.
- Documented support for the FTS4 content option. This feature has actually been in the code since version 3.7.9 but is only now considered to be officially supported.
- Pending statements no longer block ROLLBACK. Instead, the pending statement will return SQLITE_ABORT upon next access after the ROLLBACK.
- Improvements to the handling of CSV inputs in the command-line shell
- Fix a bug introduced in version 3.7.10 that might cause a LEFT JOIN to be incorrectly converted into an INNER JOIN if the WHERE clause indexable terms connected by OR.
- SQLITE_SOURCE_ID: "2012-03-20 11:35:50 00bb9c9ce4f465e6ac321ced2a9d0062dc364669"
- SHA1 for sqlite3.c: d460d7eda3a9dccd291aed2a9fda868b9b120a10

2012-01-16 (3.7.10)

- The default schema format number is changed from 1 to 4. This means that, unless the PRAGMA legacy_file_format=ON statement is run, newly created database files will be unreadable by version of SQLite prior to 3.3.0 (2006-01-10). It also means that the descending indices are enabled by default.
- The sqlite3_pcache_methods structure and the SQLITE_CONFIG_PCACHE and SQLITE_CONFIG_GETPCACHE configuration parameters are deprecated. They are replaced by a new sqlite3_pcache_methods2 structure and SQLITE_CONFIG_PCACHE2 and SQLITE_CONFIG_GETPCACHE2 configuration parameters.
- Added the powersafe overwrite property to the VFS interface. Provide the SQLITE_IOCAP_POWERSAFE_OVERWRITE I/O capability, the SQLITE_POWERSAFE_OVERWRITE compile-time option, and the "psow=BOOLEAN" query parameter for URI filenames.
- Added the sqlite3_db_release_memory() interface and the shrink_memory pragma.
- Added the sqlite3_db_filename() interface.
- Added the sqlite3_stmt_busy() interface.

- Added the sqlite3_uri_boolean() and sqlite3_uri_int64() interfaces.
- If the argument to PRAGMA cache_size is negative N, that means to use approximately -1024*N bytes of memory for the page cache regardless of the page size.
- Enhanced the default memory allocator to make use of _msize() on windows, malloc_size() on Mac, and malloc_usable_size() on Linux.
- Enhanced the query planner to support index queries with range constraints on the rowid.
- Enhanced the query planner flattening logic to allow UNION ALL compounds to be promoted upwards to replace a simple wrapper SELECT even if the compounds are joins.
- Enhanced the query planner so that the xfer optimization can be used with INTEGER PRIMARY KEY ON CONFLICT as long as the destination table is initially empty.
- Enhanced the windows VFS so that all system calls can be overridden using the xSetSystemCall interface.
- Updated the "unix-dotfile" VFS to use locking directories with mkdir() and rmdir() instead of locking files with open() and unlink().
- Enhancements to the test_quota.c extension to support stdio-like interfaces with quotas.
- Change the unix VFS to be tolerant of read() system calls that return less then the full number of requested bytes.
- Change both unix and windows VFSes to report a sector size of 4096 instead of the old default of 512.
- In the TCL Interface, add the -uri option to the "sqlite3" TCL command used for creating new database connection objects.
- Added the SQLITE_TESTCTRL_EXPLAIN_STMT test-control option with the SQLITE_ENABLE_TREE_EXPLAIN compile-time option to enable the command-line shell to display ASCII-art parse trees of SQL statements that it processes, for debugging and analysis.
- **Bug fix:** Add an additional xSync when restarting a WAL in order to prevent an exceedingly unlikely but theoretically possible database corruption following power-loss. Ticket ff5be73dee.
- **Bug fix:** Change the VDBE so that all registers are initialized to Invalid instead of NULL. Ticket 7bbfb7d442
- **Bug fix:** Fix problems that can result from 32-bit integer overflow. Ticket ac00f496b7e2
- SQLITE_SOURCE_ID: "2012-01-16 13:28:40 ebd01a8deffb5024a5d7494eef800d2366d97204"
- SHA1 for sqlite3.c: 6497cbbaad47220bd41e2e4216c54706e7ae95d4

2011-11-01 (3.7.9)

- If a search token (on the right-hand side of the MATCH operator) in FTS4 begins with

"^" then that token must be the first in its field of the document. **Potentially Incompatible Change**

- Added options SQLITE_DBSTATUS_CACHE_HIT and SQLITE_DBSTATUS_CACHE_MISS to the sqlite3_db_status() interface.
- Removed support for SQLITE_ENABLE_STAT2, replacing it with the much more capable SQLITE_ENABLE_STAT3 option.
- Enhancements to the sqlite3_analyzer utility program, including the --pageinfo and --stats options and support for multiplexed databases.
- Enhance the sqlite3_data_count() interface so that it can be used to determine if SQLITE_DONE has been seen on the prepared statement.
- Added the SQLITE_FCNTL_OVERWRITE file-control by which the SQLite core indicates to the VFS that the current transaction will overwrite the entire database file.
- Increase the default lookaside memory allocator allocation size from 100 to 128 bytes.
- Enhanced the query planner so that it can factor terms in and out of OR expressions in the WHERE clause in an effort to find better indices.
- Added the SQLITE_DIRECT_OVERFLOW_READ compile-time option, causing overflow pages to be read directly from the database file, bypassing the page cache.
- Remove limits on the magnitude of precision and width value in the format specifiers of the sqlite3_mprintf() family of string rendering routines.
- Fix a bug that prevent ALTER TABLE ... RENAME from working on some virtual tables in a database with a UTF16 encoding.
- Fix a bug in ASCII-to-float conversion that causes slow performance and incorrect results when converting numbers with ridiculously large exponents.
- Fix a bug that causes incorrect results in aggregate queries that use multiple aggregate functions whose arguments contain complicated expressions that differ only in the case of string literals contained within those expressions.
- Fix a bug that prevented the page_count and quick_check pragmas from working correctly if their names were capitalized.
- Fix a bug that caused VACUUM to fail if the count_changes pragma was engaged.
- Fix a bug in virtual table implementation that causes a crash if an FTS4 table is dropped inside a transaction and a SAVEPOINT occurs afterwards.
- SQLITE_SOURCE_ID: "2011-11-01 00:52:41 c7c6050ef060877ebe77b41d959e9df13f8c9b5e"
- SHA1 for sqlite3.c: becd16877f4f9b281b91c97e106089497d71bb47

2011-09-19 (3.7.8)

- Orders of magnitude performance improvement for CREATE INDEX on very large tables.
- Improved the windows VFS to better defend against interference from anti-virus

software.
- Improved query plan optimization when the DISTINCT keyword is present.
- Allow more system calls to be overridden in the unix VFS - to provide better support for chromium sandboxes.
- Increase the default size of a lookahead cache line from 100 to 128 bytes.
- Enhancements to the test_quota.c module so that it can track preexisting files.
- Bug fix: Virtual tables now handle IS NOT NULL constraints correctly.
- Bug fixes: Correctly handle nested correlated subqueries used with indices in a WHERE clause.
- SQLITE_SOURCE_ID: "2011-09-19 14:49:19 3e0da808d2f5b4d12046e05980ca04578f581177"
- SHA1 for sqlite3.c: bfcd74a655636b592c5dba6d0d5729c0f8e3b4de

2011-06-28 (3.7.7.1)

- Fix a bug causing PRAGMA case_sensitive_like statements compiled using sqlite3_prepare() to fail with an SQLITE_SCHEMA error.
- SQLITE_SOURCE_ID: "2011-06-28 17:39:05 af0d91adf497f5f36ec3813f04235a6e195a605f"
- SHA1 for sqlite3.c: d47594b8a02f6cf58e91fb673e96cb1b397aace0

2011-06-23 (3.7.7)

- Add support for URI filenames
- Add the sqlite3_vtab_config() interface in support of ON CONFLICT clauses with virtual tables.
- Add the xSavepoint, xRelease and xRollbackTo methods in virtual tables in support of SAVEPOINT for virtual tables.
- Update the built-in FTS3/FTS4 and RTREE virtual tables to support ON CONFLICT clauses and REPLACE.
- Avoid unnecessary reparsing of the database schema.
- Added support for the FTS4 prefix option and the FTS4 order option.
- Allow WAL-mode databases to be opened read-only as long as there is an existing read/write connection.
- Added support for short filenames.
- SQLITE_SOURCE_ID: "2011-06-23 19:49:22 4374b7e83ea0a3fbc3691f9c0c936272862f32f2"
- SHA1 for sqlite3.c: 5bbe79e206ae5ffeeca760dbd0d66862228db551

2011-05-19 (3.7.6.3)

- Fix a problem with WAL mode which could cause transactions to silently rollback if the cache_size is set very small (less than 10) and SQLite comes under memory pressure.

2011-04-17 (3.7.6.2)

- Fix the function prototype for the open(2) system call to agree with POSIX. Without this fix, pthreads does not work correctly on NetBSD.
- SQLITE_SOURCE_ID: "2011-04-17 17:25:17 154ddbc17120be2915eb03edc52af1225eb7cb5e"
- SHA1 for sqlite3.c: 806577fd524dd5f3bfd8d4d27392ed2752bc9701

2011-04-13 (3.7.6.1)

- Fix a bug in 3.7.6 that only appears if the SQLITE_FCNTL_SIZE_HINT file control is used with a build of SQLite that makes use of the HAVE_POSIX_FALLOCATE compile-time option and which has SQLITE_ENABLE_LOCKING_MODE turned off.
- SQLITE_SOURCE_ID: "2011-04-13 14:40:25 a35e83eac7b185f4d363d7fa51677f2fdfa27695"
- SHA1 for sqlite3.c: b81bfa27d3e09caf3251475863b1ce6dd9f6ab66

2011-04-12 (3.7.6)

- Added the sqlite3_wal_checkpoint_v2() interface and enhanced the wal_checkpoint pragma to support blocking checkpoints.
- Improvements to the query planner so that it makes better estimates of plan costs and hence does a better job of choosing the right plan, especially when SQLITE_ENABLE_STAT2 is used.
- Fix a bug which prevented deferred foreign key constraints from being enforced when sqlite3_finalize() was not called by one statement with a failed foreign key constraint prior to another statement with foreign key constraints running.
- Integer arithmetic operations that would have resulted in overflow are now performed using floating-point instead.
- Increased the version number on the VFS object to 3 and added new methods xSetSysCall, xGetSysCall, and xNextSysCall used for doing full-coverage testing.
- Increase the maximum value of SQLITE_MAX_ATTACHED from 30 to 62 (though the default value remains at 10).
- Enhancements to FTS4:
  1. Added the fts4aux table
  2. Added support for compressed FTS4 content
- Enhance the ANALYZE command to support the name of an index as its argument, in

order to analyze just that one index.
- Added the "unix-excl" built-in VFS on unix and unix-like platforms.
- SQLITE_SOURCE_ID: "2011-04-12 01:58:40 f9d43fa363d54beab6f45db005abac0a7c0c47a7"
- SHA1 for sqlite3.c: f38df08547efae0ff4343da607b723f588bbd66b

2011-02-01 (3.7.5)

- Added the sqlite3_vsnprintf() interface.
- Added the SQLITE_DBSTATUS_LOOKASIDE_HIT, SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE, and SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL options for the sqlite3_db_status() interface.
- Added the SQLITE_OMIT_AUTORESET compile-time option.
- Added the SQLITE_DEFAULT_FOREIGN_KEYS compile-time option.
- Updates to sqlite3_stmt_readonly() so that its result is well-defined for all prepared statements and so that it works with VACUUM.
- Added the "-heap" option to the command-line shell
- Fix a bug involving frequent changes in and out of WAL mode and VACUUM that could (in theory) cause database corruption.
- Enhance the sqlite3_trace() mechanism so that nested SQL statements such as might be generated by virtual tables are shown but are shown in comments and without parameter expansion. This greatly improves tracing output when using the FTS3/4 and/or RTREE virtual tables.
- Change the xFileControl() methods on all built-in VFSes to return SQLITE_NOTFOUND instead of SQLITE_ERROR for an unrecognized operation code.
- The SQLite core invokes the SQLITE_FCNTL_SYNC_OMITTED file control to the VFS in place of a call to xSync if the database has PRAGMA synchronous set to OFF.

2010-12-07 (3.7.4)

- Added the sqlite3_blob_reopen() interface to allow an existing sqlite3_blob object to be rebound to a new row.
- Use the new sqlite3_blob_reopen() interface to improve the performance of FTS.
- VFSes that do not support shared memory are allowed to access WAL databases if PRAGMA locking_mode is set to EXCLUSIVE.
- Enhancements to EXPLAIN QUERY PLAN.
- Added the sqlite3_stmt_readonly() interface.
- Added PRAGMA checkpoint_fullfsync.
- Added the SQLITE_FCNTL_FILE_POINTER option to sqlite3_file_control().

- Added support for FTS4 and enhancements to the FTS matchinfo() function.
- Added the test_superlock.c module which provides example code for obtaining an exclusive lock to a rollback or WAL database.
- Added the test_multiplex.c module which provides an example VFS that provides multiplexing (sharding) of a DB, splitting it over multiple files of fixed size.
- A very obscure bug associated with the or optimization was fixed.

2010-10-08 (3.7.3)

- Added the sqlite3_create_function_v2() interface that includes a destructor callback.
- Added support for custom r-tree queries using application-supplied callback routines to define the boundary of the query region.
- The default page cache strives more diligently to avoid using memory beyond what is allocated to it by SQLITE_CONFIG_PAGECACHE. Or if using page cache is allocating from the heap, it strives to avoid going over the sqlite3_soft_heap_limit64(), even if SQLITE_ENABLE_MEMORY_MANAGEMENT is not set.
- Added the sqlite3_soft_heap_limit64() interface as a replacement for sqlite3_soft_heap_limit().
- The ANALYZE command now gathers statistics on tables even if they have no indices.
- Tweaks to the query planner to help it do a better job of finding the most efficient query plan for each query.
- Enhanced the internal text-to-numeric conversion routines so that they work with UTF8 or UTF16, thereby avoiding some UTF16-to-UTF8 text conversions.
- Fix a problem that was causing excess memory usage with large WAL transactions in win32 systems.
- The interface between the VDBE and B-Tree layer is enhanced such that the VDBE provides hints to the B-Tree layer letting the B-Tree layer know when it is safe to use hashing instead of B-Trees for transient tables.
- Miscellaneous documentation enhancements.

2010-08-24 (3.7.2)

- Fix an old and very obscure bug that can lead to corruption of the database free-page list when incremental_vacuum is used.

2010-08-23 (3.7.1)

- Added new commands SQLITE_DBSTATUS_SCHEMA_USED and SQLITE_DBSTATUS_STMT_USED to the sqlite3_db_status() interface, in order to report out the amount of memory used to hold the schema and prepared statements of

a connection.
- Increase the maximum size of a database pages from 32KiB to 64KiB.
- Use the LIKE optimization even if the right-hand side string contains no wildcards.
- Added the SQLITE_FCNTL_CHUNK_SIZE verb to the sqlite3_file_control() interface for both unix and windows, to cause database files to grow in large chunks in order to reduce disk fragmentation.
- Fixed a bug in the query planner that caused performance regressions relative to 3.6.23.1 on some complex joins.
- Fixed a typo in the OS/2 backend.
- Refactored the pager module.
- The SQLITE_MAX_PAGE_SIZE compile-time option is now silently ignored. The maximum page size is hard-coded at 65536 bytes.

2010-08-04 (3.7.0.1)

- Fix a potential database corruption bug that can occur if version 3.7.0 and version 3.6.23.1 alternately write to the same database file. Ticket [51ae9cad317a1]
- Fix a performance regression related to the query planner enhancements of version 3.7.0.

2010-07-21 (3.7.0)

- Added support for write-ahead logging.
- Query planner enhancement - automatic transient indices are created when doing so reduces the estimated query time.
- Query planner enhancement - the ORDER BY becomes a no-op if the query also contains a GROUP BY clause that forces the correct output order.
- Add the SQLITE_DBSTATUS_CACHE_USED verb for sqlite3_db_status().
- The logical database size is now stored in the database header so that bytes can be appended to the end of the database file without corrupting it and so that SQLite will work correctly on systems that lack support for ftruncate().

2010-03-26 (3.6.23.1)

- Fix a bug in the offsets() function of FTS3
- Fix a missing "sync" that when omitted could lead to database corruption if a power failure or OS crash occurred just as a ROLLBACK operation was finishing.

2010-03-09 (3.6.23)

- Added the secure_delete pragma.

- Added the sqlite3_compileoption_used() and sqlite3_compileoption_get() interfaces as well as the compile_options pragma and the sqlite_compileoption_used() and sqlite_compileoption_get() SQL functions.
- Added the sqlite3_log() interface together with the SQLITE_CONFIG_LOG verb to sqlite3_config(). The ".log" command is added to the Command Line Interface.
- Improvements to FTS3.
- Improvements and bug-fixes in support for SQLITE_OMIT_FLOATING_POINT.
- The integrity_check pragma is enhanced to detect out-of-order rowids.
- The ".genfkey" operator has been removed from the Command Line Interface.
- Updates to the co-hosted Lemon LALR(1) parser generator. (These updates did not affect SQLite.)
- Various minor bug fixes and performance enhancements.

2010-01-06 (3.6.22)

- Fix bugs that can (rarely) lead to incorrect query results when the CAST or OR operators are used in the WHERE clause of a query.
- Continuing enhancements and improvements to FTS3.
- Other miscellaneous bug fixes.

2009-12-07 (3.6.21)

- The SQL output resulting from sqlite3_trace() is now modified to include the values of bound parameters.
- Performance optimizations targeting a specific use case from a single high-profile user of SQLite. A 12% reduction in the number of CPU operations is achieved (as measured by Valgrind). Actual performance improvements in practice may vary depending on workload. Changes include:
    - The ifnull() and coalesce() SQL functions are now implemented using in-line VDBE code rather than calling external functions, so that unused arguments need never be evaluated.
    - The substr() SQL function does not bother to measure the length its entire input string if it is only computing a prefix
    - Unnecessary OP_IsNull, OP_Affinity, and OP_MustBeInt VDBE opcodes are suppressed
    - Various code refactorizations for performance
- The FTS3 extension has undergone a major rework and cleanup. New FTS3 documentation is now available.
- The SQLITE_SECURE_DELETE compile-time option fixed to make sure that content is deleted even when the truncate optimization applies.

- Improvements to "dot-command" handling in the Command Line Interface.
- Other minor bug fixes and documentation enhancements.

2009-11-04 (3.6.20)

- Optimizer enhancement: prepared statements are automatically re-compiled when a binding on the RHS of a LIKE operator changes or when any range constraint changes under SQLITE_ENABLE_STAT2.
- Various minor bug fixes and documentation enhancements.

2009-10-30 (3.6.16.1)

- A small patch to version 3.6.16 to fix the OP_If bug.

2009-10-14 (3.6.19)

- Added support for foreign key constraints. Foreign key constraints are disabled by default. Use the foreign_keys pragma to turn them on.
- Generalized the IS and IS NOT operators to take arbitrary expressions on their right-hand side.
- The TCL Interface has been enhanced to use the Non-Recursive Engine (NRE) interface to the TCL interpreter when linked against TCL 8.6 or later.
- Fix a bug introduced in 3.6.18 that can lead to a segfault when an attempt is made to write on a read-only database.

2009-09-11 (3.6.18)

- Versioning of the SQLite source code has transitioned from CVS to Fossil.
- Query planner enhancements.
- The SQLITE_ENABLE_STAT2 compile-time option causes the ANALYZE command to collect a small histogram of each index, to help SQLite better select among competing range query indices.
- Recursive triggers can be enabled using the PRAGMA recursive_triggers statement.
- Delete triggers fire when rows are removed due to a REPLACE conflict resolution. This feature is only enabled when recursive triggers are enabled.
- Added the SQLITE_OPEN_SHAREDCACHE and SQLITE_OPEN_PRIVATECACHE flags for sqlite3_open_v2() used to override the global shared cache mode settings for individual database connections.
- Added improved version identification features: C-Preprocessor macro SQLITE_SOURCE_ID, C/C++ interface sqlite3_sourceid(), and SQL function sqlite_source_id().

- Obscure bug fix on triggers ([efc02f9779]).

2009-08-10 (3.6.17)

- Expose the sqlite3_strnicmp() interface for use by extensions and applications.
- Remove the restriction on virtual tables and shared cache mode. Virtual tables and shared cache can now be used at the same time.
- Many code simplifications and obscure bug fixes in support of providing 100% branch test coverage.

2009-06-27 (3.6.16)

- Fix a bug (ticket #3929) that occasionally causes INSERT or UPDATE operations to fail on an indexed table that has a self-modifying trigger.
- Other minor bug fixes and performance optimizations.

2009-06-15 (3.6.15)

- Refactor the internal representation of SQL expressions so that they use less memory on embedded platforms.
- Reduce the amount of stack space used
- Fix an 64-bit alignment bug on HP/UX and Sparc
- The sqlite3_create_function() family of interfaces now return SQLITE_MISUSE instead of SQLITE_ERROR when passed invalid parameter combinations.
- When new tables are created using CREATE TABLE ... AS SELECT ... the datatype of the columns is the simplified SQLite datatype (TEXT, INT, REAL, NUMERIC, or BLOB) instead of a copy of the original datatype from the source table.
- Resolve race conditions when checking for a hot rollback journal.
- The sqlite3_shutdown() interface frees all mutexes under windows.
- Enhanced robustness against corrupt database files
- Continuing improvements to the test suite and fixes to obscure bugs and inconsistencies that the test suite improvements are uncovering.

2009-05-25 (3.6.14.2)

- Fix a code generator bug introduced in version 3.6.14. This bug can cause incorrect query results under obscure circumstances. Ticket #3879.

2009-05-19 (3.6.14.1)

- Fix a bug in group_concat(), ticket #3841

- Fix a performance bug in the pager cache, ticket #3844
- Fix a bug in the sqlite3_backup implementation that can lead to a corrupt backup database. Ticket #3858.

2009-05-07 (3.6.14)

- Added the optional asynchronous VFS module.
- Enhanced the query optimizer so that virtual tables are able to make use of OR and IN operators in the WHERE clause.
- Speed improvements in the btree and pager layers.
- Added the SQLITE_HAVE_ISNAN compile-time option which will cause the isnan() function from the standard math library to be used instead of SQLite's own home-brew NaN checker.
- Countless minor bug fixes, documentation improvements, new and improved test cases, and code simplifications and cleanups.

2009-04-13 (3.6.13)

- Fix a bug in version 3.6.12 that causes a segfault when running a count(*) on the sqlite_master table of an empty database. Ticket #3774.
- Fix a bug in version 3.6.12 that causes a segfault that when inserting into a table using a DEFAULT value where there is a function as part of the DEFAULT value expression. Ticket #3791.
- Fix data structure alignment issues on Sparc. Ticket #3777.
- Other minor bug fixes.

2009-03-31 (3.6.12)

- Fixed a bug that caused database corruption when an incremental_vacuum is rolled back in an in-memory database. Ticket #3761.
- Added the sqlite3_unlock_notify() interface.
- Added the reverse_unordered_selects pragma.
- The default page size on windows is automatically adjusted to match the capabilities of the underlying filesystem.
- Add the new ".genfkey" command in the CLI for generating triggers to implement foreign key constraints.
- Performance improvements for "count(*)" queries.
- Reduce the amount of heap memory used, especially by TRIGGERs.

2009-02-18 (3.6.11)

- Added the hot-backup interface.
- Added new commands ".backup" and ".restore" to the CLI.
- Added new methods backup and restore to the TCL interface.
- Improvements to the syntax bubble diagrams
- Various minor bug fixes

2009-01-15 (3.6.10)

- Fix a cache coherency problem that could lead to database corruption. Ticket #3584.

2009-01-14 (3.6.9)

- Fix two bugs, which when combined might result in incorrect query results. Both bugs were harmless by themselves; only when they team up do they cause problems. Ticket #3581.

2009-01-12 (3.6.8)

- Added support for nested transactions
- Enhanced the query optimizer so that it is able to use multiple indices to efficiently process OR-connected constraints in a WHERE clause.
- Added support for parentheses in FTS3 query patterns using the SQLITE_ENABLE_FTS3_PARENTHESIS compile-time option.

2008-12-16 (3.6.7)

- Reorganize the Unix interface in os_unix.c
- Added support for "Proxy Locking" on Mac OS X.
- Changed the prototype of the sqlite3_auto_extension() interface in a way that is backwards compatible but which might cause warnings in new builds of applications that use that interface.
- Changed the signature of the xDlSym method of the sqlite3_vfs object in a way that is backwards compatible but which might cause compiler warnings.
- Added superfluous casts and variable initializations in order to suppress nuisance compiler warnings.
- Fixes for various minor bugs.

2008-11-26 (3.6.6.2)

- Fix a bug in the b-tree delete algorithm that seems like it might be able to cause database corruption. The bug was first introduced in version 3.6.6 by check-in [5899] on

2008-11-13.
- Fix a memory leak that can occur following a disk I/O error.

2008-11-22 (3.6.6.1)

- Fix a bug in the page cache that can lead database corruption following a rollback. This bug was first introduced in version 3.6.4.
- Two other very minor bug fixes

2008-11-19 (3.6.6)

- Fix a #define that prevented memsys5 from compiling
- Fix a problem in the virtual table commit mechanism that was causing a crash in FTS3. Ticket #3497.
- Add the application-defined page cache
- Added built-in support for VxWorks

2008-11-12 (3.6.5)

- Add the MEMORY option to the journal_mode pragma.
- Added the sqlite3_db_mutex() interface.
- Added the SQLITE_OMIT_TRUNCATE_OPTIMIZATION compile-time option.
- Fixed the truncate optimization so that sqlite3_changes() and sqlite3_total_changes() interfaces and the count_changes pragma return the correct values.
- Added the sqlite3_extended_errcode() interface.
- The COMMIT command now succeeds even if there are pending queries. It returns SQLITE_BUSY if there are pending incremental BLOB I/O requests.
- The error code is changed to SQLITE_BUSY (instead of SQLITE_ERROR) when an attempt is made to ROLLBACK while one or more queries are still pending.
- Drop all support for the experimental memory allocators memsys4 and memsys6.
- Added the SQLITE_ZERO_MALLOC compile-time option.

2008-10-15 (3.6.4)

- Add option support for LIMIT and ORDER BY clauses on DELETE and UPDATE statements. Only works if SQLite is compiled with SQLITE_ENABLE_UPDATE_DELETE_LIMIT.
- Added the sqlite3_stmt_status() interface for performance monitoring.
- Add the INDEXED BY clause.
- The LOCKING_STYLE extension is now enabled by default on Mac OS X
- Added the TRUNCATE option to PRAGMA journal_mode

- Performance enhancements to tree balancing logic in the B-Tree layer.
- Added the source code and documentation for the **genfkey** program for automatically generating triggers to enforce foreign key constraints.
- Added the SQLITE_OMIT_TRUNCATE_OPTIMIZATION compile-time option.
- The SQL language documentation is converted to use syntax diagrams instead of BNF.
- Other minor bug fixes

2008-09-22 (3.6.3)

- Fix for a bug in the SELECT DISTINCT logic that was introduced by the prior version.
- Other minor bug fixes

2008-08-30 (3.6.2)

- Split the pager subsystem into separate pager and pcache subsystems.
- Factor out identifier resolution procedures into separate files.
- Bug fixes

2008-08-06 (3.6.1)

- Added the lookaside memory allocator for a speed improvement in excess of 15% on some workloads. (Your mileage may vary.)
- Added the SQLITE_CONFIG_LOOKASIDE verb to sqlite3_config() to control the default lookaside configuration.
- Added verbs SQLITE_STATUS_PAGECACHE_SIZE and SQLITE_STATUS_SCRATCH_SIZE to the sqlite3_status() interface.
- Modified SQLITE_CONFIG_PAGECACHE and SQLITE_CONFIG_SCRATCH to remove the "+4" magic number in the buffer size computation.
- Added the sqlite3_db_config() and sqlite3_db_status() interfaces for controlling and monitoring the lookaside allocator separately on each database connection.
- Numerous other performance enhancements
- Miscellaneous minor bug fixes

2008-07-16 (3.6.0 beta)

- Modifications to the virtual file system interface to support a wider range of embedded systems. See 35to36.html for additional information. ***Potentially incompatible change***
- All C-preprocessor macros used to control compile-time options now begin with the prefix "SQLITE_". This may require changes to applications that compile SQLite using their own makefiles and with custom compile-time options, hence we mark this as a ***Potentially incompatible change***

- The SQLITE_MUTEX_APPDEF compile-time option is no longer supported. Alternative mutex implementations can now be added at run-time using the sqlite3_config() interface with the SQLITE_CONFIG_MUTEX verb. ***Potentially incompatible change***
- The handling of IN and NOT IN operators that contain a NULL on their right-hand side expression is brought into compliance with the SQL standard and with other SQL database engines. This is a bug fix, but as it has the potential to break legacy applications that depend on the older buggy behavior, we mark that as a ***Potentially incompatible change***
- The result column names generated for compound subqueries have been simplified to show only the name of the column of the original table and omit the table name. This makes SQLite operate more like other SQL database engines.
- Added the sqlite3_config() interface for doing run-time configuration of the entire SQLite library.
- Added the sqlite3_status() interface used for querying run-time status information about the overall SQLite library and its subsystems.
- Added the sqlite3_initialize() and sqlite3_shutdown() interfaces.
- The SQLITE_OPEN_NOMUTEX option was added to sqlite3_open_v2().
- Added the PRAGMA page_count command.
- Added the sqlite3_next_stmt() interface.
- Added a new R*Tree virtual table

2008-05-14 (3.5.9)

- Added *experimental* support for the journal_mode PRAGMA and persistent journal.
- Journal mode PERSIST is the default behavior in exclusive locking mode.
- Fix a performance regression on LEFT JOIN (see ticket #3015) that was mistakenly introduced in version 3.5.8.
- Performance enhancement: Reengineer the internal routines used to interpret and render variable-length integers.
- Fix a buffer-overrun problem in sqlite3_mprintf() which occurs when a string without a zero-terminator is passed to "%.*s".
- Always convert IEEE floating point NaN values into NULL during processing. (Ticket #3060)
- Make sure that when a connection blocks on a RESERVED lock that it is able to continue after the lock is released. (Ticket #3093)
- The "configure" scripts should now automatically configure Unix systems for large file support. Improved error messages for when large files are encountered and large file support is disabled.
- Avoid cache pages leaks following disk-full or I/O errors
- And, many more minor bug fixes and performance enhancements....

2008-04-16 (3.5.8)

- Expose SQLite's internal pseudo-random number generator (PRNG) via the sqlite3_randomness() interface
- New interface sqlite3_context_db_handle() that returns the database connection handle that has invoked an application-defined SQL function.
- New interface sqlite3_limit() allows size and length limits to be set on a per-connection basis and at run-time.
- Improved crash-robustness: write the database page size into the rollback journal header.
- Allow the VACUUM command to change the page size of a database file.
- The xAccess() method of the VFS is allowed to return -1 to signal a memory allocation error.
- Performance improvement: The OP_IdxDelete opcode uses unpacked records, obviating the need for one OP_MakeRecord opcode call for each index record deleted.
- Performance improvement: Constant subexpressions are factored out of loops.
- Performance improvement: Results of OP_Column are reused rather than issuing multiple OP_Column opcodes.
- Fix a bug in the RTRIM collating sequence.
- Fix a bug in the SQLITE_SECURE_DELETE option that was causing Firefox crashes. Make arrangements to always test SQLITE_SECURE_DELETE prior to each release.
- Other miscellaneous performance enhancements.
- Other miscellaneous minor bug fixes.

2008-03-17 (3.5.7)

- Fix a bug (ticket #2927) in the register allocation for compound selects - introduced by the new VM code in version 3.5.5.
- ALTER TABLE uses double-quotes instead of single-quotes for quoting filenames.
- Use the WHERE clause to reduce the size of a materialized VIEW in an UPDATE or DELETE statement. (Optimization)
- Do not apply the flattening optimization if the outer query is an aggregate and the inner query contains ORDER BY. (Ticket #2943)
- Additional OS/2 updates
- Added an experimental power-of-two, first-fit memory allocator.
- Remove all instances of sprintf() from the code
- Accept "Z" as the zulu timezone at the end of date strings
- Fix a bug in the LIKE optimizer that occurs when the last character before the first wildcard is an upper-case "Z"
- Added the "bitvec" object for keeping track of which pages have been journalled.

