SQLite4

LSM Design Overview

Not logged in

2019-05-29 09:34

HomeFilesTimelineBranchesTagsWikiLogin

1. Summary

The LSM embedded database software stores data in three distinct data structures:

- The **shared-memory region**. This may actually be allocated in either shared or heap memory, depending on whether LSM is running in multi (the default) or single process mode. Either way, the shared-memory region contains volatile data that is shared at run-time between database clients. Similar to the *-shm file used by SQLite in WAL mode.

  As well as various fixed-size fields, the shared-memory region contains the 'in-memory tree'. The in-memory tree is an append-only red-black tree structure used to stage user data that has not yet flushed into the database file by the system. Under normal circumstances, the in-memory tree is not allowed to grow very large.

- The **log file**. A circular log file that provides a persistent backup of the contents of the in-memory tree and any other data that has not yet been synced to disk. The log-file is not used for rollback (like an SQLite journal file) or to store data that is retrieved at runtime by database clients (like an SQLite WAL file). Its only purpose is to provide robustness.

- The **database file**. A database file consists of an 8KB header and a body. The body contains zero or more "sorted runs" - arrays of key-value pairs sorted by key.

To query a database, the contents of the in-memory tree must be merged with the contents of each sorted run in the database file. Entries from newer sorted runs are favoured over old (for the purposes of merging, the in-memory tree contains the newest data).

When an application writes to the database, the new data is written to the in-memory tree. Once the in-memory tree has grown large enough, its contents are written into the database file as a new sorted run. To reduce the number of sorted runs in the database file, chronologically adjacent sorted runs may be merged together into a single run, either automatically or on demand.

2. Locks

Read/write (shared/exclusive) file locks are used to control concurrent access. LSM uses the following "locking regions". Each locking region may be locked and unlocked separately.

DMS1

This locking region is used to serialize all connection and disconnection operations performed by read-write database connections. An EXCLUSIVE lock is taken for the duration of all such operations.

Additionally, read-only connections take a SHARED lock on this locking region while attempting to connect to a database. This ensures that a read-only connection does not attempt to connect to the database while a read-write clients connection or disconnection operation is ongoing.

DMS2

Read-write connections hold a SHARED lock on this locking region for as long as they are connected to the database.

DMS3

Read-only connections hold a SHARED lock on this locking region for as long as they are connected to the database.

RWCLIENT(n)

There are a total of 16 RWCLIENT locking regions. After a read-write client connects to the database it attempts to find a free RWCLIENT locking slot to take an EXCLUSIVE lock on. If it cannot find one, this is not an error. If it can, then the lock is held for as long as the read-write client is connected to the database for.

The sole purpose of these locks is that they allow a read-only client to detect whether or not there exists at least one read-write client connected to the database. Of course if large numbers of read-write clients connect and disconnect from the system in an inconvenient order the system may enter a state where there exists one or more connected read-write clients but none of them hold a RWCLIENT lock. This is not important - if a read-only client fails to detect that the system has read-write clients it may be less efficient, but will not malfunction.

WRITER A database client holds an EXCLUSIVE lock on this locking region while writing data to the database. Outside of recovery, only clients holding this lock may modify the contents of the in-memory b-tree. Holding this lock is synonymous with having an open write transaction on the database.

WORKER A database client holds an EXCLUSIVE lock on this locking region while performing database work (writing data into the body of the database file).

CHECKPOINTER A database client holds an EXCLUSIVE lock on this locking region while performing a checkpoint (syncing the database file and writing to the database header).

ROTRANS A read-only database client holds a SHARED lock on this locking region while reading from a non-live database system.

READER(n) There are a total of 6 READER locking regions. Unless it is a read-only client reading from a non-live database, a client holds a SHARED lock on one of these while it has an open read transaction. Each READER lock is associated with a pair of id values identifying the regions of the in-memory tree and database file that may be read by clients holding such SHARED locks.

3. Database Connect and Disconnect Operations

3.1. Read-write clients

When an LSM database connection is opened (i.e. lsm_open() is called):

```
lock(DMS1, EXCLUSIVE)                  # Block until successful
  trylock(DMS2+DMS3, EXCLUSIVE)
  if( trylock() successful ){
    zero shared memory and run recovery
  }
  if( no error during recovery ){
    lock(DMS2, SHARED)                # "cannot" fail
    lock(RWCLIENT(x), EXCLUSIVE)      # see comment below
  }
unlock(DMS1)
```

Running recovery involves reading the database file header and log file to initialize the contents of shared-memory. Recovery is only run when the first connection connects to the database file. There are no circumstances (including the unexpected failure of a writer process) that may cause recovery to take place after the first client has successfully connected.

After the SHARED lock on DMS2 is taken, an effort is made to find a free RWCLIENT locking region and take an EXCLUSIVE lock on it. If no such region can be found, this step is omitted. It is not an error if this happens.

