Database File Format

► Table Of Contents

This document describes and defines the on-disk database file format used by all releases of SQLite since version 3.0.0 (2004-06-18).

1. The Database File

The complete state of an SQLite database is usually contained in 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.

1.1. Hot Journals

If the application or host computer crashes before the transaction completes, then the rollback journal or write-ahead log contains 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.

1.2. 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^{31} - 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 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, though terabyte-size SQLite databases are known to exist in production.

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.

1.3. 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

1.3.1. 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.

1.3.2. Page Size

The two-byte value beginning at offset 16 determines the page size of the database. For SQLite versions 3.7.0.1 (2010-08-04) 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 (2010-08-23), 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 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.

1.3.3. 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.

1.3.4. 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 specified 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.

1.3.5. 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.

1.3.6. 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.

1.3.7. 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 (2010-07-21) 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.

1.3.8. 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.

1.3.9. 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.

1.3.10. 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 (2004-06-18).
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 2005-02-20.

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.

1.3.11. 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.

1.3.12. 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.

1.3.13. 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.

1.3.14. 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.

1.3.15. 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.

1.3.16. 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".

1.3.17. Header space reserved for expansion

All other bytes of the database file header are reserved for future expansion and must be set to zero.

1.4. 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.

The lock-byte page arose from the need to support Win95 which was the predominant operating system when this file format was designed and which only supported mandatory file locking. All modern operating systems that we know of support advisory file locking, and so the lock-byte page is not really needed any more, but is retained for backwards compatibility.

1.5. 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 page 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 (2008-07-16) 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.

1.6. 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 (0x02) means the page is an interior index b-tree page. • A value of 5 (0x05) means the page is an interior table b-tree page. • A value of 10 (0x0a) means the page is a leaf index b-tree page. • A value of 13 (0x0d) 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 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. The 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. The 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.
- If P>X and K<=X then the first K bytes of P are stored on the btree page and the remaining P-K bytes are stored on overflow pages.
- 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.

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 designer of the SQLite b-tree logic realized 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.

1.7. 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.

1.8. 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. 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	*variable*	*Reserved for internal use. These serial type codes will never appear in a well-formed database file, but they might be used in transient and temporary database files that SQLite sometimes generates for its own use. The meanings of these codes can shift from one release of SQLite to the next.*
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.

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. 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.

A record might have fewer values than the number of columns in the corresponding table. This can happen, for example, after an ALTER TABLE ... ADD COLUMN SQL statement has increased the number of columns in the table schema without modifying preexisting rows in the table. Missing values at the end of the record are filled in using the default value for the corresponding columns defined in the table schema.

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. 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.

2.4.1. Suppression of redundant columns in the PRIMARY KEY of WITHOUT ROWID tables

If the PRIMARY KEY of a WITHOUT ROWID tables uses the same columns with the same collating sequence more than once, then the second and subsequent occurrences of that column in the PRIMARY KEY definition are ignored. For example, the following CREATE TABLE statements all specify the same table, which will have the exact same representation on disk:

```
CREATE TABLE t1(a,b,c,d,PRIMARY KEY(a,c)) WITHOUT ROWID);
CREATE TABLE t1(a,b,c,d,PRIMARY KEY(a,c,a,c)) WITHOUT ROWID);
CREATE TABLE t1(a,b,c,d,PRIMARY KEY(a,A,a,C)) WITHOUT ROWID);
CREATE TABLE t1(a,b,c,d,PRIMARY KEY(a,a,a,a,c)) WITHOUT ROWID);
```

The first example above is the preferred definition of the table, of course. All of the examples create a WITHOUT ROWID table with two PRIMARY KEY columns, "a" and "c", in that order, followed by two data columns "b" and "d", also in that order.

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 indexes

In an index on a WITHOUT ROWID table, if a column of the PRIMARY KEY is also a column in the index and has a matching collating sequence, 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);
CREATE INDEX ex25ae ON ex25(a COLLATE nocase,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.

Each row in ex25ae contains five columns: a, e, d, c, a. The "a" column is repeated since the first occurrence of "a" has a collating function of "nocase" and the second has a collating sequence of "binary". If the "a" column is not repeated and if the table contains two or more entries with the same "e" value and where "a" differs only in case, then all of those table entries would correspond to a single entry in the index, which would break the one-to-one correspondence between the table and the index.

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 inserted into 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 inserted 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. If the sqlite_stat1.idx column is the same as the sqlite_stat1.tbl column, then the table is a WITHOUT ROWID table and the sqlite_stat1.stat field contains information about the index btree that implements the WITHOUT ROWID table.

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 (2009-09-11) and 3.7.8 (2011-09-19). 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*C*2 + 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, if 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 (2011-11-01) 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 (2013-10-17) 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 (2013-10-17) 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. The Rollback Journal

The rollback journal is a file associated with each SQLite database file that holds 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 byte at offset X into the page as an 8-bit unsigned integer and add the value of that integer to the checksum.
4. Subtract 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. The Write-Ahead Log

Beginning with version 3.7.0 (2010-07-21), 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.

A checkpoint need not run to completion. It might be that some readers are still using older transactions with data that is contained in the database file. In that case, transferring content for newer transactions from the WAL file into the database would delete the content out from under readers still using the older transactions. To avoid that, checkpoints only run to completion if all reader are using the last transaction in the WAL.

4.4. WAL Reset

After a complete checkpoint, if no other connections are in transactions that use the WAL, then subsequent write transactions can overwrite the WAL file from the beginning. This is called "resetting the WAL". 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.

The WAL file can optionally be truncated on a reset, but it need not be. Performance is usually a little better if the WAL is not truncated, since filesystems generally will overwrite an existing file faster than they will grow a file.

4.5. 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 number of value frames in the WAL as "mxFrame". (More detail) 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.

4.6. WAL-Index Format

Conceptually, the wal-index is shared memory, though the current VFS implementations use a memory-mapped file for operating-system portability. The memory-mapped 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, as all clients 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. Additional details on the format of the wal-index are contained in the separate WAL-index File Format document.