Improves speed and reduces memory consumption, especially for large database files.
- Get the SQLITE_ENABLE_LOCKING_STYLE macro working again on Mac OS X.
- Store the statement journal in the temporary file directory instead of collocated with the database file.
- Many improvements and cleanups to the configure script

2008-02-06 (3.5.6)

- Fix a bug (ticket #2913) that prevented virtual tables from working in a LEFT JOIN. The problem was introduced into shortly before the 3.5.5 release.
- Bring the OS/2 porting layer up-to-date.
- Add the new sqlite3_result_error_code() API and use it in the implementation of ATTACH so that proper error codes are returned when an ATTACH fails.

2008-01-31 (3.5.5)

- Convert the underlying virtual machine to be a register-based machine rather than a stack-based machine. The only user-visible change is in the output of EXPLAIN.
- Add the build-in RTRIM collating sequence.

2007-12-14 (3.5.4)

- Fix a critical bug in UPDATE or DELETE that occurs when an OR REPLACE clause or a trigger causes rows in the same table to be deleted as side effects. (See ticket #2832.) The most likely result of this bug is a segmentation fault, though database corruption is a possibility.
- Bring the processing of ORDER BY into compliance with the SQL standard for case where a result alias and a table column name are in conflict. Correct behavior is to prefer the result alias. Older versions of SQLite incorrectly picked the table column. (See ticket #2822.)
- The VACUUM command preserves the setting of the legacy_file_format pragma. (Ticket #2804.)
- Productize and officially support the group_concat() SQL function.
- Better optimization of some IN operator expressions.
- Add the ability to change the auto_vacuum status of a database by setting the auto_vaccum pragma and VACUUMing the database.
- Prefix search in FTS3 is much more efficient.
- Relax the SQL statement length restriction in the CLI so that the ".dump" output of databases with very large BLOBs and strings can be played back to recreate the database.

- Other small bug fixes and optimizations.

2007-11-27 (3.5.3)

- Move website and documentation files out of the source tree into a separate CM system.
- Fix a long-standing bug in INSERT INTO ... SELECT ... statements where the SELECT is compound.
- Fix a long-standing bug in RAISE(IGNORE) as used in BEFORE triggers.
- Fixed the operator precedence for the ~ operator.
- On Win32, do not return an error when attempting to delete a file that does not exist.
- Allow collating sequence names to be quoted.
- Modify the TCL interface to use sqlite3_prepare_v2().
- Fix multiple bugs that can occur following a malloc() failure.
- sqlite3_step() returns SQLITE_MISUSE instead of crashing when called with a NULL parameter.
- FTS3 now uses the SQLite memory allocator exclusively. The FTS3 amalgamation can now be appended to the SQLite amalgamation to generate a super-amalgamation containing both.
- The DISTINCT keyword now will sometimes use an INDEX if an appropriate index is available and the optimizer thinks its use might be advantageous.

2007-11-05 (3.5.2)

- Dropped support for the SQLITE_OMIT_MEMORY_ALLOCATION compile-time option.
- Always open files using FILE_FLAG_RANDOM_ACCESS under Windows.
- The 3rd parameter of the built-in SUBSTR() function is now optional.
- Bug fix: do not invoke the authorizer when reparsing the schema after a schema change.
- Added the experimental malloc-free memory allocator in mem3.c.
- Virtual machine stores 64-bit integer and floating point constants in binary instead of text for a performance boost.
- Fix a race condition in test_async.c.
- Added the ".timer" command to the CLI

2007-10-04 (3.5.1)

- ***Nota Bene:*** *We are not using terms "alpha" or "beta" on this release because the code is stable and because if we use those terms, nobody will upgrade. However, we still reserve the right to make incompatible changes to the new VFS interface in future* *releases.*

- Fix a bug in the handling of SQLITE_FULL errors that could lead to database corruption. Ticket #2686.
- The test_async.c drive now does full file locking and works correctly when used simultaneously by multiple processes on the same database.
- The CLI ignores whitespace (including comments) at the end of lines
- Make sure the query optimizer checks dependencies on all terms of a compound SELECT statement. Ticket #2640.
- Add demonstration code showing how to build a VFS for a raw mass storage without a filesystem.
- Added an output buffer size parameter to the xGetTempname() method of the VFS layer.
- Sticky SQLITE_FULL or SQLITE_IOERR errors in the pager are reset when a new transaction is started.

2007-09-04 (3.5.0) alpha

- Redesign the OS interface layer. See 34to35.html for details. ***Potentially incompatible change***
- The sqlite3_release_memory(), sqlite3_soft_heap_limit(), and sqlite3_enable_shared_cache() interfaces now work cross all threads in the process, not just the single thread in which they are invoked. ***Potentially incompatible change***
- Added the sqlite3_open_v2() interface.
- Reimplemented the memory allocation subsystem and made it replaceable at compile-time.
- Created a new mutex subsystem and made it replicable at compile-time.
- The same database connection may now be used simultaneously by separate threads.

2007-08-13 (3.4.2)

- Fix a database corruption bug that might occur if a ROLLBACK command is executed in auto-vacuum mode and a very small sqlite3_soft_heap_limit is set. Ticket #2565.
- Add the ability to run a full regression test with a small sqlite3_soft_heap_limit.
- Fix other minor problems with using small soft heap limits.
- Work-around for GCC bug 32575.
- Improved error detection of misused aggregate functions.
- Improvements to the amalgamation generator script so that all symbols are prefixed with either SQLITE_PRIVATE or SQLITE_API.

2007-07-20 (3.4.1)

- Fix a bug in VACUUM that can lead to database corruption if two processes are connected to the database at the same time and one VACUUMs then the other then modifies the database.
- The expression "+column" is now considered the same as "column" when computing the collating sequence to use on the expression.
- In the TCL language interface, "@variable" instead of "$variable" always binds as a blob.
- Added PRAGMA freelist_count for determining the current size of the freelist.
- The PRAGMA auto_vacuum=incremental setting is now persistent.
- Add FD_CLOEXEC to all open files under Unix.
- Fix a bug in the min()/max() optimization when applied to descending indices.
- Make sure the TCL language interface works correctly with 64-bit integers on 64-bit machines.
- Allow the value -9223372036854775808 as an integer literal in SQL statements.
- Add the capability of "hidden" columns in virtual tables.
- Use the macro SQLITE_PRIVATE (defaulting to "static") on all internal functions in the amalgamation.
- Add pluggable tokenizers and ICU tokenization support to FTS2
- Other minor bug fixes and documentation enhancements

2007-06-18 (3.4.0)

- Fix a bug that can lead to database corruption if an SQLITE_BUSY error occurs in the middle of an explicit transaction and that transaction is later committed. Ticket #2409. See the CorruptionFollowingBusyError wiki page for details.
- Fix a bug that can lead to database corruption if autovacuum mode is on and a malloc() failure follows a CREATE TABLE or CREATE INDEX statement which itself follows a cache overflow inside a transaction. See ticket #2418.
- Added explicit upper bounds on the sizes and quantities of things SQLite can process. This change might cause compatibility problems for applications that use SQLite in the extreme, which is why the current release is 3.4.0 instead of 3.3.18.
- Added support for Incremental BLOB I/O.
- Added the sqlite3_bind_zeroblob() API and the zeroblob() SQL function.
- Added support for Incremental Vacuum.
- Added the SQLITE_MIXED_ENDIAN_64BIT_FLOAT compile-time option to support ARM7 processors with goofy endianness.
- Removed all instances of sprintf() and strcpy() from the core library.
- Added support for International Components for Unicode (ICU) to the full-text search extensions.

- In the Windows OS driver, reacquire a SHARED lock if an attempt to acquire an EXCLUSIVE lock fails. Ticket #2354

- Fix the REPLACE() function so that it returns NULL if the second argument is an empty string. Ticket #2324.
- Document the hazards of type conversions in sqlite3_column_blob() and related APIs. Fix unnecessary type conversions. Ticket #2321.
- Internationalization of the TRIM() function. Ticket #2323
- Use memmove() instead of memcpy() when moving between memory regions that might overlap. Ticket #2334
- Fix an optimizer bug involving subqueries in a compound SELECT that has both an ORDER BY and a LIMIT clause. Ticket #2339.
- Make sure the sqlite3_snprintf() interface does not zero-terminate the buffer if the buffer size is less than 1. Ticket #2341
- Fix the built-in printf logic so that it prints "NaN" not "Inf" for floating-point NaNs. Ticket #2345
- When converting BLOB to TEXT, use the text encoding of the main database. Ticket #2349
- Keep the full precision of integers (if possible) when casting to NUMERIC. Ticket #2364
- Fix a bug in the handling of UTF16 codepoint 0xE000
- Consider explicit collate clauses when matching WHERE constraints to indices in the query optimizer. Ticket #2391
- Fix the query optimizer to correctly handle constant expressions in the ON clause of a LEFT JOIN. Ticket #2403
- Fix the query optimizer to handle rowid comparisons to NULL correctly. Ticket #2404
- Fix many potential segfaults that could be caused by malicious SQL statements.

2007-04-25 (3.3.17)

- When the "write_version" value of the database header is larger than what the library understands, make the database read-only instead of unreadable.
- Other minor bug fixes

2007-04-18 (3.3.16)

- Fix a bug that caused VACUUM to fail if NULLs appeared in a UNIQUE column.
- Reinstate performance improvements that were added in Version 3.3.14 but regressed in Version 3.3.15.
- Fix problems with the handling of ORDER BY expressions on compound SELECT statements in subqueries.
- Fix a potential segfault when destroying locks on WinCE in a multi-threaded

environment.
- Documentation updates.

2007-04-09 (3.3.15)

- Fix a bug introduced in 3.3.14 that caused a rollback of CREATE TEMP TABLE to leave the database connection wedged.
- Fix a bug that caused an extra NULL row to be returned when a descending query was interrupted by a change to the database.
- The FOR EACH STATEMENT clause on a trigger now causes a syntax error. It used to be silently ignored.
- Fix an obscure and relatively harmless problem that might have caused a resource leak following an I/O error.
- Many improvements to the test suite. Test coverage now exceeded 98%

2007-04-02 (3.3.14)

- Fix a bug (ticket #2273) that could cause a segfault when the IN operator is used one one term of a two-column index and the right-hand side of the IN operator contains a NULL.
- Added a new OS interface method for determining the sector size of underlying media: sqlite3OsSectorSize().
- A new algorithm for statements of the form INSERT INTO *table1* SELECT * FROM *table2* is faster and reduces fragmentation. VACUUM uses statements of this form and thus runs faster and defragments better.
- Performance enhancements through reductions in disk I/O:
    - Do not read the last page of an overflow chain when deleting the row - just add that page to the freelist.
    - Do not store pages being deleted in the rollback journal.
    - Do not read in the (meaningless) content of pages extracted from the freelist.
    - Do not flush the page cache (and thus avoiding a cache refill) unless another process changes the underlying database file.
    - Truncate rather than delete the rollback journal when committing a transaction in exclusive access mode, or when committing the TEMP database.
- Added support for exclusive access mode using "PRAGMA locking_mode=EXCLUSIVE"
- Use heap space instead of stack space for large buffers in the pager - useful on embedded platforms with stack-space limitations.
- Add a makefile target "sqlite3.c" that builds an amalgamation containing the core SQLite library C code in a single file.

- Get the library working correctly when compiled with GCC option "-fstrict-aliasing".
- Removed the vestigal SQLITE_PROTOCOL error.
- Improvements to test coverage, other minor bugs fixed, memory leaks plugged, code refactored and/or recommended in places for easier reading.

2007-02-13 (3.3.13)

- Add a "fragmentation" measurement in the output of sqlite3_analyzer.
- Add the COLLATE operator used to explicitly set the collating sequence used by an expression. This feature is considered experimental pending additional testing.
- Allow up to 64 tables in a join - the old limit was 32.
- Added two new experimental functions: randomBlob() and hex(). Their intended use is to facilitate generating UUIDs.
- Fix a problem where PRAGMA count_changes was causing incorrect results for updates on tables with triggers
- Fix a bug in the ORDER BY clause optimizer for joins where the left-most table in the join is constrained by a UNIQUE index.
- Fixed a bug in the "copy" method of the TCL interface.
- Bug fixes in fts1 and fts2 modules.

2007-01-27 (3.3.12)

- Fix another bug in the IS NULL optimization that was added in version 3.3.9.
- Fix an assertion fault that occurred on deeply nested views.
- Limit the amount of output that PRAGMA integrity_check generates.
- Minor syntactic changes to support a wider variety of compilers.

2007-01-22 (3.3.11)

- Fix another bug in the implementation of the new sqlite3_prepare_v2() API. We'll get it right eventually...
- Fix a bug in the IS NULL optimization that was added in version 3.3.9 - the bug was causing incorrect results on certain LEFT JOINs that included in the WHERE clause an IS NULL constraint for the right table of the LEFT JOIN.
- Make AreFileApisANSI() a no-op macro in WinCE since WinCE does not support this function.

2007-01-09 (3.3.10)

- Fix bugs in the implementation of the new sqlite3_prepare_v2() API that can lead to segfaults.

- Fix 1-second round-off errors in the strftime() function
- Enhance the Windows OS layer to provide detailed error codes
- Work around a win2k problem so that SQLite can use single-character database file names
- The user_version and schema_version pragmas correctly set their column names in the result set
- Documentation updates

2007-01-04 (3.3.9)

- Fix bugs in pager.c that could lead to database corruption if two processes both try to recover a hot journal at the same instant
- Added the sqlite3_prepare_v2() API.
- Fixed the ".dump" command in the command-line shell to show indices, triggers and views again.
- Change the table_info pragma so that it returns NULL for the default value if there is no default value
- Support for non-ASCII characters in win95 filenames
- Query optimizer enhancements:
    - Optimizer does a better job of using indices to satisfy ORDER BY clauses that sort on the integer primary key
    - Use an index to satisfy an IS NULL operator in the WHERE clause
    - Fix a bug that was causing the optimizer to miss an OR optimization opportunity
    - The optimizer has more freedom to reorder tables in the FROM clause even in there are LEFT joins.
- Extension loading supported added to WinCE
- Allow constraint names on the DEFAULT clause in a table definition
- Added the ".bail" command to the command-line shell
- Make CSV (comma separate value) output from the command-line shell more closely aligned to accepted practice
- Experimental FTS2 module added
- Use sqlite3_mprintf() instead of strdup() to avoid libc dependencies
- VACUUM uses a temporary file in the official TEMP folder, not in the same directory as the original database
- The prefix on temporary filenames on Windows is changed from "sqlite" to "etilqs".

2006-10-09 (3.3.8)

- Support for full text search using the FTS1 module (beta)
- Added Mac OS X locking patches (beta - disabled by default)

- Introduce extended error codes and add error codes for various kinds of I/O errors.
- Added support for IF EXISTS on CREATE/DROP TRIGGER/VIEW
- Fix the regression test suite so that it works with Tcl8.5
- Enhance sqlite3_set_authorizer() to provide notification of calls to SQL functions.
- Added experimental API: sqlite3_auto_extension()
- Various minor bug fixes

2006-08-12 (3.3.7)

- Added support for virtual tables (beta)
- Added support for dynamically loaded extensions (beta)
- The sqlite3_interrupt() routine can be called for a different thread
- Added the MATCH operator.
- The default file format is now 1.

2006-06-06 (3.3.6)

- Plays better with virus scanners on Windows
- Faster :memory: databases
- Fix an obscure segfault in UTF-8 to UTF-16 conversions
- Added driver for OS/2
- Correct column meta-information returned for aggregate queries
- Enhanced output from EXPLAIN QUERY PLAN
- LIMIT 0 now works on subqueries
- Bug fixes and performance enhancements in the query optimizer
- Correctly handle NULL filenames in ATTACH and DETACH
- Improved syntax error messages in the parser
- Fix type coercion rules for the IN operator

2006-04-05 (3.3.5)

- CHECK constraints use conflict resolution algorithms correctly.
- The SUM() function throws an error on integer overflow.
- Choose the column names in a compound query from the left-most SELECT instead of the right-most.
- The sqlite3_create_collation() function honors the SQLITE_UTF16_ALIGNED flag.
- SQLITE_SECURE_DELETE compile-time option causes deletes to overwrite old data with zeros.
- Detect integer overflow in abs().
- The random() function provides 64 bits of randomness instead of only 32 bits.

- Parser detects and reports automaton stack overflow.
- Change the round() function to return REAL instead of TEXT.
- Allow WHERE clause terms on the left table of a LEFT OUTER JOIN to contain aggregate subqueries.
- Skip over leading spaces in text to numeric conversions.
- Various minor bug and documentation typo fixes and performance enhancements.

2006-02-11 (3.3.4)

- Fix a blunder in the Unix mutex implementation that can lead to deadlock on multithreaded systems.
- Fix an alignment problem on 64-bit machines
- Added the fullfsync pragma.
- Fix an optimizer bug that could have caused some unusual LEFT OUTER JOINs to give incorrect results.
- The SUM function detects integer overflow and converts to accumulating an approximate result using floating point numbers
- Host parameter names can begin with '@' for compatibility with SQL Server.
- Other miscellaneous bug fixes

2006-01-31 (3.3.3)

- Removed support for an ON CONFLICT clause on CREATE INDEX - it never worked correctly so this should not present any backward compatibility problems.
- Authorizer callback now notified of ALTER TABLE ADD COLUMN commands
- After any changes to the TEMP database schema, all prepared statements are invalidated and must be recreated using a new call to sqlite3_prepare()
- Other minor bug fixes in preparation for the first stable release of version 3.3

2006-01-24 (3.3.2 beta)

- Bug fixes and speed improvements. Improved test coverage.
- Changes to the OS-layer interface: mutexes must now be recursive.
- Discontinue the use of thread-specific data for out-of-memory exception handling

2006-01-16 (3.3.1 alpha)

- Countless bug fixes
- Speed improvements
- Database connections can now be used by multiple threads, not just the thread in which they were created.

2006-01-11 (3.3.0 alpha)

- CHECK constraints
- IF EXISTS and IF NOT EXISTS clauses on CREATE/DROP TABLE/INDEX.
- DESC indices
- More efficient encoding of boolean values resulting in smaller database files
- More aggressive SQLITE_OMIT_FLOATING_POINT
- Separate INTEGER and REAL affinity
- Added a virtual function layer for the OS interface
- "exists" method added to the TCL interface
- Improved response to out-of-memory errors
- Database cache can be optionally shared between connections in the same thread
- Optional READ UNCOMMITTED isolation (instead of the default isolation level of SERIALIZABLE) and table level locking when database connections share a common cache.

2005-12-19 (3.2.8)

- Fix an obscure bug that can cause database corruption under the following unusual circumstances: A large INSERT or UPDATE statement which is part of an even larger transaction fails due to a uniqueness constraint but the containing transaction commits.

2005-12-19 (2.8.17)

- Fix an obscure bug that can cause database corruption under the following unusual circumstances: A large INSERT or UPDATE statement which is part of an even larger transaction fails due to a uniqueness contraint but the containing transaction commits.

2005-09-24 (3.2.7)

- GROUP BY now considers NULLs to be equal again, as it should
- Now compiles on Solaris and OpenBSD and other Unix variants that lack the fdatasync() function
- Now compiles on MSVC++6 again
- Fix uninitialized variables causing malfunctions for various obscure queries
- Correctly compute a LEFT OUTER JOINs that is constrained on the left table only

2005-09-17 (3.2.6)

- Fix a bug that can cause database corruption if a VACUUM (or autovacuum) fails and is rolled back on a database that is larger than 1GiB

- LIKE optimization now works for columns with COLLATE NOCASE
- ORDER BY and GROUP BY now use bounded memory
- Added support for COUNT(DISTINCT expr)
- Change the way SUM() handles NULL values in order to comply with the SQL standard
- Use fdatasync() instead of fsync() where possible in order to speed up commits slightly
- Use of the CROSS keyword in a join turns off the table reordering optimization
- Added the experimental and undocumented EXPLAIN QUERY PLAN capability
- Use the unicode API in Windows

2005-08-27 (3.2.5)

- Fix a bug effecting DELETE and UPDATE statements that changed more than 40960 rows.
- Change the makefile so that it no longer requires GNUmake extensions
- Fix the --enable-threadsafe option on the configure script
- Fix a code generator bug that occurs when the left-hand side of an IN operator is constant and the right-hand side is a SELECT statement
- The PRAGMA synchronous=off statement now disables syncing of the master journal file in addition to the normal rollback journals

2005-08-24 (3.2.4)

- Fix a bug introduced in the previous release that can cause a segfault while generating code for complex WHERE clauses.
- Allow floating point literals to begin or end with a decimal point.

2005-08-21 (3.2.3)

- Added support for the CAST operator
- Tcl interface allows BLOB values to be transferred to user-defined functions
- Added the "transaction" method to the Tcl interface
- Allow the DEFAULT value of a column to call functions that have constant operands
- Added the ANALYZE command for gathering statistics on indices and using those statistics when picking an index in the optimizer
- Remove the limit (formerly 100) on the number of terms in the WHERE clause
- The right-hand side of the IN operator can now be a list of expressions instead of just a list of constants
- Rework the optimizer so that it is able to make better use of indices
- The order of tables in a join is adjusted automatically to make better use of indices
- The IN operator is now a candidate for optimization even if the left-hand side is not the

left-most term of the index. Multiple IN operators can be used with the same index.
- WHERE clause expressions using BETWEEN and OR are now candidates for optimization
- Added the "case_sensitive_like" pragma and the SQLITE_CASE_SENSITIVE_LIKE compile-time option to set its default value to "on".
- Use indices to help with GLOB expressions and LIKE expressions too when the case_sensitive_like pragma is enabled
- Added support for grave-accent quoting for compatibility with MySQL
- Improved test coverage
- Dozens of minor bug fixes

2005-06-12 (3.2.2)

- Added the sqlite3_db_handle() API
- Added the sqlite3_get_autocommit() API
- Added a REGEXP operator to the parser. There is no function to back up this operator in the standard build but users can add their own using sqlite3_create_function()
- Speed improvements and library footprint reductions.
- Fix byte alignment problems on 64-bit architectures.
- Many, many minor bug fixes and documentation updates.

2005-03-29 (3.2.1)

- Fix a memory allocation error in the new ADD COLUMN comment.
- Documentation updates

2005-03-21 (3.2.0)

- Added support for ALTER TABLE ADD COLUMN.
- Added support for the "T" separator in ISO-8601 date/time strings.
- Improved support for Cygwin.
- Numerous bug fixes and documentation updates.

2005-03-17 (3.1.6)

- Fix a bug that could cause database corruption when inserting record into tables with around 125 columns.
- sqlite3_step() is now much more likely to invoke the busy handler and less likely to return SQLITE_BUSY.
- Fix memory leaks that used to occur after a malloc() failure.

2005-03-11 (3.1.5)

- The ioctl on Mac OS X to control syncing to disk is F_FULLFSYNC, not F_FULLSYNC. The previous release had it wrong.

2005-03-11 (3.1.4)

- Fix a bug in autovacuum that could cause database corruption if a CREATE UNIQUE INDEX fails because of a constraint violation. This problem only occurs if the new autovacuum feature introduced in version 3.1 is turned on.
- The F_FULLSYNC ioctl (currently only supported on Mac OS X) is disabled if the synchronous pragma is set to something other than "full".
- Add additional forward compatibility to the future version 3.2 database file format.
- Fix a bug in WHERE clauses of the form (rowid<'2')< li="">
- New SQLITE*OMIT*... compile-time options added
- Updates to the man page
- Remove the use of strcasecmp() from the shell
- Windows DLL exports symbols Tclsqlite_Init and Sqlite_Init

2005-02-19 (3.1.3)

- Fix a problem with VACUUM on databases from which tables containing AUTOINCREMENT have been dropped.
- Add forward compatibility to the future version 3.2 database file format.
- Documentation updates

2005-02-15 (3.1.2)

- Fix a bug that can lead to database corruption if there are two open connections to the same database and one connection does a VACUUM and the second makes some change to the database.
- Allow "?" parameters in the LIMIT clause.
- Fix VACUUM so that it works with AUTOINCREMENT.
- Fix a race condition in AUTOVACUUM that can lead to corrupt databases
- Add a numeric version number to the sqlite3.h include file.
- Other minor bug fixes and performance enhancements.

2005-02-15 (2.8.16)

- Fix a bug that can lead to database corruption if there are two open connections to the same database and one connection does a VACUUM and the second makes some change to the database.

- Correctly handle quoted names in CREATE INDEX statements.
- Fix a naming conflict between sqlite.h and sqlite3.h.
- Avoid excess heap usage when copying expressions.
- Other minor bug fixes.

2005-02-01 (3.1.1 BETA)

- Automatic caching of prepared statements in the TCL interface
- ATTACH and DETACH as well as some other operations cause existing prepared statements to expire.
- Numerous minor bug fixes

2005-01-21 (3.1.0 ALPHA)

- Autovacuum support added
- CURRENT_TIME, CURRENT_DATE, and CURRENT_TIMESTAMP added
- Support for the EXISTS clause added.
- Support for correlated subqueries added.
- Added the ESCAPE clause on the LIKE operator.
- Support for ALTER TABLE ... RENAME TABLE ... added
- AUTOINCREMENT keyword supported on INTEGER PRIMARY KEY
- Many SQLITE*OMIT* macros inserts to omit features at compile-time and reduce the library footprint.
- The REINDEX command was added.
- The engine no longer consults the main table if it can get all the information it needs from an index.
- Many nuisance bugs fixed.

2004-10-12 (3.0.8)

- Add support for DEFERRED, IMMEDIATE, and EXCLUSIVE transactions.
- Allow new user-defined functions to be created when there are already one or more precompiled SQL statements.
- Fix portability problems for MinGW/MSYS.
- Fix a byte alignment problem on 64-bit Sparc machines.
- Fix the ".import" command of the shell so that it ignores \r characters at the end of lines.
- The "csv" mode option in the shell puts strings inside double-quotes.
- Fix typos in documentation.

- Convert array constants in the code to have type "const".
- Numerous code optimizations, specially optimizations designed to make the code footprint smaller.

2004-09-18 (3.0.7)

- The BTree module allocates large buffers using malloc() instead of off of the stack, in order to play better on machines with limited stack space.
- Fixed naming conflicts so that versions 2.8 and 3.0 can be linked and used together in the same ANSI-C source file.
- New interface: sqlite3_bind_parameter_index()
- Add support for wildcard parameters of the form: "?nnn"
- Fix problems found on 64-bit systems.
- Removed encode.c file (containing unused routines) from the version 3.0 source tree.
- The sqlite3_trace() callbacks occur before each statement is executed, not when the statement is compiled.
- Makefile updates and miscellaneous bug fixes.

2004-09-02 (3.0.6 beta)

- Better detection and handling of corrupt database files.
- The sqlite3_step() interface returns SQLITE_BUSY if it is unable to commit a change because of a lock
- Combine the implementations of LIKE and GLOB into a single pattern-matching subroutine.
- Miscellaneous code size optimizations and bug fixes

2004-08-29 (3.0.5 beta)

- Support for ":AAA" style bind parameter names.
- Added the new sqlite3_bind_parameter_name() interface.
- Support for TCL variable names embedded in SQL statements in the TCL bindings.
- The TCL bindings transfer data without necessarily doing a conversion to a string.
- The database for TEMP tables is not created until it is needed.
- Add the ability to specify an alternative temporary file directory using the "sqlite_temp_directory" global variable.
- A compile-time option (SQLITE_BUSY_RESERVED_LOCK) causes the busy handler to be called when there is contention for a RESERVED lock.
- Various bug fixes and optimizations

2004-08-09 (3.0.4 beta)

- CREATE TABLE and DROP TABLE now work correctly as prepared statements.
- Fix a bug in VACUUM and UNIQUE indices.
- Add the ".import" command to the command-line shell.
- Fix a bug that could cause index corruption when an attempt to delete rows of a table is blocked by a pending query.
- Library size optimizations.
- Other minor bug fixes.

2004-07-22 (2.8.15)

- This is a maintenance release only. Various minor bugs have been fixed and some portability enhancements are added.

2004-07-22 (3.0.3 beta)

- The second beta release for SQLite 3.0.
- Add support for "PRAGMA page_size" to adjust the page size of the database.
- Various bug fixes and documentation updates.

2004-06-30 (3.0.2 beta)

- The first beta release for SQLite 3.0.

2004-06-22 (3.0.1 alpha)

- * Alpha Release - Research And Testing Use Only *
- Lots of bug fixes.

2004-06-18 (3.0.0 alpha)

- * Alpha Release - Research And Testing Use Only *
- Support for internationalization including UTF-8, UTF-16, and user defined collating sequences.
- New file format that is 25% to 35% smaller for typical use.
- Improved concurrency.
- Atomic commits for ATTACHed databases.
- Remove cruft from the APIs.
- BLOB support.
- 64-bit rowids.

- More information.

2004-06-09 (2.8.14)

- Fix the min() and max() optimizer so that it works when the FROM clause consists of a subquery.
- Ignore extra whitespace at the end of of "." commands in the shell.
- Bundle sqlite_encode_binary() and sqlite_decode_binary() with the library.
- The TEMP_STORE and DEFAULT_TEMP_STORE pragmas now work.
- Code changes to compile cleanly using OpenWatcom.
- Fix VDBE stack overflow problems with INSTEAD OF triggers and NULLs in IN operators.
- Add the global variable sqlite_temp_directory which if set defines the directory in which temporary files are stored.
- sqlite_interrupt() plays well with VACUUM.
- Other minor bug fixes.

2004-03-08 (2.8.13)

- Refactor parts of the code in order to make the code footprint smaller. The code is now also a little bit faster.
- sqlite_exec() is now implemented as a wrapper around sqlite_compile() and sqlite_step().
- The built-in min() and max() functions now honor the difference between NUMERIC and TEXT datatypes. Formerly, min() and max() always assumed their arguments were of type NUMERIC.
- New HH:MM:SS modifier to the built-in date/time functions.
- Experimental sqlite_last_statement_changes() API added. Fixed the last_insert_rowid() function so that it works correctly with triggers.
- Add functions prototypes for the database encryption API.
- Fix several nuisance bugs.

2004-02-08 (2.8.12)

- Fix a bug that will might corrupt the rollback journal if a power failure or external program halt occurs in the middle of a COMMIT. The corrupt journal can lead to database corruption when it is rolled back.
- Reduce the size and increase the speed of various modules, especially the virtual machine.
- Allow "<expr> IN <table>" as a shorthand for "<expr> IN (SELECT * FROM <table>".

- Optimizations to the sqlite_mprintf() routine.
- Make sure the MIN() and MAX() optimizations work within subqueries.

2004-01-14 (2.8.11)

- Fix a bug in how the IN operator handles NULLs in subqueries. The bug was introduced by the previous release.

2004-01-14 (2.8.10)

- Fix a potential database corruption problem on Unix caused by the fact that all POSIX advisory locks are cleared whenever you close() a file. The work around it to embargo all close() calls while locks are outstanding.
- Performance enhancements on some corner cases of COUNT(*).
- Make sure the in-memory backend response sanely if malloc() fails.
- Allow sqlite_exec() to be called from within user-defined SQL functions.
- Improved accuracy of floating-point conversions using "long double".
- Bug fixes in the experimental date/time functions.

2004-01-06 (2.8.9)

- Fix a 32-bit integer overflow problem that could result in corrupt indices in a database if large negative numbers (less than -2147483648) were inserted into an indexed numeric column.
- Fix a locking problem on multi-threaded Linux implementations.
- Always use "." instead of "," as the decimal point even if the locale requests ",".
- Added UTC to localtime conversions to the experimental date/time functions.
- Bug fixes to date/time functions.

2003-12-18 (2.8.8)

- Fix a critical bug introduced into 2.8.0 which could cause database corruption.
- Fix a problem with 3-way joins that do not use indices
- The VACUUM command now works with the non-callback API
- Improvements to the "PRAGMA integrity_check" command

2003-12-04 (2.8.7)

- Added experimental sqlite_bind() and sqlite_reset() APIs.
- If the name of the database is an empty string, open a new database in a temporary file that is automatically deleted when the database is closed.

- Performance enhancements in the lemon-generated parser
- Experimental date/time functions revised.
- Disallow temporary indices on permanent tables.
- Documentation updates and typo fixes
- Added experimental sqlite_progress_handler() callback API
- Removed support for the Oracle8 outer join syntax.
- Allow GLOB and LIKE operators to work as functions.
- Other minor documentation and makefile changes and bug fixes.

2003-08-22 (2.8.6)

- Moved the CVS repository to www.sqlite.org
- Update the NULL-handling documentation.
- Experimental date/time functions added.
- Bug fix: correctly evaluate a view of a view without segfaulting.
- Bug fix: prevent database corruption if you dropped a trigger that had the same name as a table.
- Bug fix: allow a VACUUM (without segfaulting) on an empty database after setting the EMPTY_RESULT_CALLBACKS pragma.
- Bug fix: if an integer value will not fit in a 32-bit int, store it in a double instead.
- Bug fix: Make sure the journal file directory entry is committed to disk before writing the database file.

2003-07-22 (2.8.5)

- Make LIMIT work on a compound SELECT statement.
- LIMIT 0 now shows no rows. Use LIMIT -1 to see all rows.
- Correctly handle comparisons between an INTEGER PRIMARY KEY and a floating point number.
- Fix several important bugs in the new ATTACH and DETACH commands.
- Updated the NULL-handling document.
- Allow NULL arguments in sqlite_compile() and sqlite_step().
- Many minor bug fixes

2003-06-29 (2.8.4)

- Enhanced the "PRAGMA integrity_check" command to verify indices.
- Added authorization hooks for the new ATTACH and DETACH commands.
- Many documentation updates
- Many minor bug fixes

2003-06-04 (2.8.3)

- Fix a problem that will corrupt the indices on a table if you do an INSERT OR REPLACE or an UPDATE OR REPLACE on a table that contains an INTEGER PRIMARY KEY plus one or more indices.
- Fix a bug in Windows locking code so that locks work correctly when simultaneously accessed by Win95 and WinNT systems.
- Add the ability for INSERT and UPDATE statements to refer to the "rowid" (or "*rowid*" or "oid") columns.
- Other important bug fixes

2003-05-17 (2.8.2)

- Fix a problem that will corrupt the database file if you drop a table from the main database that has a TEMP index.

2003-05-17 (2.8.1)

- Reactivated the VACUUM command that reclaims unused disk space in a database file.
- Added the ATTACH and DETACH commands to allow interacting with multiple database files at the same time.
- Added support for TEMP triggers and indices.
- Added support for in-memory databases.
- Removed the experimental sqlite_open_aux_file(). Its function is subsumed in the new ATTACH command.
- The precedence order for ON CONFLICT clauses was changed so that ON CONFLICT clauses on BEGIN statements have a higher precedence than ON CONFLICT clauses on constraints.
- Many, many bug fixes and compatibility enhancements.

2003-02-16 (2.8.0)

- Modified the journal file format to make it more resistant to corruption that can occur after an OS crash or power failure.
- Added a new C/C++ API that does not use callback for returning data.

2003-01-25 (2.7.6)

- Performance improvements. The library is now much faster.
- Added the **sqlite_set_authorizer()** API. Formal documentation has not been written - see the source code comments for instructions on how to use this function.

- Fix a bug in the GLOB operator that was preventing it from working with upper-case letters.
- Various minor bug fixes.

2002-12-28 (2.7.5)

- Fix an uninitialized variable in pager.c which could (with a probability of about 1 in 4 billion) result in a corrupted database.

2002-12-17 (2.7.4)

- Database files can now grow to be up to 2^41 bytes. The old limit was 2^31 bytes.
- The optimizer will now scan tables in the reverse if doing so will satisfy an ORDER BY ... DESC clause.
- The full pathname of the database file is now remembered even if a relative path is passed into sqlite_open(). This allows the library to continue operating correctly after a chdir().
- Speed improvements in the VDBE.
- Lots of little bug fixes.

2002-10-31 (2.7.3)

- Various compiler compatibility fixes.
- Fix a bug in the "expr IN ()" operator.
- Accept column names in parentheses.
- Fix a problem with string memory management in the VDBE
- Fix a bug in the "table_info" pragma"
- Export the sqlite_function_type() API function in the Windows DLL
- Fix locking behavior under Windows
- Fix a bug in LEFT OUTER JOIN

2002-09-25 (2.7.2)

- Prevent journal file overflows on huge transactions.
- Fix a memory leak that occurred when sqlite_open() failed.
- Honor the ORDER BY and LIMIT clause of a SELECT even if the result set is used for an INSERT.
- Do not put write locks on the file used to hold TEMP tables.
- Added documentation on SELECT DISTINCT and on how SQLite handles NULLs.
- Fix a problem that was causing poor performance when many thousands of SQL statements were executed by a single sqlite_exec() call.

2002-08-31 (2.7.1)

- Fix a bug in the ORDER BY logic that was introduced in version 2.7.0
- C-style comments are now accepted by the tokenizer.
- INSERT runs a little faster when the source is a SELECT statement.

2002-08-25 (2.7.0)

- Make a distinction between numeric and text values when sorting. Text values sort according to memcmp(). Numeric values sort in numeric order.
- Allow multiple simultaneous readers under Windows by simulating the reader/writers locks that are missing from Win95/98/ME.
- An error is now returned when trying to start a transaction if another transaction is already active.

2002-08-13 (2.6.3)

- Add the ability to read both little-endian and big-endian databases. So a database created under SunOS or Mac OS X can be read and written under Linux or Windows and vice versa.
- Convert to the new website: http://www.sqlite.org/
- Allow transactions to span Linux Threads
- Bug fix in the processing of the ORDER BY clause for GROUP BY queries

2002-07-31 (2.6.2)

- Text files read by the COPY command can now have line terminators of LF, CRLF, or CR.
- SQLITE_BUSY is handled correctly if encountered during database initialization.
- Fix to UPDATE triggers on TEMP tables.
- Documentation updates.

2002-07-19 (2.6.1)

- Include a static string in the library that responds to the RCS "ident" command and which contains the library version number.
- Fix an assertion failure that occurred when deleting all rows of a table with the "count_changes" pragma turned on.
- Better error reporting when problems occur during the automatic 2.5.6 to 2.6.0 database format upgrade.

2002-07-18 (2.6.0)

- Change the format of indices to correct a design flaw the originated with version 2.1.0. ***This is an incompatible file format change*** When version 2.6.0 or later of the library attempts to open a database file created by version 2.5.6 or earlier, it will automatically and irreversibly convert the file format. **Make backup copies of older database files before opening them with version 2.6.0 of the library.**

2002-07-07 (2.5.6)

- Fix more problems with rollback. Enhance the test suite to exercise the rollback logic extensively in order to prevent any future problems.

2002-07-06 (2.5.5)

- Fix a bug which could cause database corruption during a rollback. This bugs was introduced in version 2.4.0 by the freelist optimization of checkin [410].
- Fix a bug in aggregate functions for VIEWs.
- Other minor changes and enhancements.

2002-07-01 (2.5.4)

- Make the "AS" keyword optional again.
- The datatype of columns now appear in the 4th argument to the callback.
- Added the **sqlite_open_aux_file()** API, though it is still mostly undocumented and untested.
- Added additional test cases and fixed a few bugs that those test cases found.

2002-06-25 (2.5.3)

- Bug fix: Database corruption can occur due to the optimization that was introduced in version 2.4.0 (check-in [410]). The problem should now be fixed. The use of versions 2.4.0 through 2.5.2 is not recommended.

2002-06-25 (2.5.2)

- Added the new **SQLITE_TEMP_MASTER** table which records the schema for temporary tables in the same way that **SQLITE_MASTER** does for persistent tables.
- Added an optimization to UNION ALL
- Fixed a bug in the processing of LEFT OUTER JOIN
- The LIMIT clause now works on subselects

- ORDER BY works on subselects
- There is a new TypeOf() function used to determine if an expression is numeric or text.
- Autoincrement now works for INSERT from a SELECT.

2002-06-19 (2.5.1)

- The query optimizer now attempts to implement the ORDER BY clause using an index. Sorting is still used if not suitable index is available.

2002-06-17 (2.5.0)

- Added support for row triggers.
- Added SQL-92 compliant handling of NULLs.
- Add support for the full SQL-92 join syntax and LEFT OUTER JOINs.
- Double-quoted strings interpreted as column names not text literals.
- Parse (but do not implement) foreign keys.
- Performance improvements in the parser, pager, and WHERE clause code generator.
- Make the LIMIT clause work on subqueries. (ORDER BY still does not work, though.)
- Added the "%Q" expansion to sqlite_*_printf().
- Bug fixes too numerous to mention (see the change log).

2002-05-10 (2.4.12)

- Added logic to detect when the library API routines are called out of sequence.

2002-05-08 (2.4.11)

- Bug fix: Column names in the result set were not being generated correctly for some (rather complex) VIEWs. This could cause a segfault under certain circumstances.

2002-05-03 (2.4.10)

- Bug fix: Generate correct column headers when a compound SELECT is used as a subquery.
- Added the sqlite_encode_binary() and sqlite_decode_binary() functions to the source tree. But they are not yet linked into the library.
- Documentation updates.
- Export the sqlite_changes() function from Windows DLLs.
- Bug fix: Do not attempt the subquery flattening optimization on queries that lack a FROM clause. To do so causes a segfault.

2002-04-22 (2.4.9)

- Fix a bug that was causing the precompiled binary of SQLITE.EXE to report "out of memory" under Windows 98.

2002-04-20 (2.4.8)

- Make sure VIEWs are created after their corresponding TABLEs in the output of the **.dump** command in the shell.
- Speed improvements: Do not do synchronous updates on TEMP tables.
- Many improvements and enhancements to the shell.
- Make the GLOB and LIKE operators functions that can be overridden by a programmer. This allows, for example, the LIKE operator to be changed to be case sensitive.

2002-04-12 (2.4.7)

- Add the ability to put TABLE.* in the column list of a SELECT statement.
- Permit SELECT statements without a FROM clause.
- Added the **last_insert_rowid()** SQL function.
- Do not count rows where the IGNORE conflict resolution occurs in the row count.
- Make sure functions expressions in the VALUES clause of an INSERT are correct.
- Added the **sqlite_changes()** API function to return the number of row that changed in the most recent operation.

2002-04-02 (2.4.6)

- Bug fix: Correctly handle terms in the WHERE clause of a join that do not contain a comparison operator.

2002-04-02 (2.4.5)

- Bug fix: Correctly handle functions that appear in the WHERE clause of a join.
- When the PRAGMA vdbe_trace=ON is set, correctly print the P3 operand value when it is a pointer to a structure rather than a pointer to a string.
- When inserting an explicit NULL into an INTEGER PRIMARY KEY, convert the NULL value into a unique key automatically.

2002-03-30 (2.4.4)

- Allow "VIEW" to be a column name
- Added support for CASE expressions (patch from Dan Kennedy)

- Added RPMS to the delivery (patches from Doug Henry)
- Fix typos in the documentation
- Cut over configuration management to a new CVS repository with its own CVSTrac bug tracking system.

2002-03-23 (2.4.3)

- Fix a bug in SELECT that occurs when a compound SELECT is used as a subquery in the FROM of a SELECT.
- The **sqlite_get_table()** function now returns an error if you give it two or more SELECTs that return different numbers of columns.

2002-03-20 (2.4.2)

- Bug fix: Fix an assertion failure that occurred when ROWID was a column in a SELECT statement on a view.
- Bug fix: Fix an uninitialized variable in the VDBE that would could an assert failure.
- Make the os.h header file more robust in detecting when the compile is for Windows and when it is for Unix.

2002-03-13 (2.4.1)

- Using an unnamed subquery in a FROM clause would cause a segfault.
- The parser now insists on seeing a semicolon or the end of input before executing a statement. This avoids an accidental disaster if the WHERE keyword is misspelled in an UPDATE or DELETE statement.

2002-03-11 (2.4.0)

- Change the name of the sanity_check PRAGMA to **integrity_check** and make it available in all compiles.
- SELECT min() or max() of an indexed column with no WHERE or GROUP BY clause is handled as a special case which avoids a complete table scan.
- Automatically generated ROWIDs are now sequential.
- Do not allow dot-commands of the command-line shell to occur in the middle of a real SQL command.
- Modifications to the "lemon" parser generator so that the parser tables are 4 times smaller.
- Added support for user-defined functions implemented in C.
- Added support for new functions: **coalesce()**, **lower()**, **upper()**, and **random()**
- Added support for VIEWs.

- Added the subquery flattening optimizer.
- Modified the B-Tree and Pager modules so that disk pages that do not contain real data (free pages) are not journaled and are not written from memory back to the disk when they change. This does not impact database integrity, since the pages contain no real data, but it does make large INSERT operations about 2.5 times faster and large DELETEs about 5 times faster.
- Made the CACHE_SIZE pragma persistent
- Added the SYNCHRONOUS pragma
- Fixed a bug that was causing updates to fail inside of transactions when the database contained a temporary table.

2002-02-19 (2.3.3)

- Allow identifiers to be quoted in square brackets, for compatibility with MS-Access.
- Added support for sub-queries in the FROM clause of a SELECT.
- More efficient implementation of sqliteFileExists() under Windows. (by Joel Luscy)
- The VALUES clause of an INSERT can now contain expressions, including scalar SELECT clauses.
- Added support for CREATE TABLE AS SELECT
- Bug fix: Creating and dropping a table all within a single transaction was not working.

2002-02-14 (2.3.2)

- Bug fix: There was an incorrect assert() in pager.c. The real code was all correct (as far as is known) so everything should work OK if you compile with -DNDEBUG=1. When asserts are not disabled, there could be a fault.

2002-02-13 (2.3.1)

- Bug fix: An assertion was failing if "PRAGMA full_column_names=ON;" was set and you did a query that used a rowid, like this: "SELECT rowid, * FROM ...".

2002-02-03 (2.3.0)

- Fix a serious bug in the INSERT command which was causing data to go into the wrong columns if the data source was a SELECT and the INSERT clauses specified its columns in some order other than the default.
- Added the ability to resolve constraint conflicts is ways other than an abort and rollback. See the documentation on the "ON CONFLICT" clause for details.
- Temporary files are now automatically deleted by the operating system when closed. There are no more dangling temporary files on a program crash. (If the OS crashes,

fsck will delete the file after reboot under Unix. I do not know what happens under Windows.)
- NOT NULL constraints are honored.
- The COPY command puts NULLs in columns whose data is '\N'.
- In the COPY command, backslash can now be used to escape a newline.
- Added the SANITY_CHECK pragma.

2002-01-28 (2.2.5)

- Important bug fix: the IN operator was not working if either the left-hand or right-hand side was derived from an INTEGER PRIMARY KEY.
- Do not escape the backslash '\' character in the output of the **sqlite** command-line access program.

2002-01-22 (2.2.4)

- The label to the right of an AS in the column list of a SELECT can now be used as part of an expression in the WHERE, ORDER BY, GROUP BY, and/or HAVING clauses.
- Fix a bug in the **-separator** command-line option to the **sqlite** command.
- Fix a problem with the sort order when comparing upper-case strings against characters greater than 'Z' but less than 'a'.
- Report an error if an ORDER BY or GROUP BY expression is constant.

2002-01-16 (2.2.3)

- Fix warning messages in VC++ 7.0. (Patches from nicolas352001)
- Make the library thread-safe. (The code is there and appears to work but has not been stressed.)
- Added the new **sqlite_last_insert_rowid()** API function.

2002-01-14 (2.2.2)

- Bug fix: An assertion was failing when a temporary table with an index had the same name as a permanent table created by a separate process.
- Bug fix: Updates to tables containing an INTEGER PRIMARY KEY and an index could fail.

2002-01-09 (2.2.1)

- Bug fix: An attempt to delete a single row of a table with a WHERE clause of "ROWID=x" when no such rowid exists was causing an error.

- Bug fix: Passing in a NULL as the 3rd parameter to **sqlite_open()** would sometimes cause a coredump.
- Bug fix: DROP TABLE followed by a CREATE TABLE with the same name all within a single transaction was causing a coredump.
- Makefile updates from A. Rottmann

2001-12-22 (2.2.0)

- Columns of type INTEGER PRIMARY KEY are actually used as the primary key in underlying B-Tree representation of the table.
- Several obscure, unrelated bugs were found and fixed while implemented the integer primary key change of the previous bullet.
- Added the ability to specify "*" as part of a larger column list in the result section of a SELECT statement. For example: <nobr>"**SELECT rowid,* FROM table1;**"</nobr>.
- Updates to comments and documentation.

2001-12-15 (2.1.7)

- Fix a bug in **CREATE TEMPORARY TABLE** which was causing the table to be initially allocated in the main database file instead of in the separate temporary file. This bug could cause the library to suffer an assertion failure and it could cause "page leaks" in the main database file.
- Fix a bug in the b-tree subsystem that could sometimes cause the first row of a table to be repeated during a database scan.

2001-12-14 (2.1.6)

- Fix the locking mechanism yet again to prevent **sqlite_exec()** from returning SQLITE_PROTOCOL unnecessarily. This time the bug was a race condition in the locking code. This change affects both POSIX and Windows users.

2001-12-06 (2.1.5)

- Fix for another problem (unrelated to the one fixed in 2.1.4) that sometimes causes **sqlite_exec()** to return SQLITE_PROTOCOL unnecessarily. This time the bug was in the POSIX locking code and should not effect Windows users.

2001-12-05 (2.1.4)

- Sometimes **sqlite_exec()** would return SQLITE_PROTOCOL when it should have returned SQLITE_BUSY.

- The fix to the previous bug uncovered a deadlock which was also fixed.
- Add the ability to put a single .command in the second argument of the sqlite shell
- Updates to the FAQ

2001-11-24 (2.1.3)

- Fix the behavior of comparison operators (ex: "**<**", "**==**", etc.) so that they are consistent with the order of entries in an index.
- Correct handling of integers in SQL expressions that are larger than what can be represented by the machine integer.

2001-11-23 (2.1.2)

- Changes to support 64-bit architectures.
- Fix a bug in the locking protocol.
- Fix a bug that could (rarely) cause the database to become unreadable after a DROP TABLE due to corruption to the SQLITE_MASTER table.
- Change the code so that version 2.1.1 databases that were rendered unreadable by the above bug can be read by this version of the library even though the SQLITE_MASTER table is (slightly) corrupted.

2001-11-13 (2.1.1)

- Bug fix: Sometimes arbitrary strings were passed to the callback function when the actual value of a column was NULL.

2001-11-12 (2.1.0)

- Change the format of data records so that records up to 16MB in size can be stored.
- Change the format of indices to allow for better query optimization.
- Implement the "LIMIT ... OFFSET ..." clause on SELECT statements.

2001-11-03 (2.0.8)

- Made selected parameters in API functions **const**. This should be fully backwards compatible.
- Documentation updates
- Simplify the design of the VDBE by restricting the number of sorters and lists to 1. In practice, no more than one sorter and one list was ever used anyhow.

2001-10-22 (2.0.7)

- Any UTF-8 character or ISO8859 character can be used as part of an identifier.
- Patches from Christian Werner to improve ODBC compatibility and to fix a bug in the round() function.
- Plug some memory leaks that use to occur if malloc() failed. We have been and continue to be memory leak free as long as malloc() works.
- Changes to some test scripts so that they work on Windows in addition to Unix.

2001-10-19 (2.0.6)

- Added the EMPTY_RESULT_CALLBACKS pragma
- Support for UTF-8 and ISO8859 characters in column and table names.
- Bug fix: Compute correct table names with the FULL_COLUMN_NAMES pragma is turned on.

2001-10-15 (2.0.5)

- Added the COUNT_CHANGES pragma.
- Changes to the FULL_COLUMN_NAMES pragma to help out the ODBC driver.
- Bug fix: "SELECT count(*)" was returning NULL for empty tables. Now it returns 0.

2001-10-13 (2.0.4)

- Bug fix: an obscure and relatively harmless bug was causing one of the tests to fail when gcc optimizations are turned on. This release fixes the problem.

2001-10-13 (2.0.3)

- Bug fix: the **sqlite_busy_timeout()** function was delaying 1000 times too long before failing.
- Bug fix: an assertion was failing if the disk holding the database file became full or stopped accepting writes for some other reason. New tests were added to detect similar problems in the future.
- Added new operators: **&** (bitwise-and) **|** (bitwise-or), **~** (ones-complement), **<<** (shift left), **>>** (shift right).
- Added new functions: **round()** and **abs()**.

2001-10-09 (2.0.2)

- Fix two bugs in the locking protocol. (One was masking the other.)

- Removed some unused "#include <unistd.h>" that were causing problems for VC++. </unistd.h>
- Fixed **sqlite.h** so that it is usable from C++
- Added the FULL_COLUMN_NAMES pragma. When set to "ON", the names of columns are reported back as TABLE.COLUMN instead of just COLUMN.
- Added the TABLE_INFO() and INDEX_INFO() pragmas to help support the ODBC interface.
- Added support for TEMPORARY tables and indices.

2001-10-02 (2.0.1)

- Remove some C++ style comments from btree.c so that it will compile using compilers other than gcc.
- The ".dump" output from the shell does not work if there are embedded newlines anywhere in the data. This is an old bug that was carried forward from version 1.0. To fix it, the ".dump" output no longer uses the COPY command. It instead generates INSERT statements.
- Extend the expression syntax to support "expr NOT NULL" (with a space between the "NOT" and the "NULL") in addition to "expr NOTNULL" (with no space).

200-09-28 (2.0.0)

- Automatically build binaries for Linux and Windows and put them on the website.

2001-09-28 (2.0-alpha-4)

- Incorporate makefile patches form A. Rottmann to use LIBTOOL

2001-09-27 (2.0-alpha-3)

- SQLite now honors the UNIQUE keyword in CREATE UNIQUE INDEX. Primary keys are required to be unique.
- File format changed back to what it was for alpha-1
- Fixes to the rollback and locking behavior

2001-09-20 (2.0-alpha-2)

- Initial release of version 2.0. The idea of renaming the library to "SQLus" was abandoned in favor of keeping the "SQLite" name and bumping the major version number.
- The pager and btree subsystems added back. They are now the only available

backend.
- The Dbbe abstraction and the GDBM and memory drivers were removed.
- Copyright on all code was disclaimed. The library is now in the public domain.

2001-07-23 (1.0.32)

- Pager and btree subsystems removed. These will be used in a follow-on SQL server library named "SQLus".
- Add the ability to use quoted strings as table and column names in expressions.

2001-04-15 (1.0.31)

- Pager subsystem added but not yet used.
- More robust handling of out-of-memory errors.
- New tests added to the test suite.

2001-04-06 (1.0.30)

- Remove the **sqlite_encoding** TCL variable that was introduced in the previous version.
- Add options **-encoding** and **-tcl-uses-utf** to the **sqlite** TCL command.
- Add tests to make sure that tclsqlite was compiled using Tcl header files and libraries that match.

2001-04-05 (1.0.29)

- The library now assumes data is stored as UTF-8 if the --enable-utf8 option is given to configure. The default behavior is to assume iso8859-x, as it has always done. This only makes a difference for LIKE and GLOB operators and the LENGTH and SUBSTR functions.
- If the library is not configured for UTF-8 and the Tcl library is one of the newer ones that uses UTF-8 internally, then a conversion from UTF-8 to iso8859 and back again is done inside the TCL interface.

2001-04-04 (1.0.28)

- Added limited support for transactions. At this point, transactions will do table locking on the GDBM backend. There is no support (yet) for rollback or atomic commit.
- Added special column names ROWID, OID, and *ROWID* that refer to the unique random integer key associated with every row of every table.
- Additional tests added to the regression suite to cover the new ROWID feature and the TCL interface bugs mentioned below.

- Changes to the "lemon" parser generator to help it work better when compiled using MSVC.
- Bug fixes in the TCL interface identified by Oleg Oleinick.

2001-03-20 (1.0.27)

- When doing DELETE and UPDATE, the library used to write the record numbers of records to be deleted or updated into a temporary file. This is changed so that the record numbers are held in memory.
- The DELETE command without a WHILE clause just removes the database files from the disk, rather than going through and deleting record by record.

2001-03-20 (1.0.26)

- A serious bug fixed on Windows. Windows users should upgrade. No impact to Unix.

2001-03-15 (1.0.25)

- Modify the test scripts to identify tests that depend on system load and processor speed and to warn the user that a failure of one of those (rare) tests does not necessarily mean the library is malfunctioning. No changes to code.

2001-03-14 (1.0.24)

- Fix a bug which was causing the UPDATE command to fail on systems where "malloc(0)" returns NULL. The problem does not appear on Windows, Linux, or HPUX but does cause the library to fail on QNX.

2001-02-20 (1.0.23)

- An unrelated (and minor) bug from Mark Muranwski fixed. The algorithm for figuring out where to put temporary files for a "memory:" database was not working quite right.

2001-02-19 (1.0.22)

- The previous fix was not quite right. This one seems to work better.

2001-02-19 (1.0.21)

- The UPDATE statement was not working when the WHERE clause contained some terms that could be satisfied using indices and other terms that could not. Fixed.

2001-02-11 (1.0.20)

- Merge development changes into the main trunk. Future work toward using a BTree file structure will use a separate CVS source tree. This CVS tree will continue to support the GDBM version of SQLite only.

2001-02-06 (1.0.19)

- Fix a strange (but valid) C declaration that was causing problems for QNX. No logical changes.

2001-01-04 (1.0.18)

- Print the offending SQL statement when an error occurs.
- Do not require commas between constraints in CREATE TABLE statements.
- Added the "-echo" option to the shell.
- Changes to comments.

2000-12-10 (1.0.17)

- Rewrote **sqlite_complete()** to make it faster.
- Minor tweaks to other code to make it run a little faster.
- Added new tests for **sqlite_complete()** and for memory leaks.

2000-11-28 (1.0.16)

- Documentation updates. Mostly fixing of typos and spelling errors.

2000-10-23 (1.0.15)

- Documentation updates
- Some sanity checking code was removed from the inner loop of vdbe.c to help the library to run a little faster. The code is only removed if you compile with -DNDEBUG.

2000-10-19 (1.0.14)

- Added a "memory:" backend driver that stores its database in an in-memory hash table.

2000-10-19 (1.0.13)

- Break out the GDBM driver into a separate file in anticipation to added new drivers.

- Allow the name of a database to be prefixed by the driver type. For now, the only driver type is "gdbm:".

2000-10-17 (1.0.12)

- Fixed an off-by-one error that was causing a coredump in the '%q' format directive of the new **sqlite_..._printf()** routines.
- Added the **sqlite_interrupt()** interface.
- In the shell, **sqlite_interrupt()** is invoked when the user presses Control-C
- Fixed some instances where **sqlite_exec()** was returning the wrong error code.

2000-10-11 (1.0.10)

- Added notes on how to compile for Windows95/98.
- Removed a few variables that were not being used. Etc.

2000-10-09 (1.0.9)

- Added the **sqlite_..._printf()** interface routines.
- Modified the **sqlite** shell program to use the new interface routines.
- Modified the **sqlite** shell program to print the schema for the built-in SQLITE_MASTER table, if explicitly requested.

2000-09-30 (1.0.8)

- Begin writing documentation on the TCL interface.

2000-09-29 (Not Released)

- Added the **sqlite_get_table()** API
- Updated the documentation for due to the above change.
- Modified the **sqlite** shell to make use of the new sqlite_get_table() API in order to print a list of tables in multiple columns, similar to the way "ls" prints filenames.
- Modified the **sqlite** shell to print a semicolon at the end of each CREATE statement in the output of the ".schema" command.

2000-09-21 (Not Released)

- Change the tclsqlite "eval" method to return a list of results if no callback script is specified.
- Change tclsqlite.c to use the Tcl_Obj interface

- Add tclsqlite.c to the libsqlite.a library

2000-09-14 (1.0.5)

- Changed the print format for floating point values from "%g" to "%.15g".
- Changed the comparison function so that numbers in exponential notation (ex: 1.234e+05) sort in numerical order.

2000-08-28 (1.0.4)

- Added functions **length()** and **substr()**.
- Fix a bug in the **sqlite** shell program that was causing a coredump when the output mode was "column" and the first row of data contained a NULL.

2000-08-22 (1.0.3)

- In the sqlite shell, print the "Database opened READ ONLY" message to stderr instead of stdout.
- In the sqlite shell, now print the version number on initial startup.
- Add the **sqlite_version[]** string constant to the library
- Makefile updates
- Bug fix: incorrect VDBE code was being generated for the following circumstance: a query on an indexed table containing a WHERE clause with an IN operator that had a subquery on its right-hand side.

2000-08-18 (1.0.1)

- Fix a bug in the configure script.
- Minor revisions to the website.

2000-08-17 (1.0)

- Change the **sqlite** program so that it can read databases for which it lacks write permission. (It used to refuse all access if it could not write.)

2000-08-09

- Treat carriage returns as white space.

2000-08-08

- Added pattern matching to the ".table" command in the "sqlite" command shell.

2000-08-04

- Documentation updates
- Added "busy" and "timeout" methods to the Tcl interface

2000-08-03

- File format version number was being stored in sqlite_master.tcl multiple times. This was harmless, but unnecessary. It is now fixed.

2000-08-02

- The file format for indices was changed slightly in order to work around an inefficiency that can sometimes come up with GDBM when there are large indices having many entries with the same key. **Incompatible Change**

2000-08-01

- The parser's stack was overflowing on a very long UPDATE statement. This is now fixed.

2000-07-31

- Finish the VDBE tutorial.
- Added documentation on compiling to WinNT.
- Fix a configuration program for WinNT.
- Fix a configuration problem for HPUX.

2000-07-29

- Better labels on column names of the result.

2000-07-28

- Added the **sqlite_busy_handler()** and **sqlite_busy_timeout()** interface.

2000-06-23

- Begin writing the VDBE tutorial.

2000-06-21

- Clean up comments and variable names. Changes to documentation. No functional changes to the code.

2000-06-19

- Column names in UPDATE statements were case sensitive. This mistake has now been fixed.

2000-06-18

- Added the concatenate string operator (||)

2000-06-12

- Added the fcnt() function to the SQL interpreter. The fcnt() function returns the number of database "Fetch" operations that have occurred. This function is designed for use in test scripts to verify that queries are efficient and appropriately optimized. Fcnt() has no other useful purpose, as far as I know.
- Added a bunch more tests that take advantage of the new fcnt() function. The new tests did not uncover any new problems.

2000-06-08

- Added lots of new test cases
- Fix a few bugs discovered while adding test cases
- Begin adding lots of new documentation

2000-06-06

- Added compound select operators: **UNION**, **UNION ALL**, **INTERSECT**, and **EXCEPT**
- Added support for using **(SELECT ...)** within expressions
- Added support for **IN** and **BETWEEN** operators
- Added support for **GROUP BY** and **HAVING**
- NULL values are now reported to the callback as a NULL pointer rather than an empty string.

2000-06-03

- Added support for default values on columns of a table.

- Improved test coverage. Fixed a few obscure bugs found by the improved tests.

2000-06-02

- All database files to be modified by an UPDATE, INSERT or DELETE are now locked before any changes are made to any files. This makes it safe (I think) to access the same database simultaneously from multiple processes.
- The code appears stable so we are now calling it "beta".

2000-06-01

- Better support for file locking so that two or more processes (or threads) can access the same database simultaneously. More work needed in this area, though.

2000-05-31

- Added support for aggregate functions (Ex: **COUNT(*)**, **MIN(...)**) to the SELECT statement.
- Added support for **SELECT DISTINCT ...**

2000-05-30

- Added the **LIKE** operator.
- Added a **GLOB** operator: similar to **LIKE** but it uses Unix shell globbing wildcards instead of the '%' and '_' wildcards of SQL.
- Added the **COPY** command patterned after PostgreSQL so that SQLite can now read the output of the **pg_dump** database dump utility of PostgreSQL.
- Added a **VACUUM** command that calls the **gdbm_reorganize()** function on the underlying database files.
- And many, many bug fixes...

2000-05-29

- Initial Public Release of Alpha code

File Format Changes in SQLite

The underlying file format for SQLite databases does not change in incompatible ways. There are literally tens of billions of SQLite database files in circulation and the SQLite developers are committing to supporting those files for decades into the future.

This document describes incompatibilities that have occurred in SQLite prior to 2004. Since 2004, there have been enhancements to SQLite such that newer database files are unreadable by older versions of the SQLite library. But the most recent versions of the SQLite library should be able to read and write any older SQLite database file without any problems.

In other words, since 2004 all SQLite releases have been backwards compatible, though not necessarily forwards compatible.

The following table summarizes the SQLite file format changes that have occurred since version 1.0.0:

Version Change	Approx. Date	Description Of File Format Change
1.0.32 to 2.0.0	2001-Sep-20	Version 1.0.X of SQLite used the GDBM library as its backend interface to the disk. Beginning in version 2.0.0, GDBM was replaced by a custom B-Tree library written especially for SQLite. The new B-Tree backend is twice as fast as GDBM, supports atomic commits and rollback, and stores an entire database in a single disk file instead using a separate file for each table as GDBM does. The two file formats are not even remotely similar.
2.0.8 to 2.1.0	2001-Nov-12	The same basic B-Tree format is used but the details of the index keys were changed in order to provide better query optimization opportunities. Some of the headers were also changed in order to increase the maximum size of a row from 64KB to 24MB.This change is an exception to the version number rule described above in that it is neither forwards or backwards compatible. A complete reload of the database is required. This is the only exception.
2.1.7 to 2.2.0	2001-Dec-21	Beginning with version 2.2.0, SQLite no longer builds an index for an INTEGER PRIMARY KEY column. Instead, it uses that column as the actual B-Tree key for the main table.Version 2.2.0 and later of the library will automatically detect when it is reading a 2.1.x database and will disable the new INTEGER PRIMARY KEY feature. In other words, version 2.2.x is backwards compatible to version 2.1.x. But version 2.1.x is not forward compatible with version 2.2.x. If you try to open a 2.2.x database with an older 2.1.x library

		and that database contains an INTEGER PRIMARY KEY, you will likely get a coredump. If the database schema does not contain any INTEGER PRIMARY KEYs, then the version 2.1.x and version 2.2.x database files will be identical and completely interchangeable.
2.2.5 to 2.3.0	2002-Jan-30	Beginning with version 2.3.0, SQLite supports some additional syntax (the "ON CONFLICT" clause) in the CREATE TABLE and CREATE INDEX statements that are stored in the SQLITE_MASTER table. If you create a database that contains this new syntax, then try to read that database using version 2.2.5 or earlier, the parser will not understand the new syntax and you will get an error. Otherwise, databases for 2.2.x and 2.3.x are interchangeable.
2.3.3 to 2.4.0	2002-Mar-10	Beginning with version 2.4.0, SQLite added support for views. Information about views is stored in the SQLITE_MASTER table. If an older version of SQLite attempts to read a database that contains VIEW information in the SQLITE_MASTER table, the parser will not understand the new syntax and initialization will fail. Also, the way SQLite keeps track of unused disk blocks in the database file changed slightly. If an older version of SQLite attempts to write a database that was previously written by version 2.4.0 or later, then it may leak disk blocks.
2.4.12 to 2.5.0	2002-Jun-17	Beginning with version 2.5.0, SQLite added support for triggers. Information about triggers is stored in the SQLITE_MASTER table. If an older version of SQLite attempts to read a database that contains a CREATE TRIGGER in the SQLITE_MASTER table, the parser will not understand the new syntax and initialization will fail.
2.5.6 to 2.6.0	2002-July-17	A design flaw in the layout of indices required a file format change to correct. This change appeared in version 2.6.0.If you use version 2.6.0 or later of the library to open a database file that was originally created by version 2.5.6 or earlier, an attempt to rebuild the database into the new format will occur automatically. This can take some time for a large database. (Allow 1 or 2 seconds per megabyte of database under Unix - longer under Windows.) This format conversion is irreversible. It is **strongly** suggested that you make a backup copy of older database files prior to opening them with version 2.6.0 or later of the library, in case there are errors in the format conversion logic.Version 2.6.0 or later of the library cannot open read-only database files from version 2.5.6 or earlier, since read-only files cannot be upgraded to the new format.
		Beginning with version 2.7.0, SQLite understands two different datatypes: text and numeric. Text data sorts in memcmp() order. Numeric data sorts in numerical order if it looks like a number, or in memcmp() order if it does

2.6.3 to 2.7.0	2002-Aug-13	not.When SQLite version 2.7.0 or later opens a 2.6.3 or earlier database, it assumes all columns of all tables have type "numeric". For 2.7.0 and later databases, columns have type "text" if their datatype string contains the substrings "char" or "clob" or "blob" or "text". Otherwise they are of type "numeric".Because "text" columns have a different sort order from numeric, indices on "text" columns occur in a different order for version 2.7.0 and later database. Hence version 2.6.3 and earlier of SQLite will be unable to read a 2.7.0 or later database. But version 2.7.0 and later of SQLite will read earlier databases.
2.7.6 to 2.8.0	2003-Feb-14	Version 2.8.0 introduces a change to the format of the rollback journal file. The main database file format is unchanged. Versions 2.7.6 and earlier can read and write 2.8.0 databases and vice versa. Version 2.8.0 can rollback a transaction that was started by version 2.7.6 and earlier. But version 2.7.6 and earlier cannot rollback a transaction started by version 2.8.0 or later.The only time this would ever be an issue is when you have a program using version 2.8.0 or later that crashes with an incomplete transaction, then you try to examine the database using version 2.7.6 or earlier. The 2.7.6 code will not be able to read the journal file and thus will not be able to rollback the incomplete transaction to restore the database.
2.8.14 to 3.0.0	2004-Jun-18	Version 3.0.0 is a major upgrade for SQLite that incorporates support for UTF-16, BLOBs, and a more compact encoding that results in database files that are typically 25% to 50% smaller. The new file format is very different and is completely incompatible with the version 2 file format.

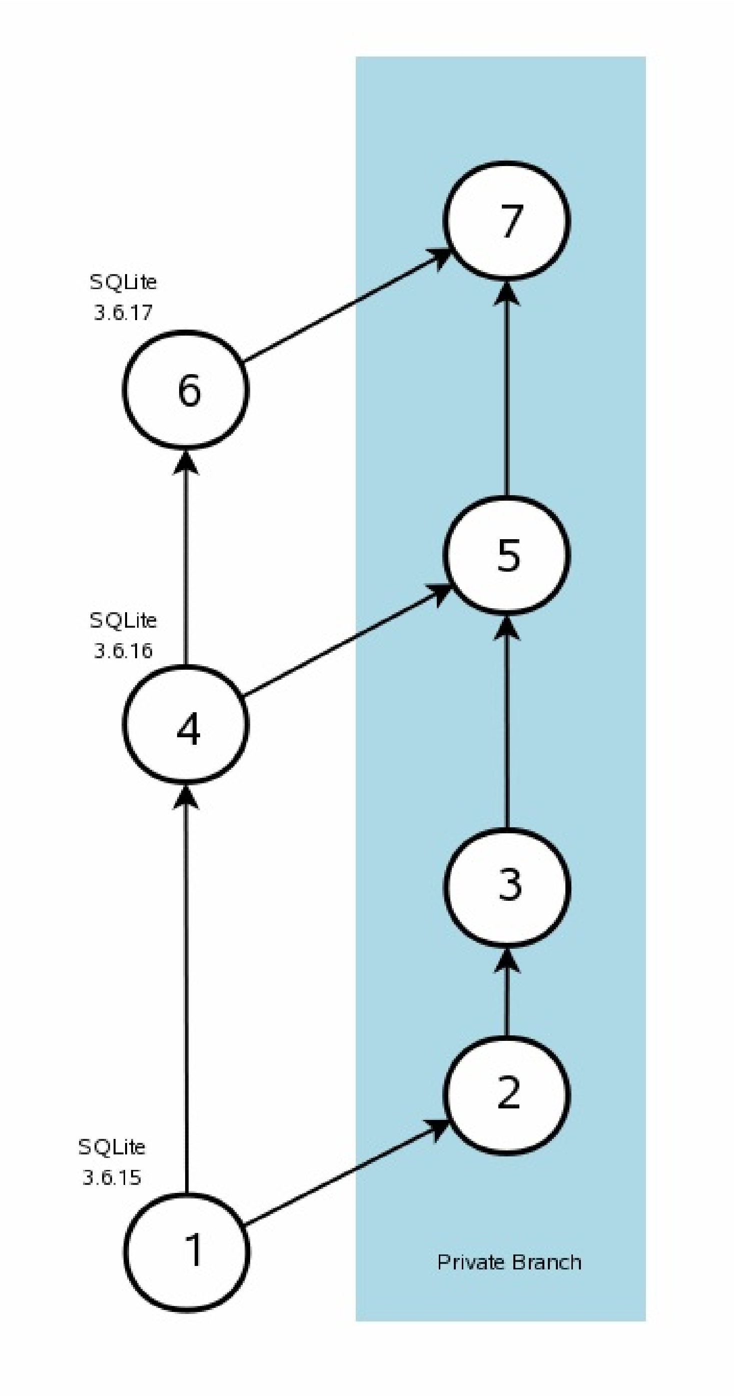
Maintaining Private Branches Of SQLite

1.0 Introduction

SQLite is designed to meet most developer's needs without any changes or customization. When changes are needed, they can normally be accomplished using start-time (1) or runtime (2) (3) (4) configuration methods or via compile-time options. It is very rare that an application developer will need to edit the SQLite source code in order to incorporate SQLite into a product.

We call custom modifications to the SQLite source code that are held for the use of a single application a "private branch". When a private branch becomes necessary, the application developer must take on the task of keeping the private branch in synchronization with the public SQLite sources. This is tedious. It can also be tricky, since while the SQLite file format and published interfaces are very stable, the internal implementation of SQLite changes quite rapidly. Hundreds or thousands of lines of code might change for any given SQLite point release.

This article outlines one possible method for keeping a private branch of SQLite in sync with the public SQLite source code. There are many ways of maintaining a private branch, of course. Nobody is compelled to use the method describe here. This article is not trying to impose a particular procedure on maintainers of private branches. The point of this article is to offer an example of one process for maintaining a private branch which can be used as a template for designing processes best suited for the circumstances of each individual project.



2.0 The Basic Idea

We propose to use the fossil software configuration management system to set up two branches. One branch (the "public branch" or "trunk") contains the published SQLite sources and the other branch is the private branch which contains the code that is customized for the project. Whenever a new public release of SQLite is made, that release is added to the public branch and then the changes are merged into the private branch.

This document proposes to use fossil, but any other distributed software configuration management system such as monotone or mercurial (a.k.a. "hg"), or git could serve just as well. The concept will be the same, though the specifics of the procedure will vary.

The diagram at the right illustrates the concept. One begins with a standard SQLite release. For the sake of example, suppose that one intends to create a private branch off of SQLite version 3.6.15. In the diagram this is version (1). The maintainer makes an exact copy of the baseline SQLite into the branch space, shown as version (2). Note that (1) and (2) are exactly the same. Then the maintainer applies the private changes to version (2) resulting in version (3). In other words, version (3) is SQLite version 3.6.15 plus edits.

Later, SQLite version 3.6.16 is released, as shown by circle (4) in the diagram. At the point, the private branch maintainer does a merge which takes all of the changes going from (1) to (4) and applies those changes to (3). The result is version (5), which is SQLite 3.6.16 plus edits.

There might be merge conflicts. In other words, it might be that the changes from (2) to (3) are incompatible with the changes from (1) to (4). In that case, the maintainer will have to manually resolve the conflicts. Hopefully conflicts will not come up that often. Conflicts are less likely to occur when the private edits are kept to a minimum.

The cycle above can be repeated many times. The diagram shows a third SQLite release, 3.6.17 in circle (6). The private branch maintainer can do another merge in order to incorporate the changes moving from (4) to (6) into the private branch, resulting in version (7).

3.0 The Procedure

The remainder of this document will guide the reader through the steps needed to maintain a private branch. The general idea is the same as outlined above. This section merely provides more detail.

We emphasize again that these steps are not intended to be the only acceptable method for maintaining private branch. This approach is one of many. Use this document as a baseline for preparing project-specific procedures. Do not be afraid to experiment.

3.1 Obtain The Software

Fossil is a computer program that must be installed on your machine before you use it. Fortunately, installing fossil is very easy. Fossil is a single "*.exe" file that you simply download and run. To uninstall fossil, simply delete the exe file. Detailed instructions for installing and getting started with fossil are available on the fossil website.

3.2 Create A Project Repository

Create a fossil repository to host the private branch using the following command:

```
fossil new private-project.fossil
```

You can call your project anything you like. The "`.fossil`" suffix is optional. For this document, we will continue to call the project "`private-project.fossil`". Note that `private-project.fossil` is an ordinary disk file (actually an SQLite database) that will contain your complete project history. You can make a backup of the project simply by making a copy of that one file.

If you want to configure the new project, type:

```
fossil ui private-project.fossil
```

The "ui" command will cause fossil to run a miniature built-in webserver and to launch your web-browser pointing at that webserver. You can use your web-browser to configure your project in various ways. See the instructions on the fossil website for additional information.

Once the project repository is created, create an open checkout of the project by moving to the directory where you want to keep all of the project source code and typing:

```
fossil open private-project.fossil
```

You can have multiple checkouts of the same project if you want. And you can "clone" the repository to different machines so that multiple developers can use it. See the fossil website for further information.

3.3 Installing The SQLite Baseline In Fossil

The repository created in the previous step is initially empty. The next step is to load the baseline SQLite release - circle (1) in the diagram above.

Begin by obtaining a copy of SQLite in whatever form you use it. The public SQLite you obtain should be as close to your private edited copy as possible. If your project uses the SQLite amalgamation, then get a copy of the amalgamation. If you use the preprocessed separate source files, get those instead. Put all the source files in the checkout directory created in the previous step.

The source code in public SQLite releases uses unix line endings (ASCII code 10: "newline" only, NL) and spaces instead of tabs. If you will be changing the line ending to windows-style line endings (ASCII codes 13, 10: "carriage-return" and "newline"; CR-NL) or if you will be changing space indents into tab indents, **make that change now** before you check in the baseline. The merging process will only work well if the differences between the public and the private branches are minimal. If every single line of the source file is changed in the private branch because you changed from NL to CR-NL line endings, then the merge steps will not work correctly.

Let us assume that you are using the amalgamation source code. Add the baseline to your project as follows:

```
fossil add sqlite3.c sqlite3.h
```

If you are using separate source files, name all of the source files instead of just the two amalgamation source files. Once this is done, commit your changes as follows:

```
fossil commit
```

You will be prompted for a check-in comment. Say whatever you like. After the commit completes, your baseline will be part of the repository. The following command, if you like, to see this on the "timeline":

```
fossil ui
```

That last command is the same "ui" command that we ran before. It starts a mini-webserver running and points your web browser at it. But this time we didn't have to specify the repository file because we are located inside a checkout and so fossil can figure out the repository for itself. If you want to type in the repository filename as the second argument, you can. But it is optional.

If you do not want to use your web browser to view the new check-in, you can get some information from the command-line using commands like these:

```
fossil timeline
fossil info
fossil status
```

3.4 Creating The Private Branch

The previous step created circle (1) in the diagram above. This step will create circle (2). Run the following command:

```

fossil branch new private trunk -bgcolor "#add8e8"
```

This command will create a new branch named "private" (you can use a different name if you like) and assign it a background color of light blue ("#add8e8"). You can omit the background color if you want, though having a distinct background does make it easier to tell the branch from the "trunk" (the public branch) on timeline displays. You can change the background color of the private branch or of the public branch (the "trunk") using the web interface if you like.

The command above created the new branch. But your checkout is still on the trunk - a fact you can see by running the command:

```
fossil info
```

To change your check-out to the private branch, type:

```
fossil update private
```

You can run the "info" command again to verify that you are on the private branch. To go back to the public branch, type:

```
fossil update trunk
```

Normally, fossil will modify all the files in your checkout when switching between the private and the public branches. But at this point, the files are identical in both branches so not modifications need to be made.

3.5 Adding Customizations To The Code In The Private Branch

Now it is time to make the private, custom modifications to SQLite which are the whole point of this exercise. Switch to the private branch (if you are not already there) using the "`fossil update private`" command, then bring up the source files in your text editor and make whatever changes you want to make. Once you have finished making changes, commit those changes using this command:

```
fossil commit
```

You will be prompted once again to enter a commit describing your changes. Then the commit will occur. The commit creates a new checkin in the repository that corresponds to circle (3) in the diagram above.

Now that the public and private branches are different, you can run the "`fossil update trunk`" and "`fossil update private`" commands and see that fossil really does change the files in the checkout as you switch back and forth between branches.

Note that in the diagram above, we showed the private edits as a single commit. This was for clarity of presentation only. There is nothing to stop you from doing dozens or hundreds of separate tiny changes and committing each separately. In fact, making many small changes is the preferred way to work. The only reason for doing all the changes in a single commit is that it makes the diagram easier to draw.

3.6 Incorporating New Public SQLite Releases

Suppose that after a while (about a month, usually) a new version of SQLite is released: 3.6.16. You will want to incorporate this new public version of SQLite into your repository in the public branch (the trunk). To do this, first change your repository over to the trunk:

```
fossil update trunk
```

Then download the new version of the SQLite sources and overwrite the files that are in the checkout.

If you made NL to CR-NL line ending changes or space to tab indentation changes in the original baseline, make the same changes to the new source file.

Once everything is ready, run the "`fossil commit`" command to check in the changes. This creates circle (4) in the diagram above.

3.7 Merging Public SQLite Updates Into The Private Branch

The next step is to move the changes in the public branch over into the private branch. In other words, we want to create circle (5) in the diagram above. Begin by changing to the private branch using "`fossil update private`". Then type this command:

```
fossil merge trunk
```

The "merge" command attempts to apply all the changes between circles (1) and (4) to the files in the local checkout. Note that circle (5) has not been created yet. You will need to run the "commit" to create circle (5).

It might be that there are conflicts in the merge. Conflicts occur when the same line of code was changed in different ways between circles (1) and (4) versus circles (2) and (3). The merge command will announce any conflicts and will include both versions of the conflicting lines in the output. You will need to bring up the files that contain conflicts and manually resolve the conflicts.

After resolving conflicts, many users like to compile and test the new version before committing it to the repository. Or you can commit first and test later. Either way, run the "`fossil commit`" command to check-in the circle (5) version.

3.8 Further Updates

As new versions of SQLite are released, repeat steps 3.6 and 3.7 to add changes in the new release to the private branch. Additional private changes can be made on the private branch in between releases if desired.

4.0 Variations

Since this document was first written, the canonical SQLite source code has been moved from the venerable CVS system into a Fossil repository at http://www.sqlite.org/src. This means that if you are working with canonical SQLite source code (as opposed to the amalgamation source code files, sqlite3.c and sqlite3.h) then you can create a private repository simply by cloning the official repository:

```
fossil clone http://www.sqlite.org/src private-project.fossil
```

This command both creates the new repository and populates it with all the latest SQLite could. You can the create a private branch as described in section 3.4.

When the private repository is created by cloning, incorporating new public SQLite releases becomes much easier too. To pull in all of the latest changes from the public SQLite repository, simply move into the open check-out and do:

```
fossil update
```

Then continue to merge the changes in "trunk" with your "private" changes as described in section 3.7.

Obsolete Documents

An Asynchronous I/O Module For SQLite

NOTE: WAL mode with PRAGMA synchronous set to NORMAL avoids calls to fsync() during transaction commit and only invokes fsync() during a checkpoint operation. The use of WAL mode largely obviates the need for this asynchronous I/O module. Hence, this module is no longer supported. The source code continues to exist in the SQLite source tree, but it is not a part of any standard build and is no longer maintained. This documentation is retained for historical reference.

Normally, when SQLite writes to a database file, it waits until the write operation is finished before returning control to the calling application. Since writing to the file-system is usually very slow compared with CPU bound operations, this can be a performance bottleneck. The asynchronous I/O backend is an extension that causes SQLite to perform all write requests using a separate thread running in the background. Although this does not reduce the overall system resources (CPU, disk bandwidth etc.), it does allow SQLite to return control to the caller quickly even when writing to the database.

1.0 FUNCTIONALITY

With asynchronous I/O, write requests are handled by a separate thread running in the background. This means that the thread that initiates a database write does not have to wait for (sometimes slow) disk I/O to occur. The write seems to happen very quickly, though in reality it is happening at its usual slow pace in the background.

Asynchronous I/O appears to give better responsiveness, but at a price. You lose the Durable property. With the default I/O backend of SQLite, once a write completes, you know that the information you wrote is safely on disk. With the asynchronous I/O, this is not the case. If your program crashes or if a power loss occurs after the database write but before the asynchronous write thread has completed, then the database change might never make it to disk and the next user of the database might not see your change.

You lose Durability with asynchronous I/O, but you still retain the other parts of ACID: Atomic, Consistent, and Isolated. Many applications get along fine without the Durability.

1.1 How it Works

Asynchronous I/O works by creating an SQLite VFS object and registering it with sqlite3_vfs_register(). When files opened via this VFS are written to (using the vfs xWrite() method), the data is not written directly to disk, but is placed in the "write-queue" to be handled by the background thread.

When files opened with the asynchronous VFS are read from (using the vfs xRead() method), the data is read from the file on disk and the write-queue, so that from the point of view of the vfs reader the xWrite() appears to have already completed.

The asynchronous I/O VFS is registered (and unregistered) by calls to the API functions sqlite3async_initialize() and sqlite3async_shutdown(). See section "Compilation and Usage" below for details.

1.2 Limitations

In order to gain experience with the main ideas surrounding asynchronous IO, this implementation is deliberately kept simple. Additional capabilities may be added in the future.

For example, as currently implemented, if writes are happening at a steady stream that exceeds the I/O capability of the background writer thread, the queue of pending write operations will grow without bound. If this goes on for long enough, the host system could run out of memory. A more sophisticated module could to keep track of the quantity of pending writes and stop accepting new write requests when the queue of pending writes grows too large.

1.3 Locking and Concurrency

Multiple connections from within a single process that use this implementation of asynchronous IO may access a single database file concurrently. From the point of view of the user, if all connections are from within a single process, there is no difference between the concurrency offered by "normal" SQLite and SQLite using the asynchronous backend.

If file-locking is enabled (it is enabled by default), then connections from multiple processes may also read and write the database file. However concurrency is reduced as follows:

- When a connection using asynchronous IO begins a database transaction, the database is locked immediately. However the lock is not released until after all relevant operations in the write-queue have been flushed to disk. This means (for example) that the database may remain locked for some time after a "COMMIT" or "ROLLBACK" is issued.

- If an application using asynchronous IO executes transactions in quick succession, other database users may be effectively locked out of the database. This is because when a BEGIN is executed, a database lock is established immediately. But when the corresponding COMMIT or ROLLBACK occurs, the lock is not released until the relevant part of the write-queue has been flushed through. As a result, if a COMMIT is followed by a BEGIN before the write-queue is flushed through, the database is never unlocked,preventing other processes from accessing the database.

File-locking may be disabled at runtime using the sqlite3async_control() API (see below). This may improve performance when an NFS or other network file-system, as the synchronous round-trips to the server be required to establish file locks are avoided. However, if multiple connections attempt to access the same database file when file-locking is disabled, application crashes and database corruption is a likely outcome.

2.0 COMPILATION AND USAGE

The asynchronous IO extension consists of a single file of C code (sqlite3async.c), and a header file (sqlite3async.h), located in the `ext/async/` subfolder of the SQLite source tree, that defines the C API used by applications to activate and control the modules functionality.

To use the asynchronous IO extension, compile sqlite3async.c as part of the application that uses SQLite. Then use the APIs defined in sqlite3async.h to initialize and configure the module.

The asynchronous IO VFS API is described in detail in comments in sqlite3async.h. Using the API usually consists of the following steps:

1. Register the asynchronous IO VFS with SQLite by calling the sqlite3async_initialize() function.
2. Create a background thread to perform write operations and call sqlite3async_run().
3. Use the normal SQLite API to read and write to databases via the asynchronous IO VFS.

Refer to comments in the sqlite3async.h header file for details.

3.0 PORTING

Currently the asynchronous IO extension is compatible with win32 systems and systems that support the pthreads interface, including Mac OS X, Linux, and other varieties of Unix.

To port the asynchronous IO extension to another platform, the user must implement mutex and condition variable primitives for the new platform. Currently there is no externally available interface to allow this, but modifying the code within sqlite3async.c to include the new platforms concurrency primitives is relatively easy. Search within sqlite3async.c for the comment string "PORTING FUNCTIONS" for details. Then implement new versions of each of the following:

```
static void async_mutex_enter(int eMutex);
static void async_mutex_leave(int eMutex);
static void async_cond_wait(int eCond, int eMutex);
static void async_cond_signal(int eCond);
static void async_sched_yield(void);
```

The functionality required of each of the above functions is described in comments in sqlite3async.c.

The C language interface to SQLite Version 2

The SQLite library is designed to be very easy to use from a C or C++ program. This document gives an overview of the C/C++ programming interface.

1.0 The Core API

The interface to the SQLite library consists of three core functions, one opaque data structure, and some constants used as return values. The core interface is as follows:

```
typedef struct sqlite sqlite;
#define SQLITE_OK           0   /* Successful result */

sqlite *sqlite_open(const char *dbname, int mode, char **errmsg);

void sqlite_close(sqlite *db);

int sqlite_exec(
  sqlite *db,
  char *sql,
  int (*xCallback)(void*,int,char**,char**),
  void *pArg,
  char **errmsg
);
```

The above is all you really need to know in order to use SQLite in your C or C++ programs. There are other interface functions available (and described below) but we will begin by describing the core functions shown above.

1.1 Opening a database

Use the **sqlite_open** function to open an existing SQLite database or to create a new SQLite database. The first argument is the database name. The second argument is intended to signal whether the database is going to be used for reading and writing or just for reading. But in the current implementation, the second argument to **sqlite_open** is ignored. The third argument is a pointer to a string pointer. If the third argument is not NULL and an error occurs while trying to open the database, then an error message will be written to memory obtained from malloc() and *errmsg will be made to point to this error message. The calling function is responsible for freeing the memory when it has finished with it.

The name of an SQLite database is the name of a file that will contain the database. If the file does not exist, SQLite attempts to create and initialize it. If the file is read-only (due to permission bits or because it is located on read-only media like a CD-ROM) then SQLite opens the database for reading only. The entire SQL database is stored in a single file on the disk. But additional temporary files may be created during the execution of an SQL command in order to store the database rollback journal or temporary and intermediate results of a query.

The return value of the **sqlite_open** function is a pointer to an opaque **sqlite** structure. This pointer will be the first argument to all subsequent SQLite function calls that deal with the same database. NULL is returned if the open fails for any reason.

1.2 Closing the database

To close an SQLite database, call the **sqlite_close** function passing it the sqlite structure pointer that was obtained from a prior call to **sqlite_open**. If a transaction is active when the database is closed, the transaction is rolled back.

1.3 Executing SQL statements

The **sqlite_exec** function is used to process SQL statements and queries. This function requires 5 parameters as follows:

1. A pointer to the sqlite structure obtained from a prior call to **sqlite_open**.
2. A null-terminated string containing the text of one or more SQL statements and/or queries to be processed.
3. A pointer to a callback function which is invoked once for each row in the result of a query. This argument may be NULL, in which case no callbacks will ever be invoked.
4. A pointer that is forwarded to become the first argument to the callback function.
5. A pointer to an error string. Error messages are written to space obtained from malloc() and the error string is made to point to the malloced space. The calling function is responsible for freeing this space when it has finished with it. This argument may be NULL, in which case error messages are not reported back to the calling function.

The callback function is used to receive the results of a query. A prototype for the callback function is as follows:

```
int Callback(void *pArg, int argc, char **argv, char **columnNames){
  return 0;
}
```

The first argument to the callback is just a copy of the fourth argument to **sqlite_exec** This parameter can be used to pass arbitrary information through to the callback function from client code. The second argument is the number of columns in the query result. The third argument is an array of pointers to strings where each string is a single column of the result for that record. Note that the callback function reports a NULL value in the database as a NULL pointer, which is very different from an empty string. If the i-th parameter is an empty string, we will get:

> ```
> argv[i][0] == 0
> ```

But if the i-th parameter is NULL we will get:

> ```
> argv[i] == 0
> ```

The names of the columns are contained in first *argc* entries of the fourth argument. If the SHOW_DATATYPES pragma is on (it is off by default) then the second *argc* entries in the 4th argument are the datatypes for the corresponding columns.

If the EMPTY_RESULT_CALLBACKS pragma is set to ON and the result of a query is an empty set, then the callback is invoked once with the third parameter (argv) set to 0. In other words

> ```
> argv == 0
> ```

The second parameter (argc) and the fourth parameter (columnNames) are still valid and can be used to determine the number and names of the result columns if there had been a result. The default behavior is not to invoke the callback at all if the result set is empty.

The callback function should normally return 0. If the callback function returns non-zero, the query is immediately aborted and **sqlite_exec** will return SQLITE_ABORT.

1.4 Error Codes

The **sqlite_exec** function normally returns SQLITE_OK. But if something goes wrong it can return a different value to indicate the type of error. Here is a complete list of the return codes:

```
#define SQLITE_OK           0   /* Successful result */
#define SQLITE_ERROR        1   /* SQL error or missing database */
#define SQLITE_INTERNAL     2   /* An internal logic error in SQLite */
#define SQLITE_PERM         3   /* Access permission denied */
#define SQLITE_ABORT        4   /* Callback routine requested an abort */
#define SQLITE_BUSY         5   /* The database file is locked */
#define SQLITE_LOCKED       6   /* A table in the database is locked */
#define SQLITE_NOMEM        7   /* A malloc() failed */
#define SQLITE_READONLY     8   /* Attempt to write a readonly database */
#define SQLITE_INTERRUPT    9   /* Operation terminated by sqlite_interrupt() */
#define SQLITE_IOERR       10   /* Some kind of disk I/O error occurred */
#define SQLITE_CORRUPT     11   /* The database disk image is malformed */
#define SQLITE_NOTFOUND    12   /* (Internal Only) Table or record not found */
#define SQLITE_FULL        13   /* Insertion failed because database is full */
#define SQLITE_CANTOPEN    14   /* Unable to open the database file */
#define SQLITE_PROTOCOL    15   /* Database lock protocol error */
#define SQLITE_EMPTY       16   /* (Internal Only) Database table is empty */
#define SQLITE_SCHEMA      17   /* The database schema changed */
#define SQLITE_TOOBIG      18   /* Too much data for one row of a table */
#define SQLITE_CONSTRAINT  19   /* Abort due to constraint violation */
#define SQLITE_MISMATCH    20   /* Data type mismatch */
#define SQLITE_MISUSE      21   /* Library used incorrectly */
#define SQLITE_NOLFS       22   /* Uses OS features not supported on host */
#define SQLITE_AUTH        23   /* Authorization denied */
#define SQLITE_ROW         100  /* sqlite_step() has another row ready */
#define SQLITE_DONE        101  /* sqlite_step() has finished executing */
```

The meanings of these various return values are as follows:

> SQLITE_OK
>
> This value is returned if everything worked and there were no errors.
>
> SQLITE_INTERNAL
>
> This value indicates that an internal consistency check within the SQLite library failed. This can only happen if there is a bug in the SQLite library. If you ever get an SQLITE_INTERNAL reply from an **sqlite_exec** call, please report the problem on the SQLite mailing list.
>
> SQLITE_ERROR
>
> This return value indicates that there was an error in the SQL that was passed into the **sqlite_exec**.
>
> SQLITE_PERM
>
> This return value says that the access permissions on the database file are such that the file cannot be opened.
>
> SQLITE_ABORT
>
> This value is returned if the callback function returns non-zero.
>
> SQLITE_BUSY

> This return code indicates that another program or thread has the database locked. SQLite allows two or more threads to read the database at the same time, but only one thread can have the database open for writing at the same time. Locking in SQLite is on the entire database.
>
> SQLITE_LOCKED
>
> This return code is similar to SQLITE_BUSY in that it indicates that the database is locked. But the source of the lock is a recursive call to **sqlite_exec**. This return can only occur if you attempt to invoke sqlite_exec from within a callback routine of a query from a prior invocation of sqlite_exec. Recursive calls to sqlite_exec are allowed as long as they do not attempt to write the same table.
>
> SQLITE_NOMEM
>
> This value is returned if a call to **malloc** fails.
>
> SQLITE_READONLY
>
> This return code indicates that an attempt was made to write to a database file that is opened for reading only.
>
> SQLITE_INTERRUPT
>
> This value is returned if a call to **sqlite_interrupt** interrupts a database operation in progress.
>
> SQLITE_IOERR
>
> This value is returned if the operating system informs SQLite that it is unable to perform some disk I/O operation. This could mean that there is no more space left on the disk.
>
> SQLITE_CORRUPT
>
> This value is returned if SQLite detects that the database it is working on has become corrupted. Corruption might occur due to a rogue process writing to the database file or it might happen due to a previously undetected logic error in of SQLite. This value is also returned if a disk I/O error occurs in such a way that SQLite is forced to leave the database file in a corrupted state. The latter should only happen due to a hardware or operating system malfunction.
>
> SQLITE_FULL
>
> This value is returned if an insertion failed because there is no space left on the disk, or the database is too big to hold any more information. The latter case should only occur for databases that are larger than 2GB in size.

> SQLITE_CANTOPEN
>
> This value is returned if the database file could not be opened for some reason.
>
> SQLITE_PROTOCOL
>
> This value is returned if some other process is messing with file locks and has violated the file locking protocol that SQLite uses on its rollback journal files.
>
> SQLITE_SCHEMA
>
> When the database first opened, SQLite reads the database schema into memory and uses that schema to parse new SQL statements. If another process changes the schema, the command currently being processed will abort because the virtual machine code generated assumed the old schema. This is the return code for such cases. Retrying the command usually will clear the problem.
>
> SQLITE_TOOBIG
>
> SQLite will not store more than about 1 megabyte of data in a single row of a single table. If you attempt to store more than 1 megabyte in a single row, this is the return code you get.
>
> SQLITE_CONSTRAINT
>
> This constant is returned if the SQL statement would have violated a database constraint.
>
> SQLITE_MISMATCH
>
> This error occurs when there is an attempt to insert non-integer data into a column labeled INTEGER PRIMARY KEY. For most columns, SQLite ignores the data type and allows any kind of data to be stored. But an INTEGER PRIMARY KEY column is only allowed to store integer data.
>
> SQLITE_MISUSE
>
> This error might occur if one or more of the SQLite API routines is used incorrectly. Examples of incorrect usage include calling **sqlite_exec** after the database has been closed using **sqlite_close** or calling **sqlite_exec** with the same database pointer simultaneously from two separate threads.
>
> SQLITE_NOLFS
>
> This error means that you have attempts to create or access a file database file that is larger that 2GB on a legacy Unix machine that lacks large file support.
>
> SQLITE_AUTH

> This error indicates that the authorizer callback has disallowed the SQL you are attempting to execute.
>
> SQLITE_ROW
>
> This is one of the return codes from the **sqlite_step** routine which is part of the non-callback API. It indicates that another row of result data is available.
>
> SQLITE_DONE
>
> This is one of the return codes from the **sqlite_step** routine which is part of the non-callback API. It indicates that the SQL statement has been completely executed and the **sqlite_finalize** routine is ready to be called.

2.0 Accessing Data Without Using A Callback Function

The **sqlite_exec** routine described above used to be the only way to retrieve data from an SQLite database. But many programmers found it inconvenient to use a callback function to obtain results. So beginning with SQLite version 2.7.7, a second access interface is available that does not use callbacks.

The new interface uses three separate functions to replace the single **sqlite_exec** function.

```
typedef struct sqlite_vm sqlite_vm;

int sqlite_compile(
  sqlite *db,              /* The open database */
  const char *zSql,        /* SQL statement to be compiled */
  const char **pzTail,     /* OUT: uncompiled tail of zSql */
  sqlite_vm **ppVm,        /* OUT: the virtual machine to execute zSql */
  char **pzErrmsg          /* OUT: Error message. */
);

int sqlite_step(
  sqlite_vm *pVm,          /* The virtual machine to execute */
  int *pN,                 /* OUT: Number of columns in result */
  const char ***pazValue,  /* OUT: Column data */
  const char ***pazColName /* OUT: Column names and datatypes */
);

int sqlite_finalize(
  sqlite_vm *pVm,          /* The virtual machine to be finalized */
  char **pzErrMsg          /* OUT: Error message */
);
```

The strategy is to compile a single SQL statement using **sqlite_compile** then invoke **sqlite_step** multiple times, once for each row of output, and finally call **sqlite_finalize** to clean up after the SQL has finished execution.

2.1 Compiling An SQL Statement Into A Virtual Machine

The **sqlite_compile** "compiles" a single SQL statement (specified by the second parameter) and generates a virtual machine that is able to execute that statement. As with must interface routines, the first parameter must be a pointer to an sqlite structure that was obtained from a prior call to **sqlite_open**.

A pointer to the virtual machine is stored in a pointer which is passed in as the 4th parameter. Space to hold the virtual machine is dynamically allocated. To avoid a memory leak, the calling function must invoke **sqlite_finalize** on the virtual machine after it has finished with it. The 4th parameter may be set to NULL if an error is encountered during compilation.

If any errors are encountered during compilation, an error message is written into memory obtained from **malloc** and the 5th parameter is made to point to that memory. If the 5th parameter is NULL, then no error message is generated. If the 5th parameter is not NULL, then the calling function should dispose of the memory containing the error message by calling **sqlite_freemem**.

If the 2nd parameter actually contains two or more statements of SQL, only the first statement is compiled. (This is different from the behavior of **sqlite_exec** which executes all SQL statements in its input string.) The 3rd parameter to **sqlite_compile** is made to point to the first character beyond the end of the first statement of SQL in the input. If the 2nd parameter contains only a single SQL statement, then the 3rd parameter will be made to point to the '\000' terminator at the end of the 2nd parameter.

On success, **sqlite_compile** returns SQLITE_OK. Otherwise and error code is returned.

2.2 Step-By-Step Execution Of An SQL Statement

After a virtual machine has been generated using **sqlite_compile** it is executed by one or more calls to **sqlite_step**. Each invocation of **sqlite_step**, except the last one, returns a single row of the result. The number of columns in the result is stored in the integer that the 2nd parameter points to. The pointer specified by the 3rd parameter is made to point to an array of pointers to column values. The pointer in the 4th parameter is made to point to an array of pointers to column names and datatypes. The 2nd through 4th parameters to **sqlite_step** convey the same information as the 2nd through 4th parameters of the **callback** routine when using the **sqlite_exec** interface. Except, with **sqlite_step** the column datatype information is always included in the in the 4th parameter regardless of whether or not the SHOW_DATATYPES pragma is on or off.

Each invocation of **sqlite_step** returns an integer code that indicates what happened during that step. This code may be SQLITE_BUSY, SQLITE_ROW, SQLITE_DONE, SQLITE_ERROR, or SQLITE_MISUSE.

If the virtual machine is unable to open the database file because it is locked by another thread or process, **sqlite_step** will return SQLITE_BUSY. The calling function should do some other activity, or sleep, for a short amount of time to give the lock a chance to clear, then invoke **sqlite_step** again. This can be repeated as many times as desired.

Whenever another row of result data is available, **sqlite_step** will return SQLITE_ROW. The row data is stored in an array of pointers to strings and the 2nd parameter is made to point to this array.

When all processing is complete, **sqlite_step** will return either SQLITE_DONE or SQLITE_ERROR. SQLITE_DONE indicates that the statement completed successfully and SQLITE_ERROR indicates that there was a run-time error. (The details of the error are obtained from **sqlite_finalize**.) It is a misuse of the library to attempt to call **sqlite_step** again after it has returned SQLITE_DONE or SQLITE_ERROR.

When **sqlite_step** returns SQLITE_DONE or SQLITE_ERROR, the *pN and* pazColName values are set to the number of columns in the result set and to the names of the columns, just as they are for an SQLITE_ROW return. This allows the calling code to find the number of result columns and the column names and datatypes even if the result set is empty. The *pazValue parameter is always set to NULL when the return codes is SQLITE_DONE or SQLITE_ERROR. If the SQL being executed is a statement that does not return a result (such as an INSERT or an UPDATE) then* pN will be set to zero and *pazColName will be set to NULL.

If you abuse the library by trying to call **sqlite_step** inappropriately it will attempt return SQLITE_MISUSE. This can happen if you call sqlite_step() on the same virtual machine at the same time from two or more threads or if you call sqlite_step() again after it returned SQLITE_DONE or SQLITE_ERROR or if you pass in an invalid virtual machine pointer to sqlite_step(). You should not depend on the SQLITE_MISUSE return code to indicate an error. It is possible that a misuse of the interface will go undetected and result in a program crash. The SQLITE_MISUSE is intended as a debugging aid only - to help you detect incorrect usage prior to a mishap. The misuse detection logic is not guaranteed to work in every case.

2.3 Deleting A Virtual Machine

Every virtual machine that **sqlite_compile** creates should eventually be handed to **sqlite_finalize**. The sqlite_finalize() procedure deallocates the memory and other resources that the virtual machine uses. Failure to call sqlite_finalize() will result in resource leaks in your program.

The **sqlite_finalize** routine also returns the result code that indicates success or failure of the SQL operation that the virtual machine carried out. The value returned by sqlite_finalize() will be the same as would have been returned had the same SQL been executed by **sqlite_exec**. The error message returned will also be the same.

It is acceptable to call **sqlite_finalize** on a virtual machine before **sqlite_step** has returned SQLITE_DONE. Doing so has the effect of interrupting the operation in progress. Partially completed changes will be rolled back and the database will be restored to its original state (unless an alternative recovery algorithm is selected using an ON CONFLICT clause in the SQL being executed.) The effect is the same as if a callback function of **sqlite_exec** had returned non-zero.

It is also acceptable to call **sqlite_finalize** on a virtual machine that has never been passed to **sqlite_step** even once.

3.0 The Extended API

Only the three core routines described in section 1.0 are required to use SQLite. But there are many other functions that provide useful interfaces. These extended routines are as follows:

```
int sqlite_last_insert_rowid(sqlite*);

int sqlite_changes(sqlite*);

int sqlite_get_table(
  sqlite*,
  char *sql,
  char ***result,
  int *nrow,
  int *ncolumn,
  char **errmsg
);

void sqlite_free_table(char**);

void sqlite_interrupt(sqlite*);

int sqlite_complete(const char *sql);

void sqlite_busy_handler(sqlite*, int (*)(void*,const char*,int), void*);

void sqlite_busy_timeout(sqlite*, int ms);

const char sqlite_version[];

const char sqlite_encoding[];

int sqlite_exec_printf(
  sqlite*,
  char *sql,
  int (*)(void*,int,char**,char**),
  void*,
  char **errmsg,
  ...
);
```

```
int sqlite_exec_vprintf(
  sqlite*,
  char *sql,
  int (*)(void*,int,char**,char**),
  void*,
  char **errmsg,
  va_list
);

int sqlite_get_table_printf(
  sqlite*,
  char *sql,
  char ***result,
  int *nrow,
  int *ncolumn,
  char **errmsg,
  ...
);

int sqlite_get_table_vprintf(
  sqlite*,
  char *sql,
  char ***result,
  int *nrow,
  int *ncolumn,
  char **errmsg,
  va_list
);

char *sqlite_mprintf(const char *zFormat, ...);

char *sqlite_vmprintf(const char *zFormat, va_list);

void sqlite_freemem(char*);

void sqlite_progress_handler(sqlite*, int, int (*)(void*), void*);
```

All of the above definitions are included in the "sqlite.h" header file that comes in the source tree.

3.1 The ROWID of the most recent insert

Every row of an SQLite table has a unique integer key. If the table has a column labeled INTEGER PRIMARY KEY, then that column serves as the key. If there is no INTEGER PRIMARY KEY column then the key is a unique integer. The key for a row can be accessed in a SELECT statement or used in a WHERE or ORDER BY clause using any of the names "ROWID", "OID", or "*ROWID*".

When you do an insert into a table that does not have an INTEGER PRIMARY KEY column, or if the table does have an INTEGER PRIMARY KEY but the value for that column is not specified in the VALUES clause of the insert, then the key is automatically generated. You can find the value of the key for the most recent INSERT statement using the **sqlite_last_insert_rowid** API function.

3.2 The number of rows that changed

The **sqlite_changes** API function returns the number of rows that have been inserted, deleted, or modified since the database was last quiescent. A "quiescent" database is one in which there are no outstanding calls to **sqlite_exec** and no VMs created by **sqlite_compile** that have not been finalized by **sqlite_finalize**. In common usage, **sqlite_changes** returns the number of rows inserted, deleted, or modified by the most recent **sqlite_exec** call or since the most recent **sqlite_compile**. But if you have nested calls to **sqlite_exec** (that is, if the callback routine of one **sqlite_exec** invokes another **sqlite_exec**) or if you invoke **sqlite_compile** to create a new VM while there is still another VM in existence, then the meaning of the number returned by **sqlite_changes** is more complex. The number reported includes any changes that were later undone by a ROLLBACK or ABORT. But rows that are deleted because of a DROP TABLE are *not* counted.

SQLite implements the command "**DELETE FROM table**" (without a WHERE clause) by dropping the table then recreating it. This is much faster than deleting the elements of the table individually. But it also means that the value returned from **sqlite_changes** will be zero regardless of the number of elements that were originally in the table. If an accurate count of the number of elements deleted is necessary, use "**DELETE FROM table WHERE 1**" instead.

3.3 Querying into memory obtained from malloc()

The **sqlite_get_table** function is a wrapper around **sqlite_exec** that collects all the information from successive callbacks and writes it into memory obtained from malloc(). This is a convenience function that allows the application to get the entire result of a database query with a single function call.

The main result from **sqlite_get_table** is an array of pointers to strings. There is one element in this array for each column of each row in the result. NULL results are represented by a NULL pointer. In addition to the regular data, there is an added row at the beginning of the array that contains the name of each column of the result.

As an example, consider the following query:

> SELECT employee_name, login, host FROM users WHERE login LIKE 'd%';

This query will return the name, login and host computer name for every employee whose login begins with the letter "d". If this query is submitted to **sqlite_get_table** the result might look like this:

> nrow = 2 ncolumn = 3 result[0] = "employee_name" result[1] = "login" result[2] = "host" result[3] = "dummy" result[4] = "No such user" result[5] = 0 result[6] = "D. Richard Hipp" result[7] = "drh" result[8] = "zadok"

Notice that the "host" value for the "dummy" record is NULL so the result[] array contains a NULL pointer at that slot.

If the result set of a query is empty, then by default **sqlite_get_table** will set nrow to 0 and leave its result parameter is set to NULL. But if the EMPTY_RESULT_CALLBACKS pragma is ON then the result parameter is initialized to the names of the columns only. For example, consider this query which has an empty result set:

> SELECT employee_name, login, host FROM users WHERE employee_name IS NULL;

The default behavior gives this results:

> nrow = 0 ncolumn = 0 result = 0

But if the EMPTY_RESULT_CALLBACKS pragma is ON, then the following is returned:

> nrow = 0 ncolumn = 3 result[0] = "employee_name" result[1] = "login" result[2] = "host"

Memory to hold the information returned by **sqlite_get_table** is obtained from malloc(). But the calling function should not try to free this information directly. Instead, pass the complete table to **sqlite_free_table** when the table is no longer needed. It is safe to call **sqlite_free_table** with a NULL pointer such as would be returned if the result set is empty.

The **sqlite_get_table** routine returns the same integer result code as **sqlite_exec**.

3.4 Interrupting an SQLite operation

The **sqlite_interrupt** function can be called from a different thread or from a signal handler to cause the current database operation to exit at its first opportunity. When this happens, the **sqlite_exec** routine (or the equivalent) that started the database operation will return SQLITE_INTERRUPT.

3.5 Testing for a complete SQL statement

The next interface routine to SQLite is a convenience function used to test whether or not a string forms a complete SQL statement. If the **sqlite_complete** function returns true when its input is a string, then the argument forms a complete SQL statement. There are no guarantees that the syntax of that statement is correct, but we at least know the statement is complete. If **sqlite_complete** returns false, then more text is required to complete the SQL statement.

For the purpose of the **sqlite_complete** function, an SQL statement is complete if it ends in a semicolon.

The **sqlite** command-line utility uses the **sqlite_complete** function to know when it needs to call **sqlite_exec**. After each line of input is received, **sqlite** calls **sqlite_complete** on all input in its buffer. If **sqlite_complete** returns true, then **sqlite_exec** is called and the input buffer is reset. If **sqlite_complete** returns false, then the prompt is changed to the continuation prompt and another line of text is read and added to the input buffer.

3.6 Library version string

The SQLite library exports the string constant named **sqlite_version** which contains the version number of the library. The header file contains a macro SQLITE_VERSION with the same information. If desired, a program can compare the SQLITE_VERSION macro against the **sqlite_version** string constant to verify that the version number of the header file and the library match.

3.7 Library character encoding

By default, SQLite assumes that all data uses a fixed-size 8-bit character (iso8859). But if you give the --enable-utf8 option to the configure script, then the library assumes UTF-8 variable sized characters. This makes a difference for the LIKE and GLOB operators and the LENGTH() and SUBSTR() functions. The static string **sqlite_encoding** will be set to either "UTF-8" or "iso8859" to indicate how the library was compiled. In addition, the **sqlite.h** header file will define one of the macros **SQLITE_UTF8** or **SQLITE_ISO8859**, as appropriate.

Note that the character encoding mechanism used by SQLite cannot be changed at run-time. This is a compile-time option only. The **sqlite_encoding** character string just tells you how the library was compiled.

3.8 Changing the library's response to locked files

The **sqlite_busy_handler** procedure can be used to register a busy callback with an open SQLite database. The busy callback will be invoked whenever SQLite tries to access a database that is locked. The callback will typically do some other useful work, or perhaps sleep, in order to give the lock a chance to clear. If the callback returns non-zero, then SQLite tries again to access the database and the cycle repeats. If the callback returns zero, then SQLite aborts the current operation and returns SQLITE_BUSY.

The arguments to **sqlite_busy_handler** are the opaque structure returned from **sqlite_open**, a pointer to the busy callback function, and a generic pointer that will be passed as the first argument to the busy callback. When SQLite invokes the busy callback, it sends it three arguments: the generic pointer that was passed in as the third argument to

sqlite_busy_handler, the name of the database table or index that the library is trying to access, and the number of times that the library has attempted to access the database table or index.

For the common case where we want the busy callback to sleep, the SQLite library provides a convenience routine **sqlite_busy_timeout**. The first argument to **sqlite_busy_timeout** is a pointer to an open SQLite database and the second argument is a number of milliseconds. After **sqlite_busy_timeout** has been executed, the SQLite library will wait for the lock to clear for at least the number of milliseconds specified before it returns SQLITE_BUSY. Specifying zero milliseconds for the timeout restores the default behavior.

3.9 Using the `_printf()` wrapper functions

The four utility functions

- **sqlite_exec_printf()**
- **sqlite_exec_vprintf()**
- **sqlite_get_table_printf()**
- **sqlite_get_table_vprintf()**

implement the same query functionality as **sqlite_exec** and **sqlite_get_table**. But instead of taking a complete SQL statement as their second argument, the four **_printf** routines take a printf-style format string. The SQL statement to be executed is generated from this format string and from whatever additional arguments are attached to the end of the function call.

There are two advantages to using the SQLite printf functions instead of **sprintf**. First of all, with the SQLite printf routines, there is never a danger of overflowing a static buffer as there is with **sprintf**. The SQLite printf routines automatically allocate (and later frees) as much memory as is necessary to hold the SQL statements generated.

The second advantage the SQLite printf routines have over **sprintf** are two new formatting options specifically designed to support string literals in SQL. Within the format string, the %q formatting option works very much like %s in that it reads a null-terminated string from the argument list and inserts it into the result. But %q translates the inserted string by making two copies of every single-quote (') character in the substituted string. This has the effect of escaping the end-of-string meaning of single-quote within a string literal. The %Q formatting option works similar; it translates the single-quotes like %q and additionally encloses the resulting string in single-quotes. If the argument for the %Q formatting options is a NULL pointer, the resulting string is NULL without single quotes.

Consider an example. Suppose you are trying to insert a string value into a database table where the string value was obtained from user input. Suppose the string to be inserted is stored in a variable named zString. The code to do the insertion might look like this:

```
sqlite_exec_printf(db,
  "INSERT INTO table1 VALUES('%s')",
  0, 0, 0, zString);
```

If the zString variable holds text like "Hello", then this statement will work just fine. But suppose the user enters a string like "Hi y'all!". The SQL statement generated reads as follows:

```
INSERT INTO table1 VALUES('Hi y'all')
```

This is not valid SQL because of the apostrophe in the word "y'all". But if the %q formatting option is used instead of %s, like this:

```
sqlite_exec_printf(db,
  "INSERT INTO table1 VALUES('%q')",
  0, 0, 0, zString);
```

Then the generated SQL will look like the following:

```
INSERT INTO table1 VALUES('Hi y''all')
```

Here the apostrophe has been escaped and the SQL statement is well-formed. When generating SQL on-the-fly from data that might contain a single-quote character ('), it is always a good idea to use the SQLite printf routines and the %q formatting option instead of **sprintf**.

If the %Q formatting option is used instead of %q, like this:

```
sqlite_exec_printf(db,
  "INSERT INTO table1 VALUES(%Q)",
  0, 0, 0, zString);
```

Then the generated SQL will look like the following:

```
INSERT INTO table1 VALUES('Hi y''all')
```

If the value of the zString variable is NULL, the generated SQL will look like the following:

```
INSERT INTO table1 VALUES(NULL)
```

All of the _printf() routines above are built around the following two functions:

```
char *sqlite_mprintf(const char *zFormat, ...);
char *sqlite_vmprintf(const char *zFormat, va_list);
```

The **sqlite_mprintf()** routine works like the standard library **sprintf()** except that it writes its results into memory obtained from malloc() and returns a pointer to the malloced buffer. **sqlite_mprintf()** also understands the %q and %Q extensions described above. The **sqlite_vmprintf()** is a varargs version of the same routine. The string pointer that these routines return should be freed by passing it to **sqlite_freemem()**.

3.10 Performing background jobs during large queries

The **sqlite_progress_handler()** routine can be used to register a callback routine with an SQLite database to be invoked periodically during long running calls to **sqlite_exec()**, **sqlite_step()** and the various wrapper functions.

The callback is invoked every N virtual machine operations, where N is supplied as the second argument to **sqlite_progress_handler()**. The third and fourth arguments to **sqlite_progress_handler()** are a pointer to the routine to be invoked and a void pointer to be passed as the first argument to it.

The time taken to execute each virtual machine operation can vary based on many factors. A typical value for a 1 GHz PC is between half and three million per second but may be much higher or lower, depending on the query. As such it is difficult to schedule background operations based on virtual machine operations. Instead, it is recommended that a callback be scheduled relatively frequently (say every 1000 instructions) and external timer routines used to determine whether or not background jobs need to be run.

4.0 Adding New SQL Functions

Beginning with version 2.4.0, SQLite allows the SQL language to be extended with new functions implemented as C code. The following interface is used:

```
typedef struct sqlite_func sqlite_func;

int sqlite_create_function(
  sqlite *db,
  const char *zName,
  int nArg,
  void (*xFunc)(sqlite_func*,int,const char**),
  void *pUserData
);
int sqlite_create_aggregate(
  sqlite *db,
  const char *zName,
  int nArg,
  void (*xStep)(sqlite_func*,int,const char**),
  void (*xFinalize)(sqlite_func*),
  void *pUserData
);

char *sqlite_set_result_string(sqlite_func*,const char*,int);
void sqlite_set_result_int(sqlite_func*,int);
void sqlite_set_result_double(sqlite_func*,double);
void sqlite_set_result_error(sqlite_func*,const char*,int);

void *sqlite_user_data(sqlite_func*);
void *sqlite_aggregate_context(sqlite_func*, int nBytes);
int sqlite_aggregate_count(sqlite_func*);
```

The **sqlite_create_function()** interface is used to create regular functions and **sqlite_create_aggregate()** is used to create new aggregate functions. In both cases, the **db** parameter is an open SQLite database on which the functions should be registered, **zName** is the name of the new function, **nArg** is the number of arguments, and **pUserData** is a pointer which is passed through unchanged to the C implementation of the function. Both routines return 0 on success and non-zero if there are any errors.

The length of a function name may not exceed 255 characters. Any attempt to create a function whose name exceeds 255 characters in length will result in an error.

For regular functions, the **xFunc** callback is invoked once for each function call. The implementation of xFunc should call one of the **sqlite*set_result*...** interfaces to return its result. The **sqlite_user_data()** routine can be used to retrieve the **pUserData** pointer that was passed in when the function was registered.

For aggregate functions, the **xStep** callback is invoked once for each row in the result and then **xFinalize** is invoked at the end to compute a final answer. The xStep routine can use the **sqlite_aggregate_context()** interface to allocate memory that will be unique to that particular instance of the SQL function. This memory will be automatically deleted after xFinalize is called. The **sqlite_aggregate_count()** routine can be used to find out how many rows of data were passed to the aggregate. The xFinalize callback should invoke one of the **sqlite*set_result*...** interfaces to set the final result of the aggregate.

SQLite now implements all of its built-in functions using this interface. For additional information and examples on how to create new SQL functions, review the SQLite source code in the file **func.c**.

5.0 Multi-Threading And SQLite

If SQLite is compiled with the THREADSAFE preprocessor macro set to 1, then it is safe to use SQLite from two or more threads of the same process at the same time. But each thread should have its own **sqlite*** pointer returned from **sqlite_open**. It is never safe for two or more threads to access the same **sqlite*** pointer at the same time.

In precompiled SQLite libraries available on the website, the Unix versions are compiled with THREADSAFE turned off but the Windows versions are compiled with THREADSAFE turned on. If you need something different that this you will have to recompile.

Under Unix, an **sqlite*** pointer should not be carried across a **fork()** system call into the child process. The child process should open its own copy of the database after the **fork()**.

6.0 Usage Examples

For examples of how the SQLite C/C++ interface can be used, refer to the source code for the **sqlite** program in the file **src/shell.c** of the source tree. Additional information about sqlite is available at cli.html. See also the sources to the Tcl interface for SQLite in the source file **src/tclsqlite.c**.

Datatypes In SQLite Version 2

1.0 Typelessness

SQLite is "typeless". This means that you can store any kind of data you want in any column of any table, regardless of the declared datatype of that column. (See the one exception to this rule in section 2.0 below.) This behavior is a feature, not a bug. A database is supposed to store and retrieve data and it should not matter to the database what format that data is in. The strong typing system found in most other SQL engines and codified in the SQL language spec is a misfeature - it is an example of the implementation showing through into the interface. SQLite seeks to overcome this misfeature by allowing you to store any kind of data into any kind of column and by allowing flexibility in the specification of datatypes.

A datatype to SQLite is any sequence of zero or more names optionally followed by a parenthesized lists of one or two signed integers. Notice in particular that a datatype may be *zero* or more names. That means that an empty string is a valid datatype as far as SQLite is concerned. So you can declare tables where the datatype of each column is left unspecified, like this:

```
CREATE TABLE ex1(a,b,c);
```

Even though SQLite allows the datatype to be omitted, it is still a good idea to include it in your CREATE TABLE statements, since the data type often serves as a good hint to other programmers about what you intend to put in the column. And if you ever port your code to another database engine, that other engine will probably require a datatype of some kind. SQLite accepts all the usual datatypes. For example:

```
CREATE TABLE ex2(
  a VARCHAR(10),
  b NVARCHAR(15),
  c TEXT,
  d INTEGER,
  e FLOAT,
  f BOOLEAN,
  g CLOB,
  h BLOB,
  i TIMESTAMP,
  j NUMERIC(10,5)
  k VARYING CHARACTER (24),
  l NATIONAL VARYING CHARACTER(16)
);
```

And so forth. Basically any sequence of names optionally followed by one or two signed integers in parentheses will do.

2.0 The INTEGER PRIMARY KEY

One exception to the typelessness of SQLite is a column whose type is INTEGER PRIMARY KEY. (And you must use "INTEGER" not "INT". A column of type INT PRIMARY KEY is typeless just like any other.) INTEGER PRIMARY KEY columns must contain a 32-bit signed integer. Any attempt to insert non-integer data will result in an error.

INTEGER PRIMARY KEY columns can be used to implement the equivalent of AUTOINCREMENT. If you try to insert a NULL into an INTEGER PRIMARY KEY column, the column will actually be filled with an integer that is one greater than the largest key already in the table. Or if the largest key is 2147483647, then the column will be filled with a random integer. Either way, the INTEGER PRIMARY KEY column will be assigned a unique integer. You can retrieve this integer using the **sqlite_last_insert_rowid()** API function or using the **last_insert_rowid()** SQL function in a subsequent SELECT statement.

3.0 Comparison and Sort Order

SQLite is typeless for the purpose of deciding what data is allowed to be stored in a column. But some notion of type comes into play when sorting and comparing data. For these purposes, a column or an expression can be one of two types: **numeric** and **text**. The sort or comparison may give different results depending on which type of data is being sorted or compared.

If data is of type **text** then the comparison is determined by the standard C data comparison functions **memcmp()** or **strcmp()**. The comparison looks at bytes from two inputs one by one and returns the first non-zero difference. Strings are '\000' terminated so shorter strings sort before longer strings, as you would expect.

For numeric data, this situation is more complex. If both inputs look like well-formed numbers, then they are converted into floating point values using **atof()** and compared numerically. If one input is not a well-formed number but the other is, then the number is considered to be less than the non-number. If neither inputs is a well-formed number, then **strcmp()** is used to do the comparison.

Do not be confused by the fact that a column might have a "numeric" datatype. This does not mean that the column can contain only numbers. It merely means that if the column does contain a number, that number will sort in numerical order.

For both text and numeric values, NULL sorts before any other value. A comparison of any value against NULL using operators like "<" or ">=" is always false.

4.0 How SQLite Determines Datatypes

For SQLite version 2.6.3 and earlier, all values used the numeric datatype. The text datatype appears in version 2.7.0 and later. In the sequel it is assumed that you are using version 2.7.0 or later of SQLite.

For an expression, the datatype of the result is often determined by the outermost operator. For example, arithmetic operators ("+", "", "%") always return a numeric results. The string concatenation operator ("||") returns a text result. And so forth. If you are ever in doubt about the datatype of an expression you can use the special *typeof() SQL function to determine what the datatype is. For example:

```
sqlite> SELECT typeof('abc'+123);
numeric
sqlite> SELECT typeof('abc'||123);
text
```

For table columns, the datatype is determined by the type declaration of the CREATE TABLE statement. The datatype is text if and only if the type declaration contains one or more of the following strings:

> BLOB CHAR CLOB TEXT

The search for these strings in the type declaration is case insensitive, of course. If any of the above strings occur anywhere in the type declaration, then the datatype of the column is text. Notice that the type "VARCHAR" contains "CHAR" as a substring so it is considered text.

If none of the strings above occur anywhere in the type declaration, then the datatype is numeric. Note in particular that the datatype for columns with an empty type declaration is numeric.

5.0 Examples

Consider the following two command sequences:

```
CREATE TABLE t1(a INTEGER UNIQUE);        CREATE TABLE t2(b TEXT UNIQUE);
INSERT INTO t1 VALUES('0');               INSERT INTO t2 VALUES(0);
INSERT INTO t1 VALUES('0.0');             INSERT INTO t2 VALUES(0.0);
```

In the sequence on the left, the second insert will fail. In this case, the strings '0' and '0.0' are treated as numbers since they are being inserted into a numeric column but 0==0.0 which violates the uniqueness constraint. However, the second insert in the right-hand sequence works. In this case, the constants 0 and 0.0 are treated a strings which means that they are distinct.

SQLite always converts numbers into double-precision (64-bit) floats for comparison purposes. This means that a long sequence of digits that differ only in insignificant digits will compare equal if they are in a numeric column but will compare unequal if they are in a text column. We have:

```
INSERT INTO t1                            INSERT INTO t2
   VALUES('12345678901234567890');           VALUES(12345678901234567890);
INSERT INTO t1                            INSERT INTO t2
   VALUES('12345678901234567891');           VALUES(12345678901234567891);
```

As before, the second insert on the left will fail because the comparison will convert both strings into floating-point number first and the only difference in the strings is in the 20-th digit which exceeds the resolution of a 64-bit float. In contrast, the second insert on the right will work because in that case, the numbers being inserted are strings and are compared using memcmp().

Numeric and text types make a difference for the DISTINCT keyword too:

```
CREATE TABLE t3(a INTEGER);               CREATE TABLE t4(b TEXT);
INSERT INTO t3 VALUES('0');               INSERT INTO t4 VALUES(0);
INSERT INTO t3 VALUES('0.0');             INSERT INTO t4 VALUES(0.0);
SELECT DISTINCT * FROM t3;                SELECT DISTINCT * FROM t4;
```

The SELECT statement on the left returns a single row since '0' and '0.0' are treated as numbers and are therefore indistinct. But the SELECT statement on the right returns two rows since 0 and 0.0 are treated a strings which are different.

The Virtual Database Engine of SQLite

> **Obsolete Documentation Warning:** This document describes the virtual machine used in SQLite version 2.8.0. The virtual machine in SQLite version 3.0 and 3.1 is similar in concept but is now register-based instead of stack-based, has five operands per opcode instead of three, and has a different set of opcodes from those shown below. See the virtual machine instructions document for the current set of VDBE opcodes and a brief overview of how the VDBE operates. This document is retained as an historical reference.

If you want to know how the SQLite library works internally, you need to begin with a solid understanding of the Virtual Database Engine or VDBE. The VDBE occurs right in the middle of the processing stream (see the architecture diagram) and so it seems to touch most parts of the library. Even parts of the code that do not directly interact with the VDBE are usually in a supporting role. The VDBE really is the heart of SQLite.

This article is a brief introduction to how the VDBE works and in particular how the various VDBE instructions (documented here) work together to do useful things with the database. The style is tutorial, beginning with simple tasks and working toward solving more complex problems. Along the way we will visit most submodules in the SQLite library. After completing this tutorial, you should have a pretty good understanding of how SQLite works and will be ready to begin studying the actual source code.

Preliminaries

The VDBE implements a virtual computer that runs a program in its virtual machine language. The goal of each program is to interrogate or change the database. Toward this end, the machine language that the VDBE implements is specifically designed to search, read, and modify databases.

Each instruction of the VDBE language contains an opcode and three operands labeled P1, P2, and P3. Operand P1 is an arbitrary integer. P2 is a non-negative integer. P3 is a pointer to a data structure or null-terminated string, possibly null. Only a few VDBE instructions use all three operands. Many instructions use only one or two operands. A significant number of instructions use no operands at all but instead take their data and store their results on the execution stack. The details of what each instruction does and which operands it uses are described in the separate opcode description document.

A VDBE program begins execution on instruction 0 and continues with successive instructions until it either (1) encounters a fatal error, (2) executes a Halt instruction, or (3) advances the program counter past the last instruction of the program. When the VDBE completes execution, all open database cursors are closed, all memory is freed, and everything is popped from the stack. So there are never any worries about memory leaks or undeallocated resources.

If you have done any assembly language programming or have worked with any kind of abstract machine before, all of these details should be familiar to you. So let's jump right in and start looking as some code.

Inserting Records Into The Database

We begin with a problem that can be solved using a VDBE program that is only a few instructions long. Suppose we have an SQL table that was created like this:

```
CREATE TABLE examp(one text, two int);
```

In words, we have a database table named "examp" that has two columns of data named "one" and "two". Now suppose we want to insert a single record into this table. Like this:

```
INSERT INTO examp VALUES('Hello, World!',99);
```

We can see the VDBE program that SQLite uses to implement this INSERT using the **sqlite** command-line utility. First start up **sqlite** on a new, empty database, then create the table. Next change the output format of **sqlite** to a form that is designed to work with VDBE program dumps by entering the ".explain" command. Finally, enter the [INSERT] statement shown above, but precede the [INSERT] with the special keyword [EXPLAIN]. The [EXPLAIN] keyword will cause **sqlite** to print the VDBE program rather than execute it. We have:

> `$ **sqlite test_database_1** sqlite> **CREATE TABLE examp(one text, two int);** sqlite> **.explain** sqlite> **EXPLAIN INSERT INTO examp VALUES('Hello, World!',99);** addr opcode p1 p2 p3
>
> ---
>
> 0 Transaction 0 0
> 1 VerifyCookie 0 81
> 2 Transaction 1 0
> 3 Integer 0 0
> 4 OpenWrite 0 3 examp
> 5 NewRecno 0 0
> 6 String 0 0 Hello, World!
> 7 Integer 99 0 99
> 8 MakeRecord 2 0
> 9 PutIntKey 0 1
> 10 Close 0 0
> 11 Commit 0 0
> 12 Halt 0 0`

As you can see above, our simple insert statement is implemented in 12 instructions. The first 3 and last 2 instructions are a standard prologue and epilogue, so the real work is done in the middle 7 instructions. There are no jumps, so the program executes once through from top to bottom. Let's now look at each instruction in detail.

> `0 Transaction 0 0 1 VerifyCookie 0 81 2 Transaction 1 0`

The instruction Transaction begins a transaction. The transaction ends when a Commit or Rollback opcode is encountered. P1 is the index of the database file on which the transaction is started. Index 0 is the main database file. A write lock is obtained on the database file when a transaction is started. No other process can read or write the file while the transaction is underway. Starting a transaction also creates a rollback journal. A transaction must be started before any changes can be made to the database.

The instruction VerifyCookie checks cookie 0 (the database schema version) to make sure it is equal to P2 (the value obtained when the database schema was last read). P1 is the database number (0 for the main database). This is done to make sure the database schema hasn't been changed by another thread, in which case it has to be reread.

The second Transaction instruction begins a transaction and starts a rollback journal for database 1, the database used for temporary tables.

> `3 Integer 0 0 4 OpenWrite 0 3 examp`

The instruction Integer pushes the integer value P1 (0) onto the stack. Here 0 is the number of the database to use in the following OpenWrite instruction. If P3 is not NULL then it is a string representation of the same integer. Afterwards the stack looks like this:

> | (integer) 0 |

The instruction OpenWrite opens a new read/write cursor with handle P1 (0 in this case) on table "examp", whose root page is P2 (3, in this database file). Cursor handles can be any non-negative integer. But the VDBE allocates cursors in an array with the size of the array being one more than the largest cursor. So to conserve memory, it is best to use handles beginning with zero and working upward consecutively. Here P3 ("examp") is the name of the table being opened, but this is unused, and only generated to make the code easier to read. This instruction pops the database number to use (0, the main database) from the top of the stack, so afterwards the stack is empty again.

> ```
> 5 NewRecno 0 0
> ```

The instruction NewRecno creates a new integer record number for the table pointed to by cursor P1. The record number is one not currently used as a key in the table. The new record number is pushed onto the stack. Afterwards the stack looks like this:

> | (integer) new record key |
>
> ```
> 6 String 0 0 Hello, World!
> ```

The instruction String pushes its P3 operand onto the stack. Afterwards the stack looks like this:

> | (string) "Hello, World!" | | (integer) new record key |
>
> ```
> 7 Integer 99 0 99
> ```

The instruction Integer pushes its P1 operand (99) onto the stack. Afterwards the stack looks like this:

> | (integer) 99 | | (string) "Hello, World!" | | (integer) new record key |
>
> ```
> 8 MakeRecord 2 0
> ```

The instruction MakeRecord pops the top P1 elements off the stack (2 in this case) and converts them into the binary format used for storing records in a database file. (See the file format description for details.) The new record generated by the MakeRecord instruction is pushed back onto the stack. Afterwards the stack looks like this:

> | (record) "Hello, World!", 99 | | (integer) new record key |
>
> ```
> 9 PutIntKey 0 1
> ```

The instruction PutIntKey uses the top 2 stack entries to write an entry into the table pointed to by cursor P1. A new entry is created if it doesn't already exist or the data for an existing entry is overwritten. The record data is the top stack entry, and the key is the next entry down. The stack is popped twice by this instruction. Because operand P2 is 1 the row change count is incremented and the rowid is stored for subsequent return by the sqlite_last_insert_rowid() function. If P2 is 0 the row change count is unmodified. This instruction is where the insert actually occurs.

```
10 Close 0 0
```

The instruction Close closes a cursor previously opened as P1 (0, the only open cursor). If P1 is not currently open, this instruction is a no-op.

```
11 Commit 0 0
```

The instruction Commit causes all modifications to the database that have been made since the last Transaction to actually take effect. No additional modifications are allowed until another transaction is started. The Commit instruction deletes the journal file and releases the write lock on the database. A read lock continues to be held if there are still cursors open.

```
12 Halt 0 0
```

The instruction Halt causes the VDBE engine to exit immediately. All open cursors, Lists, Sorts, etc are closed automatically. P1 is the result code returned by sqlite_exec(). For a normal halt, this should be SQLITE_OK (0). For errors, it can be some other value. The operand P2 is only used when there is an error. There is an implied "Halt 0 0 0" instruction at the end of every program, which the VDBE appends when it prepares a program to run.

Tracing VDBE Program Execution

If the SQLite library is compiled without the NDEBUG preprocessor macro, then the PRAGMA vdbe_trace causes the VDBE to trace the execution of programs. Though this feature was originally intended for testing and debugging, it can also be useful in learning about how the VDBE operates. Use " `PRAGMA vdbe_trace=ON;` " to turn tracing on and " `PRAGMA vdbe_trace=OFF` " to turn tracing back off. Like this:

```
sqlite> **PRAGMA vdbe_trace=ON;** 0 Halt 0 0 sqlite> **INSERT INTO examp VALUES('Hello,
```

With tracing mode on, the VDBE prints each instruction prior to executing it. After the instruction is executed, the top few entries in the stack are displayed. The stack display is omitted if the stack is empty.

On the stack display, most entries are shown with a prefix that tells the datatype of that stack entry. Integers begin with " `i:` ". Floating point values begin with " `r:` ". (The "r" stands for "real-number".) Strings begin with either " `s:` ", " `t:` ", " `e:` " or " `z:` ". The difference among the string prefixes is caused by how their memory is allocated. The z: strings are stored in memory obtained from **malloc()**. The t: strings are statically allocated. The e: strings are ephemeral. All other strings have the s: prefix. This doesn't make any difference to you, the observer, but it is vitally important to the VDBE since the z: strings need to be passed to **free()** when they are popped to avoid a memory leak. Note that only the first 10 characters of string values are displayed and that binary values (such as the result of the MakeRecord instruction) are treated as strings. The only other datatype that can be stored on the VDBE stack is a NULL, which is display without prefix as simply " `NULL` ". If an integer has been placed on the stack as both an integer and a string, its prefix is " `si:` ".

Simple Queries

At this point, you should understand the basics of how the VDBE writes to a database. Now let's look at how it does queries. We will use the following simple SELECT statement as our example:

```
SELECT * FROM examp;
```

The VDBE program generated for this SQL statement is as follows:

```
`sqlite> EXPLAIN SELECT * FROM examp; addr opcode p1 p2 p3
----------------------------------------------------------
0 ColumnName 0 0 one
1 ColumnName 1 0 two
2 Integer 0 0
3 OpenRead 0 3 examp
4 VerifyCookie 0 81
5 Rewind 0 10
6 Column 0 0
7 Column 0 1
8 Callback 2 0
9 Next 0 6
10 Close 0 0
11 Halt 0 0`
```

Before we begin looking at this problem, let's briefly review how queries work in SQLite so that we will know what we are trying to accomplish. For each row in the result of a query, SQLite will invoke a callback function with the following prototype:

```
int Callback(void *pUserData, int nColumn, char *azData[], char *azColumnName[]);
```

The SQLite library supplies the VDBE with a pointer to the callback function and the **pUserData** pointer. (Both the callback and the user data were originally passed in as arguments to the **sqlite_exec()** API function.) The job of the VDBE is to come up with values for **nColumn**, **azData[]**, and **azColumnName[]**. **nColumn** is the number of columns in the results, of course. **azColumnName[]** is an array of strings where each string is the name of one of the result columns. **azData[]** is an array of strings holding the actual data.

```
0 ColumnName 0 0 one 1 ColumnName 1 0 two
```

The first two instructions in the VDBE program for our query are concerned with setting up values for **azColumn**. The ColumnName instructions tell the VDBE what values to fill in for each element of the **azColumnName[]** array. Every query will begin with one ColumnName instruction for each column in the result, and there will be a matching Column instruction for each one later in the query.

```
2 Integer 0 0 3 OpenRead 0 3 examp 4 VerifyCookie 0 81
```

Instructions 2 and 3 open a read cursor on the database table that is to be queried. This works the same as the OpenWrite instruction in the INSERT example except that the cursor is opened for reading this time instead of for writing. Instruction 4 verifies the database schema as in the INSERT example.

```
5 Rewind 0 10
```

The Rewind instruction initializes a loop that iterates over the "examp" table. It rewinds the cursor P1 to the first entry in its table. This is required by the Column and Next instructions, which use the cursor to iterate through the table. If the table is empty, then jump to P2 (10), which is the instruction just past the loop. If the table is not empty, fall through to the following instruction at 6, which is the beginning of the loop body.

```
6 Column 0 0 7 Column 0 1 8 Callback 2 0
```

The instructions 6 through 8 form the body of the loop that will execute once for each record in the database file. The Column instructions at addresses 6 and 7 each take the P2-th column from the P1-th cursor and push it onto the stack. In this example, the first Column instruction is pushing the value for the column "one" onto the stack and the second Column instruction is pushing the value for column "two". The Callback instruction at address 8 invokes the callback() function. The P1 operand to Callback becomes the value for **nColumn**. The Callback instruction pops P1 values from the stack and uses them to fill the **azData[]** array.

```
9 Next 0 6
```

The instruction at address 9 implements the branching part of the loop. Together with the Rewind at address 5 it forms the loop logic. This is a key concept that you should pay close attention to. The Next instruction advances the cursor P1 to the next record. If the cursor advance was successful, then jump immediately to P2 (6, the beginning of the loop body). If the cursor was at the end, then fall through to the following instruction, which ends the loop.

```
10 Close 0 0 11 Halt 0 0
```

The Close instruction at the end of the program closes the cursor that points into the table "examp". It is not really necessary to call Close here since all cursors will be automatically closed by the VDBE when the program halts. But we needed an instruction for the Rewind to jump to so we might as well go ahead and have that instruction do something useful. The Halt instruction ends the VDBE program.

Note that the program for this SELECT query didn't contain the Transaction and Commit instructions used in the INSERT example. Because the SELECT is a read operation that doesn't alter the database, it doesn't require a transaction.

A Slightly More Complex Query

The key points of the previous example were the use of the Callback instruction to invoke the callback function, and the use of the Next instruction to implement a loop over all records of the database file. This example attempts to drive home those ideas by demonstrating a slightly more complex query that involves more columns of output, some of which are computed values, and a WHERE clause that limits which records actually make it to the callback function. Consider this query:

```
SELECT one, two, one || two AS 'both'
FROM examp
WHERE one LIKE 'H%'
```

This query is perhaps a bit contrived, but it does serve to illustrate our points. The result will have three column with names "one", "two", and "both". The first two columns are direct copies of the two columns in the table and the third result column is a string formed by concatenating the first and second columns of the table. Finally, the WHERE clause says that we will only chose rows for the results where the "one" column begins with an "H". Here is what the VDBE program looks like for this query:

> `addr opcode p1 p2 p3
>
> ---
>
> 0 ColumnName 0 0 one 1 ColumnName 1 0 two 2 ColumnName 2 0 both 3 Integer 0 0 4 OpenRead 0 3 examp 5 VerifyCookie 0 81 6 Rewind 0 18 7 String 0 0 H% 8 Column 0 0 9 Function 2 0 ptr(0x7f1ac0) 10 IfNot 1 17 11 Column 0 0 12 Column 0 1 13 Column 0 0 14 Column 0 1 15 Concat 2 0 16 Callback 3 0 17 Next 0 7 18 Close 0 0 19 Halt 0 0`

Except for the WHERE clause, the structure of the program for this example is very much like the prior example, just with an extra column. There are now 3 columns, instead of 2 as before, and there are three ColumnName instructions. A cursor is opened using the OpenRead instruction, just like in the prior example. The Rewind instruction at address 6 and the Next at address 17 form a loop over all records of the table. The Close instruction at the end is there to give the Rewind instruction something to jump to when it is done. All of this is just like in the first query demonstration.

The Callback instruction in this example has to generate data for three result columns instead of two, but is otherwise the same as in the first query. When the Callback instruction is invoked, the left-most column of the result should be the lowest in the stack and the right-most result column should be the top of the stack. We can see the stack being set up this way at addresses 11 through 15. The Column instructions at 11 and 12 push the values for the first two columns in the result. The two Column instructions at 13 and 14 pull in the values needed to compute the third result column and the Concat instruction at 15 joins them together into a single entry on the stack.

The only thing that is really new about the current example is the WHERE clause which is implemented by instructions at addresses 7 through 10. Instructions at address 7 and 8 push onto the stack the value of the "one" column from the table and the literal string "H%". The Function instruction at address 9 pops these two values from the stack and pushes the result of the LIKE() function back onto the stack. The IfNot instruction pops the top stack value and causes an immediate jump forward to the Next instruction if the top value was false (*not* not like the literal string "H%"). Taking this jump effectively skips the callback, which is the whole point of the WHERE clause. If the result of the comparison is true, the jump is not taken and control falls through to the Callback instruction below.

Notice how the LIKE operator is implemented. It is a user-defined function in SQLite, so the address of its function definition is specified in P3. The operand P1 is the number of function arguments for it to take from the stack. In this case the LIKE() function takes 2 arguments. The arguments are taken off the stack in reverse order (right-to-left), so the pattern to match is the top stack element, and the next element is the data to compare. The return value is pushed onto the stack.

A Template For SELECT Programs

The first two query examples illustrate a kind of template that every SELECT program will follow. Basically, we have:

1. Initialize the **azColumnName[]** array for the callback.
2. Open a cursor into the table to be queried.
3. For each record in the table, do:
    i. If the WHERE clause evaluates to FALSE, then skip the steps that follow and continue to the next record.
    ii. Compute all columns for the current row of the result.
    iii. Invoke the callback function for the current row of the result.
4. Close the cursor.

This template will be expanded considerably as we consider additional complications such as joins, compound selects, using indices to speed the search, sorting, and aggregate functions with and without GROUP BY and HAVING clauses. But the same basic ideas will continue to apply.

UPDATE And DELETE Statements

The UPDATE and DELETE statements are coded using a template that is very similar to the SELECT statement template. The main difference, of course, is that the end action is to modify the database rather than invoke a callback function. Because it modifies the database it will also use transactions. Let's begin by looking at a DELETE statement:

```
DELETE FROM examp WHERE two&lt;50; &lt;="" pre=""&gt;
```

This DELETE statement will remove every record from the "examp" table where the "two" column is less than 50. The code generated to do this is as follows:

> `addr opcode p1 p2 p3
>
> ---
>
> 0 Transaction 1 0 1 Transaction 0 0 2 VerifyCookie 0 178 3 Integer 0 0 4 OpenRead 0 3 examp 5 Rewind 0 12 6 Column 0 1 7 Integer 50 0 50 8 Ge 1 11 9 Recno 0 0 10 ListWrite 0 0 11 Next 0 6 12 Close 0 0 13 ListRewind 0 0 14 Integer 0 0 15 OpenWrite 0 3 16 ListRead 0 20 17 NotExists 0 19 18 Delete 0 1 19 Goto 0 16 20 ListReset 0 0 21 Close 0 0 22 Commit 0 0 23 Halt 0 0`

Here is what the program must do. First it has to locate all of the records in the table "examp" that are to be deleted. This is done using a loop very much like the loop used in the SELECT examples above. Once all records have been located, then we can go back through and delete them one by one. Note that we cannot delete each record as soon as we find it. We have to locate all records first, then go back and delete them. This is because the SQLite database backend might change the scan order after a delete operation. And if the scan order changes in the middle of the scan, some records might be visited more than once and other records might not be visited at all.

So the implementation of DELETE is really in two loops. The first loop (instructions 5 through 11) locates the records that are to be deleted and saves their keys onto a temporary list, and the second loop (instructions 16 through 19) uses the key list to delete the records one by one.

```
0 Transaction 1 0 1 Transaction 0 0 2 VerifyCookie 0 178 3 Integer 0 0 4 OpenRead 0 3 ex
```

Instructions 0 though 4 are as in the INSERT example. They start transactions for the main and temporary databases, verify the database schema for the main database, and open a read cursor on the table "examp". Notice that the cursor is opened for reading, not writing. At this stage of the program we are only going to be scanning the table, not changing it. We will reopen the same table for writing later, at instruction 15.

```
5 Rewind 0 12
```

As in the SELECT example, the Rewind instruction rewinds the cursor to the beginning of the table, readying it for use in the loop body.

```
6 Column 0 1 7 Integer 50 0 50 8 Ge 1 11
```

The WHERE clause is implemented by instructions 6 through 8. The job of the where clause is to skip the ListWrite if the WHERE condition is false. To this end, it jumps ahead to the Next instruction if the "two" column (extracted by the Column instruction) is greater than or equal to 50.

As before, the Column instruction uses cursor P1 and pushes the data record in column P2 (1, column "two") onto the stack. The Integer instruction pushes the value 50 onto the top of the stack. After these two instructions the stack looks like:

> | (integer) 50 | | (record) current record for column "two" |

The Ge operator compares the top two elements on the stack, pops them, and then branches based on the result of the comparison. If the second element is >= the top element, then jump to address P2 (the Next instruction at the end of the loop). Because P1 is true, if either operand is NULL (and thus the result is NULL) then take the jump. If we don't jump, just advance to the next instruction.

```
9 Recno 0 0 10 ListWrite 0 0
```

The Recno instruction pushes onto the stack an integer which is the first 4 bytes of the key to the current entry in a sequential scan of the table pointed to by cursor P1. The ListWrite instruction writes the integer on the top of the stack into a temporary storage list and pops the top element. This is the important work of this loop, to store the keys of the records to be deleted so we can delete them in the second loop. After this ListWrite instruction the stack is empty again.

```
11 Next 0 6 12 Close 0 0
```

The Next instruction increments the cursor to point to the next element in the table pointed to by cursor P0, and if it was successful branches to P2 (6, the beginning of the loop body). The Close instruction closes cursor P1. It doesn't affect the temporary storage list because it isn't associated with cursor P1; it is instead a global working list (which can be saved with ListPush).

```
13 ListRewind 0 0
```

The ListRewind instruction rewinds the temporary storage list to the beginning. This prepares it for use in the second loop.

```
14 Integer 0 0 15 OpenWrite 0 3
```

As in the INSERT example, we push the database number P1 (0, the main database) onto the stack and use OpenWrite to open the cursor P1 on table P2 (base page 3, "examp") for modification.

```
16 ListRead 0 20 17 NotExists 0 19 18 Delete 0 1 19 Goto 0 16
```

This loop does the actual deleting. It is organized differently from the one in the UPDATE example. The ListRead instruction plays the role that the Next did in the INSERT loop, but because it jumps to P2 on failure, and Next jumps on success, we put it at the start of the loop instead of the end. This means that we have to put a Goto at the end of the loop to jump back to the loop test at the beginning. So this loop has the form of a C while(){...} loop, while the loop in the INSERT example had the form of a do{...}while() loop. The Delete instruction fills the role that the callback function did in the preceding examples.

The ListRead instruction reads an element from the temporary storage list and pushes it onto the stack. If this was successful, it continues to the next instruction. If this fails because the list is empty, it branches to P2, which is the instruction just after the loop. Afterwards the stack looks like:

> | (integer) key for current record |

Notice the similarity between the ListRead and Next instructions. Both operations work according to this rule:

> Push the next "thing" onto the stack and fall through OR jump to P2, depending on whether or not there is a next "thing" to push.

One difference between Next and ListRead is their idea of a "thing". The "things" for the Next instruction are records in a database file. "Things" for ListRead are integer keys in a list. Another difference is whether to jump or fall through if there is no next "thing". In this case, Next falls through, and ListRead jumps. Later on, we will see other looping instructions (NextIdx and SortNext) that operate using the same principle.

The NotExists instruction pops the top stack element and uses it as an integer key. If a record with that key does not exist in table P1, then jump to P2. If a record does exist, then fall through to the next instruction. In this case P2 takes us to the Goto at the end of the loop, which jumps back to the ListRead at the beginning. This could have been coded to have P2 be 16, the ListRead at the start of the loop, but the SQLite parser which generated this code didn't make that optimization.

The Delete does the work of this loop; it pops an integer key off the stack (placed there by the preceding ListRead) and deletes the record of cursor P1 that has that key. Because P2 is true, the row change counter is incremented.

The Goto jumps back to the beginning of the loop. This is the end of the loop.

> `20 ListReset 0 0 21 Close 0 0 22 Commit 0 0 23 Halt 0 0`

This block of instruction cleans up the VDBE program. Three of these instructions aren't really required, but are generated by the SQLite parser from its code templates, which are designed to handle more complicated cases.

The ListReset instruction empties the temporary storage list. This list is emptied automatically when the VDBE program terminates, so it isn't necessary in this case. The Close instruction closes the cursor P1. Again, this is done by the VDBE engine when it is finished running this program. The Commit ends the current transaction successfully, and causes all changes that occurred in this transaction to be saved to the database. The final Halt is also unnecessary, since it is added to every VDBE program when it is prepared to run.

UPDATE statements work very much like DELETE statements except that instead of deleting the record they replace it with a new one. Consider this example:

```
UPDATE examp SET one= '(' || one || ')' WHERE two &lt; 50;
```

Instead of deleting records where the "two" column is less than 50, this statement just puts the "one" column in parentheses The VDBE program to implement this statement follows:

```
`addr opcode p1 p2 p3

0 Transaction 1 0
1 Transaction 0 0
2 VerifyCookie 0 178
3 Integer 0 0
4 OpenRead 0 3 examp
5 Rewind 0 12
6 Column 0 1
7 Integer 50 0 50
8 Ge 1 11
9 Recno 0 0
10 ListWrite 0 0
11 Next 0 6
12 Close 0 0
13 Integer 0 0
14 OpenWrite 0 3
15 ListRewind 0 0
16 ListRead 0 28
17 Dup 0 0
18 NotExists 0 16
19 String 0 0 (
20 Column 0 0
21 Concat 2 0
22 String 0 0 )
23 Concat 2 0
24 Column 0 1
25 MakeRecord 2 0
26 PutIntKey 0 1
27 Goto 0 16
28 ListReset 0 0
29 Close 0 0
30 Commit 0 0
31 Halt 0 0`
```

This program is essentially the same as the DELETE program except that the body of the second loop has been replace by a sequence of instructions (at addresses 17 through 26) that update the record rather than delete it. Most of this instruction sequence should already be familiar to you, but there are a couple of minor twists so we will go over it briefly. Also note that the order of some of the instructions before and after the 2nd loop has changed. This is just the way the SQLite parser chose to output the code using a different template.

As we enter the interior of the second loop (at instruction 17) the stack contains a single integer which is the key of the record we want to modify. We are going to need to use this key twice: once to fetch the old value of the record and a second time to write back the revised record. So the first instruction is a Dup to make a duplicate of the key on the top of the stack. The Dup instruction will duplicate any element of the stack, not just the top element. You specify which element to duplication using the P1 operand. When P1 is 0, the top of the stack is duplicated. When P1 is 1, the next element down on the stack duplication. And so forth.

After duplicating the key, the next instruction, NotExists, pops the stack once and uses the value popped as a key to check the existence of a record in the database file. If there is no record for this key, it jumps back to the ListRead to get another key.

Instructions 19 through 25 construct a new database record that will be used to replace the existing record. This is the same kind of code that we saw in the description of INSERT and will not be described further. After instruction 25 executes, the stack looks like this:

> | (record) new data record | | (integer) key |

The PutIntKey instruction (also described during the discussion about INSERT) writes an entry into the database file whose data is the top of the stack and whose key is the next on the stack, and then pops the stack twice. The PutIntKey instruction will overwrite the data of an existing record with the same key, which is what we want here. Overwriting was not an issue with INSERT because with INSERT the key was generated by the NewRecno instruction which is guaranteed to provide a key that has not been used before.

CREATE and DROP

Using CREATE or DROP to create or destroy a table or index is really the same as doing an INSERT or DELETE from the special "sqlite_master" table, at least from the point of view of the VDBE. The sqlite_master table is a special table that is automatically created for every SQLite database. It looks like this:

```
CREATE TABLE sqlite_master (
  type      TEXT,    -- either "table" or "index"
  name      TEXT,    -- name of this table or index
  tbl_name  TEXT,    -- for indices: name of associated table
  sql       TEXT     -- SQL text of the original CREATE statement
)
```

Every table (except the "sqlite_master" table itself) and every named index in an SQLite database has an entry in the sqlite_master table. You can query this table using a SELECT statement just like any other table. But you are not allowed to directly change the table using UPDATE, INSERT, or DELETE. Changes to sqlite_master have to occur using the CREATE and DROP commands because SQLite also has to update some of its internal data structures when tables and indices are added or destroyed.

But from the point of view of the VDBE, a CREATE works pretty much like an INSERT and a DROP works like a DELETE. When the SQLite library opens to an existing database, the first thing it does is a SELECT to read the "sql" columns from all entries of the sqlite_master table. The "sql" column contains the complete SQL text of the CREATE statement that originally generated the index or table. This text is fed back into the SQLite parser and used to reconstruct the internal data structures describing the index or table.

Using Indexes To Speed Searching

In the example queries above, every row of the table being queried must be loaded off of the disk and examined, even if only a small percentage of the rows end up in the result. This can take a long time on a big table. To speed things up, SQLite can use an index.

An SQLite file associates a key with some data. For an SQLite table, the database file is set up so that the key is an integer and the data is the information for one row of the table. Indices in SQLite reverse this arrangement. The index key is (some of) the information being stored and the index data is an integer. To access a table row that has some particular content, we first look up the content in the index table to find its integer index, then we use that integer to look up the complete record in the table.

Note that SQLite uses b-trees, which are a sorted data structure, so indices can be used when the WHERE clause of the SELECT statement contains tests for equality or inequality. Queries like the following can use an index if it is available:

```
SELECT * FROM examp WHERE two==50;
SELECT * FROM examp WHERE two&lt;50; select="" *="" from="" examp="" where="" two=""
```

If there exists an index that maps the "two" column of the "examp" table into integers, then SQLite will use that index to find the integer keys of all rows in examp that have a value of 50 for column two, or all rows that are less than 50, etc. But the following queries cannot use the index:

```
SELECT * FROM examp WHERE two%50 == 10;
SELECT * FROM examp WHERE two&127 == 3;
```

Note that the SQLite parser will not always generate code to use an index, even if it is possible to do so. The following queries will not currently use the index:

```
SELECT * FROM examp WHERE two+10 == 50;
SELECT * FROM examp WHERE two==50 OR two==100;
```

To understand better how indices work, lets first look at how they are created. Let's go ahead and put an index on the two column of the examp table. We have:

```
CREATE INDEX examp_idx1 ON examp(two);
```

The VDBE code generated by the above statement looks like the following:

```
`addr opcode p1 p2 p3

0 Transaction 1 0
1 Transaction 0 0
2 VerifyCookie 0 178
3 Integer 0 0
4 OpenWrite 0 2
5 NewRecno 0 0
6 String 0 0 index
7 String 0 0 examp_idx1
8 String 0 0 examp
9 CreateIndex 0 0 ptr(0x791380)
10 Dup 0 0
11 Integer 0 0
12 OpenWrite 1 0
13 String 0 0 CREATE INDEX examp_idx1 ON examp(tw 14 MakeRecord 5 0
15 PutIntKey 0 0
16 Integer 0 0
17 OpenRead 2 3 examp
18 Rewind 2 24
19 Recno 2 0
20 Column 2 1
21 MakeIdxKey 1 0 n
22 IdxPut 1 0 indexed columns are not unique
23 Next 2 19
24 Close 2 0
25 Close 1 0
26 Integer 333 0
27 SetCookie 0 0
28 Close 0 0
29 Commit 0 0
30 Halt 0 0`
```

Remember that every table (except sqlite_master) and every named index has an entry in the sqlite_master table. Since we are creating a new index, we have to add a new entry to sqlite_master. This is handled by instructions 3 through 15. Adding an entry to sqlite_master works just like any other INSERT statement so we will not say any more about it here. In this example, we want to focus on populating the new index with valid data, which happens on instructions 16 through 23.

```
16 Integer 0 0 17 OpenRead 2 3 examp
```

The first thing that happens is that we open the table being indexed for reading. In order to construct an index for a table, we have to know what is in that table. The index has already been opened for writing using cursor 0 by instructions 3 and 4.

```
18 Rewind 2 24 19 Recno 2 0 20 Column 2 1 21 MakeIdxKey 1 0 n 22 IdxPut 1 0 indexed colu
```

Instructions 18 through 23 implement a loop over every row of the table being indexed. For each table row, we first extract the integer key for that row using Recno in instruction 19, then get the value of the "two" column using Column in instruction 20. The MakeIdxKey instruction at 21 converts data from the "two" column (which is on the top of the stack) into a valid index key. For an index on a single column, this is basically a no-op. But if the P1 operand to MakeIdxKey had been greater than one multiple entries would have been popped from the stack and converted into a single index key. The IdxPut instruction at 22 is what actually creates the index entry. IdxPut pops two elements from the stack. The top of the stack is used as a key to fetch an entry from the index table. Then the integer which was second on stack is added to the set of integers for that index and the new record is written back to the database file. Note that the same index entry can store multiple integers if there are two or more table entries with the same value for the two column.

Now let's look at how this index will be used. Consider the following query:

```
SELECT * FROM examp WHERE two==50;
```

SQLite generates the following VDBE code to handle this query:

> `addr opcode p1 p2 p3
>
> ---
>
> 0 ColumnName 0 0 one
> 1 ColumnName 1 0 two
> 2 Integer 0 0
> 3 OpenRead 0 3 examp
> 4 VerifyCookie 0 256
> 5 Integer 0 0
> 6 OpenRead 1 4 examp_idx1
> 7 Integer 50 0 50
> 8 MakeKey 1 0 n
> 9 MemStore 0 0
> 10 MoveTo 1 19
> 11 MemLoad 0 0
> 12 IdxGT 1 19
> 13 IdxRecno 1 0
> 14 MoveTo 0 0
> 15 Column 0 0
> 16 Column 0 1
> 17 Callback 2 0
> 18 Next 1 11
> 19 Close 0 0
> 20 Close 1 0
> 21 Halt 0 0`

The SELECT begins in a familiar fashion. First the column names are initialized and the table being queried is opened. Things become different beginning with instructions 5 and 6 where the index file is also opened. Instructions 7 and 8 make a key with the value of 50. The MemStore instruction at 9 stores the index key in VDBE memory location 0. The VDBE memory is used to avoid having to fetch a value from deep in the stack, which can be done, but makes the program harder to generate. The following instruction MoveTo at address 10 pops the key off the stack and moves the index cursor to the first row of the index with that key. This initializes the cursor for use in the following loop.

Instructions 11 through 18 implement a loop over all index records with the key that was fetched by instruction 8. All of the index records with this key will be contiguous in the index table, so we walk through them and fetch the corresponding table key from the index. This table key is then used to move the cursor to that row in the table. The rest of the loop is the same as the loop for the non-indexed SELECT query.

The loop begins with the MemLoad instruction at 11 which pushes a copy of the index key back onto the stack. The instruction IdxGT at 12 compares the key to the key in the current index record pointed to by cursor P1. If the index key at the current cursor location is greater than the index we are looking for, then jump out of the loop.

The instruction IdxRecno at 13 pushes onto the stack the table record number from the index. The following MoveTo pops it and moves the table cursor to that row. The next 3 instructions select the column data the same way as in the non- indexed case. The Column instructions fetch the column data and the callback function is invoked. The final Next instruction advances the index cursor, not the table cursor, to the next row, and then branches back to the start of the loop if there are any index records left.

Since the index is used to look up values in the table, it is important that the index and table be kept consistent. Now that there is an index on the examp table, we will have to update that index whenever data is inserted, deleted, or changed in the examp table. Remember the first example above where we were able to insert a new row into the "examp" table using 12 VDBE instructions. Now that this table is indexed, 19 instructions are required. The SQL statement is this:

```
INSERT INTO examp VALUES('Hello, World!',99);
```

And the generated code looks like this:

> `addr opcode p1 p2 p3
>
> ---
>
> 0 Transaction 1 0
> 1 Transaction 0 0
> 2 VerifyCookie 0 256
> 3 Integer 0 0
> 4 OpenWrite 0 3 examp
> 5 Integer 0 0
> 6 OpenWrite 1 4 examp_idx1
> 7 NewRecno 0 0
> 8 String 0 0 Hello, World!
> 9 Integer 99 0 99
> 10 Dup 2 1
> 11 Dup 1 1
> 12 MakeIdxKey 1 0 n
> 13 IdxPut 1 0
> 14 MakeRecord 2 0
> 15 PutIntKey 0 1
> 16 Close 0 0
> 17 Close 1 0
> 18 Commit 0 0
> 19 Halt 0 0`

At this point, you should understand the VDBE well enough to figure out on your own how the above program works. So we will not discuss it further in this text.

Joins

In a join, two or more tables are combined to generate a single result. The result table consists of every possible combination of rows from the tables being joined. The easiest and most natural way to implement this is with nested loops.

Recall the query template discussed above where there was a single loop that searched through every record of the table. In a join we have basically the same thing except that there are nested loops. For example, to join two tables, the query template might look something like this:

1. Initialize the **azColumnName[]** array for the callback.
2. Open two cursors, one to each of the two tables being queried.
3. For each record in the first table, do:

    i. For each record in the second table do:
        i. If the WHERE clause evaluates to FALSE, then skip the steps that follow and continue to the next record.
        ii. Compute all columns for the current row of the result.
        iii. Invoke the callback function for the current row of the result.
4. Close both cursors.

This template will work, but it is likely to be slow since we are now dealing with an O(N<sup>2</sup>) loop. But it often works out that the WHERE clause can be factored into terms and that one or more of those terms will involve only columns in the first table. When this happens, we can factor part of the WHERE clause test out of the inner loop and gain a lot of efficiency. So a better template would be something like this:

1. Initialize the **azColumnName[]** array for the callback.
2. Open two cursors, one to each of the two tables being queried.
3. For each record in the first table, do:
    i. Evaluate terms of the WHERE clause that only involve columns from the first table. If any term is false (meaning that the whole WHERE clause must be false) then skip the rest of this loop and continue to the next record.
    ii. For each record in the second table do:
        i. If the WHERE clause evaluates to FALSE, then skip the steps that follow and continue to the next record.
        ii. Compute all columns for the current row of the result.
        iii. Invoke the callback function for the current row of the result.
4. Close both cursors.

Additional speed-up can occur if an index can be used to speed the search of either or the two loops.

SQLite always constructs the loops in the same order as the tables appear in the FROM clause of the SELECT statement. The left-most table becomes the outer loop and the right-most table becomes the inner loop. It is possible, in theory, to reorder the loops in some circumstances to speed the evaluation of the join. But SQLite does not attempt this optimization.

You can see how SQLite constructs nested loops in the following example:

> ```
> CREATE TABLE examp2(three int, four int);
> SELECT * FROM examp, examp2 WHERE two&lt;50 and="" four="=two;" &lt;="" pre=""&gt;
> ```
>
> `addr opcode p1 p2 p3
>
> ---
>
> 0 ColumnName 0 0 examp.one
>
> 1 ColumnName 1 0 examp.two
>
> 2 ColumnName 2 0 examp2.three
>
> 3 ColumnName 3 0 examp2.four
>
> 4 Integer 0 0
>
> 5 OpenRead 0 3 examp
>
> 6 VerifyCookie 0 909
>
> 7 Integer 0 0
>
> 8 OpenRead 1 5 examp2
>
> 9 Rewind 0 24
>
> 10 Column 0 1
>
> 11 Integer 50 0 50
>
> 12 Ge 1 23
>
> 13 Rewind 1 23
>
> 14 Column 1 1
>
> 15 Column 0 1
>
> 16 Ne 1 22
>
> 17 Column 0 0
>
> 18 Column 0 1
>
> 19 Column 1 0
>
> 20 Column 1 1
>
> 21 Callback 4 0
>
> 22 Next 1 14
>
> 23 Next 0 10
>
> 24 Close 0 0
>
> 25 Close 1 0
>
> 26 Halt 0 0`

The outer loop over table examp is implement by instructions 7 through 23. The inner loop is instructions 13 through 22. Notice that the "two<50" 10="" 14="" 16="" term="" of="" the="" where="" expression="" involves="" only="" columns="" from="" first="" table="" and="" can="" be="" factored="" out="" inner="" loop.="" sqlite="" does="" this="" implements="" "two<50"="" test="" in="" instructions="" through="" 12.="" "four="=two"" is="" implement="" by="" loop.<="" p="">

SQLite does not impose any arbitrary limits on the tables in a join. It also allows a table to be joined with itself.

The ORDER BY clause

For historical reasons, and for efficiency, all sorting is currently done in memory.

SQLite implements the ORDER BY clause using a special set of instructions to control an object called a sorter. In the inner-most loop of the query, where there would normally be a Callback instruction, instead a record is constructed that contains both callback parameters and a key. This record is added to the sorter (in a linked list). After the query loop finishes, the list of records is sorted and this list is walked. For each record on the list, the callback is invoked. Finally, the sorter is closed and memory is deallocated.

We can see the process in action in the following query:

> ```
> SELECT * FROM examp ORDER BY one DESC, two;
> ```
>
> ```
> `addr opcode p1 p2 p3
> ```
>
> ---
>
> ```
> 0 ColumnName 0 0 one
> 1 ColumnName 1 0 two
> 2 Integer 0 0
> 3 OpenRead 0 3 examp
> 4 VerifyCookie 0 909
> 5 Rewind 0 14
> 6 Column 0 0
> 7 Column 0 1
> 8 SortMakeRec 2 0
> 9 Column 0 0
> 10 Column 0 1
> 11 SortMakeKey 2 0 D+
> 12 SortPut 0 0
> 13 Next 0 6
> 14 Close 0 0
> 15 Sort 0 0
> 16 SortNext 0 19
> 17 SortCallback 2 0
> 18 Goto 0 16
> 19 SortReset 0 0
> 20 Halt 0 0`
> ```

There is only one sorter object, so there are no instructions to open or close it. It is opened automatically when needed, and it is closed when the VDBE program halts.

The query loop is built from instructions 5 through 13. Instructions 6 through 8 build a record that contains the azData[] values for a single invocation of the callback. A sort key is generated by instructions 9 through 11. Instruction 12 combines the invocation record and the sort key into a single entry and puts that entry on the sort list.

The P3 argument of instruction 11 is of particular interest. The sort key is formed by prepending one character from P3 to each string and concatenating all the strings. The sort comparison function will look at this character to determine whether the sort order is ascending or descending, and whether to sort as a string or number. In this example, the first column should be sorted as a string in descending order so its prefix is "D" and the second column should sorted numerically in ascending order so its prefix is "+". Ascending string sorting uses "A", and descending numeric sorting uses "-".

After the query loop ends, the table being queried is closed at instruction 14. This is done early in order to allow other processes or threads to access that table, if desired. The list of records that was built up inside the query loop is sorted by the instruction at 15. Instructions 16 through 18 walk through the record list (which is now in sorted order) and invoke the callback once for each record. Finally, the sorter is closed at instruction 19.

Aggregate Functions And The GROUP BY and HAVING Clauses

To compute aggregate functions, the VDBE implements a special data structure and instructions for controlling that data structure. The data structure is an unordered set of buckets, where each bucket has a key and one or more memory locations. Within the query loop, the GROUP BY clause is used to construct a key and the bucket with that key is brought into focus. A new bucket is created with the key if one did not previously exist. Once the bucket is in focus, the memory locations of the bucket are used to accumulate the values of the various aggregate functions. After the query loop terminates, each bucket is visited once to generate a single row of the results.

An example will help to clarify this concept. Consider the following query:

```
SELECT three, min(three+four)+avg(four)
FROM examp2
GROUP BY three;
```

The VDBE code generated for this query is as follows:

```
`addr opcode p1 p2 p3

0 ColumnName 0 0 three
1 ColumnName 1 0 min(three+four)+avg(four)
2 AggReset 0 3
3 AggInit 0 1 ptr(0x7903a0)
4 AggInit 0 2 ptr(0x790700)
5 Integer 0 0
6 OpenRead 0 5 examp2
7 VerifyCookie 0 909
8 Rewind 0 23
9 Column 0 0
10 MakeKey 1 0 n
11 AggFocus 0 14
12 Column 0 0
13 AggSet 0 0
14 Column 0 0
15 Column 0 1
16 Add 0 0
17 Integer 1 0
18 AggFunc 0 1 ptr(0x7903a0)
19 Column 0 1
20 Integer 2 0
21 AggFunc 0 1 ptr(0x790700)
22 Next 0 9
23 Close 0 0
24 AggNext 0 31
25 AggGet 0 0
26 AggGet 0 1
27 AggGet 0 2
28 Add 0 0
29 Callback 2 0
30 Goto 0 24
31 Noop 0 0
32 Halt 0 0`
```

The first instruction of interest is the AggReset at 2. The AggReset instruction initializes the set of buckets to be the empty set and specifies the number of memory slots available in each bucket as P2. In this example, each bucket will hold 3 memory slots. It is not obvious, but if you look closely at the rest of the program you can figure out what each of these slots is intended for.

Memory Slot	Intended Use Of This Memory Slot
0	The "three" column -- the key to the bucket
1	The minimum "three+four" value
2	The sum of all "four" values. This is used to compute "avg(four)".

The query loop is implemented by instructions 8 through 22. The aggregate key specified by the GROUP BY clause is computed by instructions 9 and 10. Instruction 11 causes the appropriate bucket to come into focus. If a bucket with the given key does not already exists, a new bucket is created and control falls through to instructions 12 and 13 which initialize the bucket. If the bucket does already exist, then a jump is made to instruction 14. The values of aggregate functions are updated by the instructions between 11 and 21. Instructions 14 through 18 update memory slot 1 to hold the next value "min(three+four)". Then the sum of the "four" column is updated by instructions 19 through 21.

After the query loop is finished, the table "examp2" is closed at instruction 23 so that its lock will be released and it can be used by other threads or processes. The next step is to loop over all aggregate buckets and output one row of the result for each bucket. This is done by the loop at instructions 24 through 30. The AggNext instruction at 24 brings the next bucket into focus, or jumps to the end of the loop if all buckets have been examined already. The 3 columns of the result are fetched from the aggregator bucket in order at instructions 25 through 27. Finally, the callback is invoked at instruction 29.

In summary then, any query with aggregate functions is implemented by two loops. The first loop scans the input table and computes aggregate information into buckets and the second loop scans through all the buckets to compute the final result.

The realization that an aggregate query is really two consecutive loops makes it much easier to understand the difference between a WHERE clause and a HAVING clause in SQL query statement. The WHERE clause is a restriction on the first loop and the HAVING clause is a restriction on the second loop. You can see this by adding both a WHERE and a HAVING clause to our example query:

```
SELECT three, min(three+four)+avg(four)
FROM examp2
WHERE three&gt;four
GROUP BY three
HAVING avg(four)&lt;10; &lt;="" pre=""&gt;
```

> `addr opcode p1 p2 p3

```
0 ColumnName 0 0 three
1 ColumnName 1 0 min(three+four)+avg(four)
2 AggReset 0 3
3 AggInit 0 1 ptr(0x7903a0)
4 AggInit 0 2 ptr(0x790700)
5 Integer 0 0
6 OpenRead 0 5 examp2
7 VerifyCookie 0 909
8 Rewind 0 26
9 Column 0 0
10 Column 0 1
11 Le 1 25
12 Column 0 0
13 MakeKey 1 0 n
14 AggFocus 0 17
15 Column 0 0
16 AggSet 0 0
17 Column 0 0
18 Column 0 1
19 Add 0 0
20 Integer 1 0
21 AggFunc 0 1 ptr(0x7903a0)
22 Column 0 1
23 Integer 2 0
24 AggFunc 0 1 ptr(0x790700)
25 Next 0 9
26 Close 0 0
27 AggNext 0 37
28 AggGet 0 2
29 Integer 10 0 10
30 Ge 1 27
31 AggGet 0 0
32 AggGet 0 1
33 AggGet 0 2
34 Add 0 0
35 Callback 2 0
36 Goto 0 27
37 Noop 0 0
38 Halt 0 0`
```

The code generated in this last example is the same as the previous except for the addition of two conditional jumps used to implement the extra WHERE and HAVING clauses. The WHERE clause is implemented by instructions 9 through 11 in the query loop. The HAVING clause is implemented by instruction 28 through 30 in the output loop.

Using SELECT Statements As Terms In An Expression

The very name "Structured Query Language" tells us that SQL should support nested queries. And, in fact, two different kinds of nesting are supported. Any SELECT statement that returns a single-row, single-column result can be used as a term in an expression of another SELECT statement. And, a SELECT statement that returns a single-column, multi-row result can be used as the right-hand operand of the IN and NOT IN operators. We will begin this section with an example of the first kind of nesting, where a single-row, single-column SELECT is used as a term in an expression of another SELECT. Here is our example:

```
SELECT * FROM examp
WHERE two!=(SELECT three FROM examp2
            WHERE four=5);
```

The way SQLite deals with this is to first run the inner SELECT (the one against examp2) and store its result in a private memory cell. SQLite then substitutes the value of this private memory cell for the inner SELECT when it evaluates the outer SELECT. The code looks like this:

```
`addr opcode p1 p2 p3

0 String 0 0
1 MemStore 0 1
2 Integer 0 0
3 OpenRead 1 5 examp2
4 VerifyCookie 0 909
5 Rewind 1 13
6 Column 1 1
7 Integer 5 0 5
8 Ne 1 12
9 Column 1 0
10 MemStore 0 1
11 Goto 0 13
12 Next 1 6
13 Close 1 0
14 ColumnName 0 0 one
15 ColumnName 1 0 two
16 Integer 0 0
17 OpenRead 0 3 examp
18 Rewind 0 26
19 Column 0 1
20 MemLoad 0 0
21 Eq 1 25
22 Column 0 0
23 Column 0 1
24 Callback 2 0
25 Next 0 19
26 Close 0 0
27 Halt 0 0`
```

The private memory cell is initialized to NULL by the first two instructions. Instructions 2 through 13 implement the inner SELECT statement against the examp2 table. Notice that instead of sending the result to a callback or storing the result on a sorter, the result of the query is pushed into the memory cell by instruction 10 and the loop is abandoned by the jump at instruction 11. The jump at instruction at 11 is vestigial and never executes.

The outer SELECT is implemented by instructions 14 through 25. In particular, the WHERE clause that contains the nested select is implemented by instructions 19 through 21. You can see that the result of the inner select is loaded onto the stack by instruction 20 and used by the conditional jump at 21.

When the result of a sub-select is a scalar, a single private memory cell can be used, as shown in the previous example. But when the result of a sub-select is a vector, such as when the sub-select is the right-hand operand of IN or NOT IN, a different approach is needed. In this case, the result of the sub-select is stored in a transient table and the contents of that table are tested using the Found or NotFound operators. Consider this example:

```
SELECT * FROM examp
WHERE two IN (SELECT three FROM examp2);
```

The code generated to implement this last query is as follows:

```
`addr opcode p1 p2 p3

0 OpenTemp 1 1
1 Integer 0 0
2 OpenRead 2 5 examp2
3 VerifyCookie 0 909
4 Rewind 2 10
5 Column 2 0
6 IsNull -1 9
7 String 0 0
8 PutStrKey 1 0
9 Next 2 5
10 Close 2 0
11 ColumnName 0 0 one
12 ColumnName 1 0 two
13 Integer 0 0
14 OpenRead 0 3 examp
15 Rewind 0 25
16 Column 0 1
17 NotNull -1 20
18 Pop 1 0
19 Goto 0 24
20 NotFound 1 24
21 Column 0 0
22 Column 0 1
23 Callback 2 0
24 Next 0 16
25 Close 0 0
26 Halt 0 0`
```

The transient table in which the results of the inner SELECT are stored is created by the OpenTemp instruction at 0. This opcode is used for tables that exist for the duration of a single SQL statement only. The transient cursor is always opened read/write even if the main database is read-only. The transient table is deleted automatically when the cursor is closed. The P2 value of 1 means the cursor points to a BTree index, which has no data but can have an arbitrary key.

The inner SELECT statement is implemented by instructions 1 through 10. All this code does is make an entry in the temporary table for each row of the examp2 table with a non-NULL value for the "three" column. The key for each temporary table entry is the "three" column of examp2 and the data is an empty string since it is never used.

The outer SELECT is implemented by instructions 11 through 25. In particular, the WHERE clause containing the IN operator is implemented by instructions at 16, 17, and 20. Instruction 16 pushes the value of the "two" column for the current row onto the stack and instruction 17 checks to see that it is non-NULL. If this is successful, execution jumps to 20, where it tests to see if top of the stack matches any key in the temporary table. The rest of the code is the same as what has been shown before.

Compound SELECT Statements

SQLite also allows two or more SELECT statements to be joined as peers using operators UNION, UNION ALL, INTERSECT, and EXCEPT. These compound select statements are implemented using transient tables. The implementation is slightly different for each operator, but the basic ideas are the same. For an example we will use the EXCEPT operator.

```
SELECT two FROM examp
EXCEPT
SELECT four FROM examp2;
```

The result of this last example should be every unique value of the "two" column in the examp table, except any value that is in the "four" column of examp2 is removed. The code to implement this query is as follows:

```
`addr opcode p1 p2 p3

0 OpenTemp 0 1
1 KeyAsData 0 1
2 Integer 0 0
3 OpenRead 1 3 examp
4 VerifyCookie 0 909
5 Rewind 1 11
6 Column 1 1
7 MakeRecord 1 0
8 String 0 0
9 PutStrKey 0 0
10 Next 1 6
11 Close 1 0
12 Integer 0 0
13 OpenRead 2 5 examp2
14 Rewind 2 20
15 Column 2 1
16 MakeRecord 1 0
17 NotFound 0 19
18 Delete 0 0
19 Next 2 15
20 Close 2 0
21 ColumnName 0 0 four
22 Rewind 0 26
23 Column 0 0
24 Callback 1 0
25 Next 0 23
26 Close 0 0
27 Halt 0 0`
```

The transient table in which the result is built is created by instruction 0. Three loops then follow. The loop at instructions 5 through 10 implements the first SELECT statement. The second SELECT statement is implemented by the loop at instructions 14 through 19. Finally, a loop at instructions 22 through 25 reads the transient table and invokes the callback once for each row in the result.

Instruction 1 is of particular importance in this example. Normally, the Column instruction extracts the value of a column from a larger record in the data of an SQLite file entry. Instruction 1 sets a flag on the transient table so that Column will instead treat the key of the

SQLite file entry as if it were data and extract column information from the key.

Here is what is going to happen: The first SELECT statement will construct rows of the result and save each row as the key of an entry in the transient table. The data for each entry in the transient table is a never used so we fill it in with an empty string. The second SELECT statement also constructs rows, but the rows constructed by the second SELECT are removed from the transient table. That is why we want the rows to be stored in the key of the SQLite file instead of in the data -- so they can be easily located and deleted.

Let's look more closely at what is happening here. The first SELECT is implemented by the loop at instructions 5 through 10. Instruction 5 initializes the loop by rewinding its cursor. Instruction 6 extracts the value of the "two" column from "examp" and instruction 7 converts this into a row. Instruction 8 pushes an empty string onto the stack. Finally, instruction 9 writes the row into the temporary table. But remember, the PutStrKey opcode uses the top of the stack as the record data and the next on stack as the key. For an INSERT statement, the row generated by the MakeRecord opcode is the record data and the record key is an integer created by the NewRecno opcode. But here the roles are reversed and the row created by MakeRecord is the record key and the record data is just an empty string.

The second SELECT is implemented by instructions 14 through 19. Instruction 14 initializes the loop by rewinding its cursor. A new result row is created from the "four" column of table "examp2" by instructions 15 and 16. But instead of using PutStrKey to write this new row into the temporary table, we instead call Delete to remove it from the temporary table if it exists.

The result of the compound select is sent to the callback routine by the loop at instructions 22 through 25. There is nothing new or remarkable about this loop, except for the fact that the Column instruction at 23 will be extracting a column out of the record key rather than the record data.

Summary

This article has reviewed all of the major techniques used by SQLite's VDBE to implement SQL statements. What has not been shown is that most of these techniques can be used in combination to generate code for an appropriately complex query statement. For example, we have shown how sorting is accomplished on a simple query and we have shown how to implement a compound query. But we did not give an example of sorting in a compound query. This is because sorting a compound query does not introduce any new concepts: it merely combines two previous ideas (sorting and compounding) in the same VDBE program.

For additional information on how the SQLite library functions, the reader is directed to look at the SQLite source code directly. If you understand the material in this article, you should not have much difficulty in following the sources. Serious students of the internals of SQLite will probably also want to make a careful study of the VDBE opcodes as documented here. Most of the opcode documentation is extracted from comments in the source code using a script so you can also get information about the various opcodes directly from the **vdbe.c** source file. If you have successfully read this far, you should have little difficulty understanding the rest.

If you find errors in either the documentation or the code, feel free to fix them and/or contact the author at drh@hwaci.com. Your bug fixes or suggestions are always welcomed.

SQLite Version 3 Overview

Editorial Note: This document was written in 2004 as a guide to programmers who were transitioning from SQLite2 to SQLite3. It is retained as part of the historical record of SQLite. Modern programmers should refer to more up-to-date documentation on SQLite is available elsewhere on this website.

SQLite version 3.0 introduces important changes to the library, including:

- A more compact format for database files.
- Manifest typing and BLOB support.
- Support for both UTF-8 and UTF-16 text.
- User-defined text collating sequences.
- 64-bit ROWIDs.
- Improved Concurrency.

This document is a quick introduction to the changes for SQLite 3.0 for users who are already familiar with SQLite version 2.8.

Naming Changes

SQLite version 2.8 will continue to be supported with bug fixes for the foreseeable future. In order to allow SQLite version 2.8 and SQLite version 3.0 to peacefully coexist, the names of key files and APIs in SQLite version 3.0 have been changed to include the character "3". For example, the include file used by C programs has been changed from "sqlite.h" to "sqlite3.h". And the name of the shell program used to interact with databases has been changed from "sqlite.exe" to "sqlite3.exe". With these changes, it is possible to have both SQLite 2.8 and SQLite 3.0 installed on the same system at the same time. And it is possible for the same C program to link against both SQLite 2.8 and SQLite 3.0 at the same time and to use both libraries at the same time.

New File Format

The format used by SQLite database files has been completely revised. The old version 2.1 format and the new 3.0 format are incompatible with one another. Version 2.8 of SQLite will not read a version 3.0 database files and version 3.0 of SQLite will not read a version 2.8 database file.

To convert an SQLite 2.8 database into an SQLite 3.0 database, have ready the command-line shells for both version 2.8 and 3.0. Then enter a command like the following:

```
sqlite OLD.DB .dump | sqlite3 NEW.DB
```

The new database file format uses B+trees for tables. In a B+tree, all data is stored in the leaves of the tree instead of in both the leaves and the intermediate branch nodes. The use of B+trees for tables allows for better scalability and the storage of larger data fields without the use of overflow pages. Traditional B-trees are still used for indices.

The new file format also supports variable pages sizes between 512 and 65536 bytes. The size of a page is stored in the file header so the same library can read databases with different pages sizes, in theory, though this feature has not yet been implemented in practice.

The new file format omits unused fields from its disk images. For example, indices use only the key part of a B-tree record and not the data. So for indices, the field that records the length of the data is omitted. Integer values such as the length of key and data are stored using a variable-length encoding so that only one or two bytes are required to store the most common cases but up to 64-bits of information can be encoded if needed. Integer and floating point data is stored on the disk in binary rather than being converted into ASCII as in SQLite version 2.8. These changes taken together result in database files that are typically 25% to 35% smaller than the equivalent files in SQLite version 2.8.

Details of the low-level B-tree format used in SQLite version 3.0 can be found in header comments to the btreeInt.h source file and in the file format documentation.

Manifest Typing and BLOB Support

SQLite version 2.8 will deal with data in various formats internally, but when writing to the disk or interacting through its API, SQLite 2.8 always converts data into ASCII text. SQLite 3.0, in contrast, exposes its internal data representations to the user and stores binary representations to disk when appropriate. The exposing of non-ASCII representations was added in order to support BLOBs.

SQLite version 2.8 had the feature that any type of data could be stored in any table column regardless of the declared type of that column. This feature is retained in version 3.0, though in a slightly modified form. Each table column will store any type of data, though columns have an affinity for the format of data defined by their declared datatype. When data is inserted into a column, that column will make an attempt to convert the data format into the column's declared type. All SQL database engines do this. The difference is that SQLite 3.0 will still store the data even if a format conversion is not possible.

For example, if you have a table column declared to be of type "INTEGER" and you try to insert a string, the column will look at the text string and see if it looks like a number. If the string does look like a number it is converted into a number and into an integer if the number does not have a fractional part, and stored that way. But if the string is not a well-formed number it is still stored as a string. A column with a type of "TEXT" tries to convert numbers into an ASCII-Text representation before storing them. But BLOBs are stored in TEXT columns as BLOBs because you cannot in general convert a BLOB into text.

In most other SQL database engines the datatype is associated with the table column that holds the data - with the data container. In SQLite 3.0, the datatype is associated with the data itself, not with its container. Paul Graham in his book *ANSI Common Lisp* calls this property "Manifest Typing". Other writers have other definitions for the term "manifest typing", so beware of confusion. But by whatever name, that is the datatype model supported by SQLite 3.0.

Additional information about datatypes in SQLite version 3.0 is available separately.

Support for UTF-8 and UTF-16

The new API for SQLite 3.0 contains routines that accept text as both UTF-8 and UTF-16 in the native byte order of the host machine. Each database file manages text as either UTF-8, UTF-16BE (big-endian), or UTF-16LE (little-endian). Internally and in the disk file, the same text representation is used everywhere. If the text representation specified by the database file (in the file header) does not match the text representation required by the interface routines, then text is converted on-the-fly. Constantly converting text from one representation to another can be computationally expensive, so it is suggested that programmers choose a single representation and stick with it throughout their application.

In the current implementation of SQLite, the SQL parser only works with UTF-8 text. So if you supply UTF-16 text it will be converted. This is just an implementation issue and there is nothing to prevent future versions of SQLite from parsing UTF-16 encoded SQL natively.

When creating new user-defined SQL functions and collating sequences, each function or collating sequence can specify if it works with UTF-8, UTF-16be, or UTF-16le. Separate implementations can be registered for each encoding. If an SQL function or collating sequence is required but a version for the current text encoding is not available, then the text is automatically converted. As before, this conversion takes computation time, so programmers are advised to pick a single encoding and stick with it in order to minimize the amount of unnecessary format juggling.

SQLite is not particular about the text it receives and is more than happy to process text strings that are not normalized or even well-formed UTF-8 or UTF-16. Thus, programmers who want to store IS08859 data can do so using the UTF-8 interfaces. As long as no attempts are made to use a UTF-16 collating sequence or SQL function, the byte sequence of the text will not be modified in any way.

User-defined Collating Sequences

A collating sequence is just a defined order for text. When SQLite 3.0 sorts (or uses a comparison operator like "<" or ">=") the sort order is first determined by the data type.

- NULLs sort first
- Numeric values sort next in numerical order
- Text values come after numerics
- BLOBs sort last

Collating sequences are used for comparing two text strings. The collating sequence does not change the ordering of NULLs, numbers, or BLOBs, only text.

A collating sequence is implemented as a function that takes the two strings being compared as inputs and returns negative, zero, or positive if the first string is less than, equal to, or greater than the second. SQLite 3.0 comes with a single built-in collating sequence named "BINARY" which is implemented using the memcmp() routine from the standard C library. The BINARY collating sequence works well for English text. For other languages or locales, alternative collating sequences may be preferred.

The decision of which collating sequence to use is controlled by the COLLATE clause in SQL. A COLLATE clause can occur on a table definition, to define a default collating sequence to a table column, or on field of an index, or in the ORDER BY clause of a SELECT statement. Planned enhancements to SQLite are to include standard CAST() syntax to allow the collating sequence of an expression to be defined.

64-bit ROWIDs

Every row of a table has a unique rowid. If the table defines a column with the type "INTEGER PRIMARY KEY" then that column becomes an alias for the rowid. But with or without an INTEGER PRIMARY KEY column, every row still has a rowid.

In SQLite version 3.0, the rowid is a 64-bit signed integer. This is an expansion of SQLite version 2.8 which only permitted rowids of 32-bits.

To minimize storage space, the 64-bit rowid is stored as a variable length integer. Rowids between 0 and 127 use only a single byte. Rowids between 0 and 16383 use just 2 bytes. Up to 2097152 uses three bytes. And so forth. Negative rowids are allowed but they always use nine bytes of storage and so their use is discouraged. When rowids are generated automatically by SQLite, they will always be non-negative.

Improved Concurrency

SQLite version 2.8 allowed multiple simultaneous readers or a single writer but not both. SQLite version 3.0 allows one process to begin writing the database while other processes continue to read. The writer must still obtain an exclusive lock on the database for a brief interval in order to commit its changes, but the exclusive lock is no longer required for the entire write operation. A more detailed report on the locking behavior of SQLite version 3.0 is available separately.

A limited form of table-level locking is now also available in SQLite. If each table is stored in a separate database file, those separate files can be attached to the main database (using the ATTACH command) and the combined databases will function as one. But locks will only be acquired on individual files as needed. So if you redefine "database" to mean two or more database files, then it is entirely possible for two processes to be writing to the same database at the same time. To further support this capability, commits of transactions involving two or more ATTACHed database are now atomic.

Credits

SQLite version 3.0 is made possible in part by AOL developers supporting and embracing great Open-Source Software.

C/C++ Interface For SQLite Version 3

Note: This document was written in 2004 as a guide to helping programmers move from using SQLite version 2 to SQLite version 3. The information in this document is still essentially correct, however there have been many changes and enhancements over the years. We recommend that the following documents be used instead:

- An Introduction To The SQLite C/C++ Interface
- SQLite C/C++ Reference Guide

1.0 Overview

SQLite version 3.0 is a new version of SQLite, derived from the SQLite 2.8.13 code base, but with an incompatible file format and API. SQLite version 3.0 was created to answer demand for the following features:

- Support for UTF-16.
- User-definable text collating sequences.
- The ability to store BLOBs in indexed columns.

It was necessary to move to version 3.0 to implement these features because each requires incompatible changes to the database file format. Other incompatible changes, such as a cleanup of the API, were introduced at the same time under the theory that it is best to get your incompatible changes out of the way all at once.

The API for version 3.0 is similar to the version 2.X API, but with some important changes. Most noticeably, the " `sqlite_` " prefix that occurs on the beginning of all API functions and data structures are changed to " `sqlite3_` ". This avoids confusion between the two APIs and allows linking against both SQLite 2.X and SQLite 3.0 at the same time.

There is no agreement on what the C datatype for a UTF-16 string should be. Therefore, SQLite uses a generic type of void *to refer to UTF-16 strings. Client software can cast the void* to whatever datatype is appropriate for their system.

2.0 C/C++ Interface

The API for SQLite 3.0 includes 83 separate functions in addition to several data structures and #defines. (A complete API reference is provided as a separate document.) Fortunately, the interface is not nearly as complex as its size implies. Simple programs can still make do with only 3 functions: sqlite3_open(), sqlite3_exec(), and sqlite3_close(). More control over the execution of the database engine is provided using sqlite3_prepare_v2() to compile an

SQLite statement into byte code and sqlite3_step() to execute that bytecode. A family of routines with names beginning with sqlite3*column* is used to extract information about the result set of a query. Many interface functions come in pairs, with both a UTF-8 and UTF-16 version. And there is a collection of routines used to implement user-defined SQL functions and user-defined text collating sequences.

2.1 Opening and closing a database

```
typedef struct sqlite3 sqlite3;
int sqlite3_open(const char*, sqlite3**);
int sqlite3_open16(const void*, sqlite3**);
int sqlite3_close(sqlite3*);
const char *sqlite3_errmsg(sqlite3*);
const void *sqlite3_errmsg16(sqlite3*);
int sqlite3_errcode(sqlite3*);
```

The sqlite3_open() routine returns an integer error code rather than a pointer to the sqlite3 structure as the version 2 interface did. The difference between sqlite3_open() and sqlite3_open16() is that sqlite3_open16() takes UTF-16 (in host native byte order) for the name of the database file. If a new database file needs to be created, then sqlite3_open16() sets the internal text representation to UTF-16 whereas sqlite3_open() sets the text representation to UTF-8.

The opening and/or creating of the database file is deferred until the file is actually needed. This allows options and parameters, such as the native text representation and default page size, to be set using PRAGMA statements.

The sqlite3_errcode() routine returns a result code for the most recent major API call. sqlite3_errmsg() returns an English-language text error message for the most recent error. The error message is represented in UTF-8 and will be ephemeral - it could disappear on the next call to any SQLite API function. sqlite3_errmsg16() works like sqlite3_errmsg() except that it returns the error message represented as UTF-16 in host native byte order.

The error codes for SQLite version 3 are unchanged from version 2. They are as follows:

```
#define SQLITE_OK           0   /* Successful result */
#define SQLITE_ERROR        1   /* SQL error or missing database */
#define SQLITE_INTERNAL     2   /* An internal logic error in SQLite */
#define SQLITE_PERM         3   /* Access permission denied */
#define SQLITE_ABORT        4   /* Callback routine requested an abort */
#define SQLITE_BUSY         5   /* The database file is locked */
#define SQLITE_LOCKED       6   /* A table in the database is locked */
#define SQLITE_NOMEM        7   /* A malloc() failed */
#define SQLITE_READONLY     8   /* Attempt to write a readonly database */
#define SQLITE_INTERRUPT    9   /* Operation terminated by sqlite_interrupt() */
#define SQLITE_IOERR       10   /* Some kind of disk I/O error occurred */
#define SQLITE_CORRUPT     11   /* The database disk image is malformed */
#define SQLITE_NOTFOUND    12   /* (Internal Only) Table or record not found */
#define SQLITE_FULL        13   /* Insertion failed because database is full */
#define SQLITE_CANTOPEN    14   /* Unable to open the database file */
#define SQLITE_PROTOCOL    15   /* Database lock protocol error */
#define SQLITE_EMPTY       16   /* (Internal Only) Database table is empty */
#define SQLITE_SCHEMA      17   /* The database schema changed */
#define SQLITE_TOOBIG      18   /* Too much data for one row of a table */
#define SQLITE_CONSTRAINT  19   /* Abort due to contraint violation */
#define SQLITE_MISMATCH    20   /* Data type mismatch */
#define SQLITE_MISUSE      21   /* Library used incorrectly */
#define SQLITE_NOLFS       22   /* Uses OS features not supported on host */
#define SQLITE_AUTH        23   /* Authorization denied */
#define SQLITE_ROW        100   /* sqlite_step() has another row ready */
#define SQLITE_DONE       101   /* sqlite_step() has finished executing */
```

2.2 Executing SQL statements

```
   typedef int (*sqlite_callback)(void*,int,char**, char**);
   int sqlite3_exec(sqlite3*, const char *sql, sqlite_callback, void*, char**);
```

The sqlite3_exec() function works much as it did in SQLite version 2. Zero or more SQL statements specified in the second parameter are compiled and executed. Query results are returned to a callback routine.

In SQLite version 3, the sqlite3_exec routine is just a wrapper around calls to the prepared statement interface.

```
   typedef struct sqlite3_stmt sqlite3_stmt;
   int sqlite3_prepare(sqlite3*, const char*, int, sqlite3_stmt**, const char**);
   int sqlite3_prepare16(sqlite3*, const void*, int, sqlite3_stmt**, const void**);
   int sqlite3_finalize(sqlite3_stmt*);
   int sqlite3_reset(sqlite3_stmt*);
```

The sqlite3_prepare interface compiles a single SQL statement into byte code for later execution. This interface is now the preferred way of accessing the database.

The SQL statement is a UTF-8 string for sqlite3_prepare(). The sqlite3_prepare16() works the same way except that it expects a UTF-16 string as SQL input. Only the first SQL statement in the input string is compiled. The fifth parameter is filled in with a pointer to the next (uncompiled) SQLite statement in the input string, if any. The sqlite3_finalize() routine deallocates a prepared SQL statement. All prepared statements must be finalized before the database can be closed. The sqlite3_reset() routine resets a prepared SQL statement so that it can be executed again.

The SQL statement may contain tokens of the form "?" or "?nnn" or ":aaa" where "nnn" is an integer and "aaa" is an identifier. Such tokens represent unspecified literal values (or "wildcards") to be filled in later by the sqlite3_bind interface. Each wildcard has an associated number which is its sequence in the statement or the "nnn" in the case of a "?nnn" form. It is allowed for the same wildcard to occur more than once in the same SQL statement, in which case all instance of that wildcard will be filled in with the same value. Unbound wildcards have a value of NULL.

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_int64(sqlite3_stmt*, int, long long int);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int n, void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_value(sqlite3_stmt*, int, const sqlite3_value*);
```

There is an assortment of sqlite3_bind routines used to assign values to wildcards in a prepared SQL statement. Unbound wildcards are interpreted as NULLs. Bindings are not reset by sqlite3_reset(). But wildcards can be rebound to new values after an sqlite3_reset().

After an SQL statement has been prepared (and optionally bound), it is executed using:

```
int sqlite3_step(sqlite3_stmt*);
```

The sqlite3_step() routine return SQLITE_ROW if it is returning a single row of the result set, or SQLITE_DONE if execution has completed, either normally or due to an error. It might also return SQLITE_BUSY if it is unable to open the database file. If the return value is SQLITE_ROW, then the following routines can be used to extract information about that row of the result set:

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
int sqlite3_column_count(sqlite3_stmt*);
const char *sqlite3_column_decltype(sqlite3_stmt *, int iCol);
const void *sqlite3_column_decltype16(sqlite3_stmt *, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);
long long int sqlite3_column_int64(sqlite3_stmt*, int iCol);
const char *sqlite3_column_name(sqlite3_stmt*, int iCol);
const void *sqlite3_column_name16(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);
```

The sqlite3_column_count() function returns the number of columns in the results set. sqlite3_column_count() can be called at any time after sqlite3_prepare_v2(). sqlite3_data_count() works similarly to sqlite3_column_count() except that it only works following sqlite3_step(). If the previous call to sqlite3_step() returned SQLITE_DONE or an error code, then sqlite3_data_count() will return 0 whereas sqlite3_column_count() will continue to return the number of columns in the result set.

Returned data is examined using the other sqlite3*column**() functions, all of which take a column number as their second parameter. Columns are zero-indexed from left to right. Note that this is different to parameters, which are indexed starting at one.

The sqlite3_column_type() function returns the datatype for the value in the Nth column. The return value is one of these:

```
    #define SQLITE_INTEGER  1
    #define SQLITE_FLOAT    2
    #define SQLITE_TEXT     3
    #define SQLITE_BLOB     4
    #define SQLITE_NULL     5
```

The sqlite3_column_decltype() routine returns text which is the declared type of the column in the CREATE TABLE statement. For an expression, the return type is an empty string. sqlite3_column_name() returns the name of the Nth column. sqlite3_column_bytes() returns the number of bytes in a column that has type BLOB or the number of bytes in a TEXT string with UTF-8 encoding. sqlite3_column_bytes16() returns the same value for BLOBs but for TEXT strings returns the number of bytes in a UTF-16 encoding. sqlite3_column_blob() return BLOB data. sqlite3_column_text() return TEXT data as UTF-8. sqlite3_column_text16() return TEXT data as UTF-16. sqlite3_column_int() return INTEGER data in the host machines native integer format. sqlite3_column_int64() returns 64-bit INTEGER data. Finally, sqlite3_column_double() return floating point data.

It is not necessary to retrieve data in the format specify by sqlite3_column_type(). If a different format is requested, the data is converted automatically.

Data format conversions can invalidate the pointer returned by prior calls to sqlite3_column_blob(), sqlite3_column_text(), and/or sqlite3_column_text16(). Pointers might be invalided in the following cases:

- The initial content is a BLOB and sqlite3_column_text() or sqlite3_column_text16() is called. A zero-terminator might need to be added to the string.
- The initial content is UTF-8 text and sqlite3_column_bytes16() or sqlite3_column_text16() is called. The content must be converted to UTF-16.

- The initial content is UTF-16 text and sqlite3_column_bytes() or sqlite3_column_text() is called. The content must be converted to UTF-8.

Note that conversions between UTF-16be and UTF-16le are always done in place and do not invalidate a prior pointer, though of course the content of the buffer that the prior pointer points to will have been modified. Other kinds of conversion are done in place when it is possible, but sometime it is not possible and in those cases prior pointers are invalidated.

The safest and easiest to remember policy is this: assume that any result from

- sqlite3_column_blob(),
- sqlite3_column_text(), or
- sqlite3_column_text16()

is invalided by subsequent calls to

- sqlite3_column_bytes(),
- sqlite3_column_bytes16(),
- sqlite3_column_text(), or
- sqlite3_column_text16().

This means that you should always call sqlite3_column_bytes() or sqlite3_column_bytes16() before calling sqlite3_column_blob(), sqlite3_column_text(), or sqlite3_column_text16().

2.3 User-defined functions

User defined functions can be created using the following routine:

```
typedef struct sqlite3_value sqlite3_value;
int sqlite3_create_function(
  sqlite3 *,
  const char *zFunctionName,
  int nArg,
  int eTextRep,
  void*,
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
int sqlite3_create_function16(
  sqlite3*,
  const void *zFunctionName,
  int nArg,
  int eTextRep,
  void*,
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
#define SQLITE_UTF8     1
#define SQLITE_UTF16    2
#define SQLITE_UTF16BE  3
#define SQLITE_UTF16LE  4
#define SQLITE_ANY      5
```

The nArg parameter specifies the number of arguments to the function. A value of 0 indicates that any number of arguments is allowed. The eTextRep parameter specifies what representation text values are expected to be in for arguments to this function. The value of this parameter should be one of the parameters defined above. SQLite version 3 allows multiple implementations of the same function using different text representations. The database engine chooses the function that minimization the number of text conversions required.

Normal functions specify only xFunc and leave xStep and xFinal set to NULL. Aggregate functions specify xStep and xFinal and leave xFunc set to NULL. There is no separate sqlite3_create_aggregate() API.

The function name is specified in UTF-8. A separate sqlite3_create_function16() API works the same as sqlite_create_function() except that the function name is specified in UTF-16 host byte order.

Notice that the parameters to functions are now pointers to sqlite3_value structures instead of pointers to strings as in SQLite version 2.X. The following routines are used to extract useful information from these "values":

```
    const void *sqlite3_value_blob(sqlite3_value*);
    int sqlite3_value_bytes(sqlite3_value*);
    int sqlite3_value_bytes16(sqlite3_value*);
    double sqlite3_value_double(sqlite3_value*);
    int sqlite3_value_int(sqlite3_value*);
    long long int sqlite3_value_int64(sqlite3_value*);
    const unsigned char *sqlite3_value_text(sqlite3_value*);
    const void *sqlite3_value_text16(sqlite3_value*);
    int sqlite3_value_type(sqlite3_value*);
```

Function implementations use the following APIs to acquire context and to report results:

```
    void *sqlite3_aggregate_context(sqlite3_context*, int nbyte);
    void *sqlite3_user_data(sqlite3_context*);
    void sqlite3_result_blob(sqlite3_context*, const void*, int n, void(*)(void*));
    void sqlite3_result_double(sqlite3_context*, double);
    void sqlite3_result_error(sqlite3_context*, const char*, int);
    void sqlite3_result_error16(sqlite3_context*, const void*, int);
    void sqlite3_result_int(sqlite3_context*, int);
    void sqlite3_result_int64(sqlite3_context*, long long int);
    void sqlite3_result_null(sqlite3_context*);
    void sqlite3_result_text(sqlite3_context*, const char*, int n, void(*)(void*));
    void sqlite3_result_text16(sqlite3_context*, const void*, int n, void(*)(void*));
    void sqlite3_result_value(sqlite3_context*, sqlite3_value*);
    void *sqlite3_get_auxdata(sqlite3_context*, int);
    void sqlite3_set_auxdata(sqlite3_context*, int, void*, void (*)(void*));
```

2.4 User-defined collating sequences

The following routines are used to implement user-defined collating sequences:

```
sqlite3_create_collation(sqlite3*, const char *zName, int eTextRep, void*,
   int(*xCompare)(void*,int,const void*,int,const void*));
sqlite3_create_collation16(sqlite3*, const void *zName, int eTextRep, void*,
   int(*xCompare)(void*,int,const void*,int,const void*));
sqlite3_collation_needed(sqlite3*, void*,
   void(*)(void*,sqlite3*,int eTextRep,const char*));
sqlite3_collation_needed16(sqlite3*, void*,
   void(*)(void*,sqlite3*,int eTextRep,const void*));
```

The sqlite3_create_collation() function specifies a collating sequence name and a comparison function to implement that collating sequence. The comparison function is only used for comparing text values. The eTextRep parameter is one of SQLITE_UTF8, SQLITE_UTF16LE, SQLITE_UTF16BE, or SQLITE_ANY to specify which text representation the comparison function works with. Separate comparison functions can exist for the same collating sequence for each of the UTF-8, UTF-16LE and UTF-16BE text representations. The sqlite3_create_collation16() works like sqlite3_create_collation() except that the collation name is specified in UTF-16 host byte order instead of in UTF-8.

The sqlite3_collation_needed() routine registers a callback which the database engine will invoke if it encounters an unknown collating sequence. The callback can lookup an appropriate comparison function and invoke sqlite_3_create_collation() as needed. The fourth parameter to the callback is the name of the collating sequence in UTF-8. For sqlite3_collation_need16() the callback sends the collating sequence name in UTF-16 host byte order.

Database Speed Comparison

Note: This document is very very old. It describes a speed comparison between archaic versions of SQLite, MySQL and PostgreSQL.

The numbers here have become meaningless. This page has been retained only as an historical artifact.

Executive Summary

A series of tests were run to measure the relative performance of SQLite 2.7.6, PostgreSQL 7.1.3, and MySQL 3.23.41. The following are general conclusions drawn from these experiments:

- SQLite 2.7.6 is significantly faster (sometimes as much as 10 or 20 times faster) than the default PostgreSQL 7.1.3 installation on RedHat 7.2 for most common operations.
- SQLite 2.7.6 is often faster (sometimes more than twice as fast) than MySQL 3.23.41 for most common operations.
- SQLite does not execute CREATE INDEX or DROP TABLE as fast as the other databases. But this is not seen as a problem because those are infrequent operations.
- SQLite works best if you group multiple operations together into a single transaction.

The results presented here come with the following caveats:

- These tests did not attempt to measure multi-user performance or optimization of complex queries involving multiple joins and subqueries.
- These tests are on a relatively small (approximately 14 megabyte) database. They do not measure how well the database engines scale to larger problems.

Test Environment

The platform used for these tests is a 1.6GHz Athlon with 1GB or memory and an IDE disk drive. The operating system is RedHat Linux 7.2 with a stock kernel.

The PostgreSQL and MySQL servers used were as delivered by default on RedHat 7.2. (PostgreSQL version 7.1.3 and MySQL version 3.23.41.) No effort was made to tune these engines. Note in particular the default MySQL configuration on RedHat 7.2 does not support transactions. Not having to support transactions gives MySQL a big speed advantage, but SQLite is still able to hold its own on most tests.

I am told that the default PostgreSQL configuration in RedHat 7.3 is unnecessarily conservative (it is designed to work on a machine with 8MB of RAM) and that PostgreSQL could be made to run a lot faster with some knowledgeable configuration tuning. Matt Sergeant reports that he has tuned his PostgreSQL installation and rerun the tests shown below. His results show that PostgreSQL and MySQL run at about the same speed. For Matt's results, visit

> http://www.sergeant.org/sqlite_vs_pgsync.html

SQLite was tested in the same configuration that it appears on the website. It was compiled with -O6 optimization and with the -DNDEBUG=1 switch which disables the many "assert()" statements in the SQLite code. The -DNDEBUG=1 compiler option roughly doubles the speed of SQLite.

All tests are conducted on an otherwise quiescent machine. A simple Tcl script was used to generate and run all the tests. A copy of this Tcl script can be found in the SQLite source tree in the file **tools/speedtest.tcl**.

The times reported on all tests represent wall-clock time in seconds. Two separate time values are reported for SQLite. The first value is for SQLite in its default configuration with full disk synchronization turned on. With synchronization turned on, SQLite executes an **fsync()** system call (or the equivalent) at key points to make certain that critical data has actually been written to the disk drive surface. Synchronization is necessary to guarantee the integrity of the database if the operating system crashes or the computer powers down unexpectedly in the middle of a database update. The second time reported for SQLite is when synchronization is turned off. With synchronization off, SQLite is sometimes much faster, but there is a risk that an operating system crash or an unexpected power failure could damage the database. Generally speaking, the synchronous SQLite times are for comparison against PostgreSQL (which is also synchronous) and the asynchronous SQLite times are for comparison against the asynchronous MySQL engine.

Test 1: 1000 INSERTs

```
CREATE TABLE t1(a INTEGER, b INTEGER, c VARCHAR(100));
INSERT INTO t1 VALUES(1,13153,'thirteen thousand one hundred fifty three');
INSERT INTO t1 VALUES(2,75560,'seventy five thousand five hundred sixty');
_... 995 lines omitted_
INSERT INTO t1 VALUES(998,66289,'sixty six thousand two hundred eighty nine');
INSERT INTO t1 VALUES(999,24322,'twenty four thousand three hundred twenty two');
INSERT INTO t1 VALUES(1000,94142,'ninety four thousand one hundred forty two');
```

```
| PostgreSQL: |    4.373 |
| MySQL: |    0.114 |
| SQLite 2.7.6: |    13.061 |
| SQLite 2.7.6 (nosync): |    0.223 |
```

Because it does not have a central server to coordinate access, SQLite must close and reopen the database file, and thus invalidate its cache, for each transaction. In this test, each SQL statement is a separate transaction so the database file must be opened and closed and the cache must be flushed 1000 times. In spite of this, the asynchronous version of SQLite is still nearly as fast as MySQL. Notice how much slower the synchronous version is, however. SQLite calls **fsync()** after each synchronous transaction to make sure that all data is safely on the disk surface before continuing. For most of the 13 seconds in the synchronous test, SQLite was sitting idle waiting on disk I/O to complete.

Test 2: 25000 INSERTs in a transaction

```
BEGIN;
CREATE TABLE t2(a INTEGER, b INTEGER, c VARCHAR(100));
INSERT INTO t2 VALUES(1,59672,'fifty nine thousand six hundred seventy two');
_... 24997 lines omitted_
INSERT INTO t2 VALUES(24999,89569,'eighty nine thousand five hundred sixty nine');
INSERT INTO t2 VALUES(25000,94666,'ninety four thousand six hundred sixty six');
COMMIT;
```

```
| PostgreSQL: |    4.900 |
| MySQL: |    2.184 |
| SQLite 2.7.6: |    0.914 |
| SQLite 2.7.6 (nosync): |    0.757 |
```

When all the INSERTs are put in a transaction, SQLite no longer has to close and reopen the database or invalidate its cache between each statement. It also does not have to do any fsync()s until the very end. When unshackled in this way, SQLite is much faster than either PostgreSQL and MySQL.

Test 3: 25000 INSERTs into an indexed table

```
BEGIN;
CREATE TABLE t3(a INTEGER, b INTEGER, c VARCHAR(100));
CREATE INDEX i3 ON t3(c);
_... 24998 lines omitted_
INSERT INTO t3 VALUES(24999,88509,'eighty eight thousand five hundred nine');
INSERT INTO t3 VALUES(25000,84791,'eighty four thousand seven hundred ninety one');
COMMIT;
```

```
| PostgreSQL: |    8.175 |
| MySQL: |    3.197 |
| SQLite 2.7.6: |    1.555 |
| SQLite 2.7.6 (nosync): |    1.402 |
```

There were reports that SQLite did not perform as well on an indexed table. This test was recently added to disprove those rumors. It is true that SQLite is not as fast at creating new index entries as the other engines (see Test 6 below) but its overall speed is still better.

Test 4: 100 SELECTs without an index

```
BEGIN;
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=0 AND b&lt;1000;
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=100 AND b&lt;1100;
_... 96 lines omitted_
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=9800 AND b&lt;10800;
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=9900 AND b&lt;10900;
COMMIT;
```

```
| PostgreSQL: |    3.629 |
| MySQL: |    2.760 |
| SQLite 2.7.6: |    2.494 |
| SQLite 2.7.6 (nosync): |    2.526 |
```

This test does 100 queries on a 25000 entry table without an index, thus requiring a full table scan. Prior versions of SQLite used to be slower than PostgreSQL and MySQL on this test, but recent performance enhancements have increased its speed so that it is now the fastest of the group.

Test 5: 100 SELECTs on a string comparison

```
BEGIN;
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%one%';
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%two%';
_... 96 lines omitted_
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%ninety nine%';
SELECT count(*), avg(b) FROM t2 WHERE c LIKE '%one hundred%';
COMMIT;
```

```
| PostgreSQL: |    13.409 |
| MySQL: |    4.640 |
| SQLite 2.7.6: |    3.362 |
| SQLite 2.7.6 (nosync): |    3.372 |
```

This test still does 100 full table scans but it uses uses string comparisons instead of numerical comparisons. SQLite is over three times faster than PostgreSQL here and about 30% faster than MySQL.

Test 6: Creating an index

```
CREATE INDEX i2a ON t2(a);
CREATE INDEX i2b ON t2(b);
```

```
| PostgreSQL: |    0.381 |
| MySQL: |    0.318 |
| SQLite 2.7.6: |    0.777 |
| SQLite 2.7.6 (nosync): |    0.659 |
```

SQLite is slower at creating new indices. This is not a huge problem (since new indices are not created very often) but it is something that is being worked on. Hopefully, future versions of SQLite will do better here.

Test 7: 5000 SELECTs with an index

```
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=0 AND b&lt;100;
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=100 AND b&lt;200;
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=200 AND b&lt;300;
_... 4994 lines omitted_
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=499700 AND b&lt;499800;
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=499800 AND b&lt;499900;
SELECT count(*), avg(b) FROM t2 WHERE b&gt;=499900 AND b&lt;500000;
```

```
| PostgreSQL: |    4.614 |
| MySQL: |    1.270 |
| SQLite 2.7.6: |    1.121 |
| SQLite 2.7.6 (nosync): |    1.162 |
```

All three database engines run faster when they have indices to work with. But SQLite is still the fastest.

Test 8: 1000 UPDATEs without an index

```
BEGIN;
UPDATE t1 SET b=b*2 WHERE a&gt;=0 AND a&lt;10;
UPDATE t1 SET b=b*2 WHERE a&gt;=10 AND a&lt;20;
_... 996 lines omitted_
UPDATE t1 SET b=b*2 WHERE a&gt;=9980 AND a&lt;9990;
UPDATE t1 SET b=b*2 WHERE a&gt;=9990 AND a&lt;10000;
COMMIT;
```

```
| PostgreSQL: |    1.739 |
| MySQL: |    8.410 |
| SQLite 2.7.6: |    0.637 |
| SQLite 2.7.6 (nosync): |    0.638 |
```

For this particular UPDATE test, MySQL is consistently five or ten times slower than PostgreSQL and SQLite. I do not know why. MySQL is normally a very fast engine. Perhaps this problem has been addressed in later versions of MySQL.

Test 9: 25000 UPDATEs with an index

```
BEGIN;
UPDATE t2 SET b=468026 WHERE a=1;
UPDATE t2 SET b=121928 WHERE a=2;
_... 24996 lines omitted_
UPDATE t2 SET b=35065 WHERE a=24999;
UPDATE t2 SET b=347393 WHERE a=25000;
COMMIT;
```

```
| PostgreSQL: |    18.797 |
| MySQL: |    8.134 |
| SQLite 2.7.6: |    3.520 |
| SQLite 2.7.6 (nosync): |    3.104 |
```

As recently as version 2.7.0, SQLite ran at about the same speed as MySQL on this test. But recent optimizations to SQLite have more than doubled speed of UPDATEs.

Test 10: 25000 text UPDATEs with an index

```
BEGIN;
UPDATE t2 SET c='one hundred forty eight thousand three hundred eighty two' WHERE a=1;
UPDATE t2 SET c='three hundred sixty six thousand five hundred two' WHERE a=2;
_... 24996 lines omitted_
UPDATE t2 SET c='three hundred eighty three thousand ninety nine' WHERE a=24999;
UPDATE t2 SET c='two hundred fifty six thousand eight hundred thirty' WHERE a=25000;
COMMIT;
```

```
| PostgreSQL: |    48.133 |
| MySQL: |    6.982 |
| SQLite 2.7.6: |    2.408 |
| SQLite 2.7.6 (nosync): |    1.725 |
```

Here again, version 2.7.0 of SQLite used to run at about the same speed as MySQL. But now version 2.7.6 is over two times faster than MySQL and over twenty times faster than PostgreSQL.

In fairness to PostgreSQL, it started thrashing on this test. A knowledgeable administrator might be able to get PostgreSQL to run a lot faster here by tweaking and tuning the server a little.

Test 11: INSERTs from a SELECT

```
BEGIN;
INSERT INTO t1 SELECT b,a,c FROM t2;
INSERT INTO t2 SELECT b,a,c FROM t1;
COMMIT;
```

```
| PostgreSQL: |    61.364 |
| MySQL: |    1.537 |
| SQLite 2.7.6: |    2.787 |
| SQLite 2.7.6 (nosync): |    1.599 |
```

The asynchronous SQLite is just a shade slower than MySQL on this test. (MySQL seems to be especially adept at INSERT...SELECT statements.) The PostgreSQL engine is still thrashing - most of the 61 seconds it used were spent waiting on disk I/O.

Test 12: DELETE without an index

```
DELETE FROM t2 WHERE c LIKE '%fifty%';
```

```
| PostgreSQL: |    1.509 |
| MySQL: |    0.975 |
| SQLite 2.7.6: |    4.004 |
| SQLite 2.7.6 (nosync): |    0.560 |
```

The synchronous version of SQLite is the slowest of the group in this test, but the asynchronous version is the fastest. The difference is the extra time needed to execute fsync().

Test 13: DELETE with an index

```
DELETE FROM t2 WHERE a&gt;10 AND a&lt;20000; &lt;="" blockquote=""&gt;
```

```
| PostgreSQL: |    1.316 |
| MySQL: |    2.262 |
| SQLite 2.7.6: |    2.068 |
| SQLite 2.7.6 (nosync): |    0.752 |
```

This test is significant because it is one of the few where PostgreSQL is faster than MySQL. The asynchronous SQLite is, however, faster then both the other two.

Test 14: A big INSERT after a big DELETE

```
INSERT INTO t2 SELECT * FROM t1;
```

```
| PostgreSQL: |    13.168 |
| MySQL: |    1.815 |
| SQLite 2.7.6: |    3.210 |
| SQLite 2.7.6 (nosync): |    1.485 |
```

Some older versions of SQLite (prior to version 2.4.0) would show decreasing performance after a sequence of DELETEs followed by new INSERTs. As this test shows, the problem has now been resolved.

Test 15: A big DELETE followed by many small INSERTs

```
BEGIN;
DELETE FROM t1;
INSERT INTO t1 VALUES(1,10719,'ten thousand seven hundred nineteen');
_... 11997 lines omitted_
INSERT INTO t1 VALUES(11999,72836,'seventy two thousand eight hundred thirty six');
INSERT INTO t1 VALUES(12000,64231,'sixty four thousand two hundred thirty one');
COMMIT;
```

```
| PostgreSQL: |    4.556 |
| MySQL: |    1.704 |
| SQLite 2.7.6: |    0.618 |
| SQLite 2.7.6 (nosync): |    0.406 |
```

SQLite is very good at doing INSERTs within a transaction, which probably explains why it is so much faster than the other databases at this test.

Test 16: DROP TABLE

```
DROP TABLE t1;
DROP TABLE t2;
DROP TABLE t3;
```

```
| PostgreSQL: |    0.135 |
| MySQL: |    0.015 |
| SQLite 2.7.6: |    0.939 |
| SQLite 2.7.6 (nosync): |    0.254 |
```

SQLite is slower than the other databases when it comes to dropping tables. This probably is because when SQLite drops a table, it has to go through and erase the records in the database file that deal with that table. MySQL and PostgreSQL, on the other hand, use separate files to represent each table so they can drop a table simply by deleting a file, which is much faster.

On the other hand, dropping tables is not a very common operation so if SQLite takes a little longer, that is not seen as a big problem.