Assuming recovery is successful (or not required), the database client is left holding a SHARED lock on DMS2 and, possibly, an EXCLUSIVE lock on one of the RWCLIENT locks. These locks are held for as long as the database connection remains open.

When disconnecting from a database (i.e. an lsm_close() call following a successful lsm_open()):

```
  lock(DMS1, EXCLUSIVE)           # Block until successful
    trylock(DMS2, EXCLUSIVE)
    if( trylock() successful ){
      flush in-memory tree
      checkpoint database

      trylock(DMS3, EXCLUSIVE)
      if( trylock() successful ){
        delete shared memory
      }

      trylock(ROTRANS, EXCLUSIVE)
      if( trylock() successful ){
        unlink log file
      }
    }
    unlock(RWCLIENT(x))           # If holding RWCLIENT lock
    unlock(DMS2)
  unlock(DMS1)
```

3.2. Read-only clients

It is assumed that read-only clients:

- may take SHARED locks only,
- may not write to shared-memory, the database file or the log file, and
- may not use the trylock(x, EXCLUSIVE) primitive to detect SHARED locks held by other clients.

A read-only client does not attempt to connect to the database from within the lsm_open() call. Instead, this is defered until the first time the client attempts to read from the database.

```
  lock(DMS1, SHARED)              # Block until successful
    trylock(RWCLIENT(all), SHARED)
    if( trylock() successful ){
      # System is not live. The database must be read directly from disk.
      lock(ROTRANS, SHARED)
    }else{
      # Client is now connected. The read transaction may now be opened
      # in the same way as it would be for a read-write client.
      lock(DMS3, SHARED)
    }
  unlock(DMS1)
```

Assuming no error occurs, the procedure above leads to one of two possible outcomes:

- DMS3 is locked and the read-only client is now connected to the database. From this point on, the read-only client uses the same procedure to open a read transaction as any other client. The lock on DMS3 is held until the client disconnects from the database.

- ROTRANS is locked and the read-only client is still disconnected. Holding the lock on ROTRANS guarantees that no read-write client will overwrite any existing data in the database file or log file. This allows the read-only client to run a disconnected read transaction - it recovers any data in the log file into private memory, then reads data as required from the database file for the duration of the users read transaction.

A disconnected read transaction is closed by dropping the ROTRANS lock.

4. Data Structures

In the following sections, "the WRITER lock", refers to an exclusive lock on the WRITER locking region. For example "holding the WRITER lock" is equivalent to "holding an exclusive lock on the WRITER locking region". Similar interpretations apply to "the WORKER lock" and "the CHECKPOINTER lock".

4.1. Database file

This section summarizes the contents of the database file informally. A detailed description is found in the header comments for source code files lsm_file.c (blocks, pages etc.), lsm_sorted.c (sorted run format) and lsm_ckpt.c (database snapshot format).

The first 8KB of an LSM database file contains the database header. The database header is in turn divided into two 4KB "meta-pages". It is assumed that either of these two pages may be written without risking damage to the other in the event of a power failure. TODO: This assumption must be a problem. Do something about it.

The entire database file is divided into "blocks". The block-size is configurable (default 1MB). The first block includes the header, so contains less usable space than each subsequent block. Each block is sub-divided into configurably sized (default 4KB) pages. When reading from the database, clients read data into memory a page at a time (as they do when accessing SQLite's b-tree implementation). When writing to the database, the system attempts to write contiguously to an entire block before moving the disk head and writing elsewhere in the file.

As with an SQLite database file, each page in the database may be addressed by its 32-bit page number. This means the maximum database size is roughly (pgsz * 2^32) bytes. The first and last pages in each block are 4 bytes smaller than the others. This is to make room for a single page-number.

4.1.1. Sorted Runs

A single sorted run is spread across one or more database pages (each page is a part of at most one sorted run). Given the page number of a page in a sorted run the following statements are true:

- If the page is not the last page in the sorted run or the last page in a database file block, the next page in the same sorted run is the next page in the file. Or, if the page is the last page in a database file block, the next page is identified by the 4 byte page number stored as part of the last page on each block.
- If the page is not the first page in the sorted run or the first page in a database file block, the previous page in the same sorted run is the previous page in the file. Or, if the page is the first page in a database file block, the previous page is identified by the 4 byte page number stored as part of the first page on each block.

As well as pages containing user data, each sorted run contains "b-tree pages". These pages are similar to the internal nodes of b+-tree structures. They make it possible to treat the sorted-run as a b+tree when querying for a specific value or bounded range.

In other words, given the page numbers of the first and last pages of a sorted run and the page number of the root page for the embedded b-tree, it is possible to traverse the entire run in either direction or query for arbitrary values.

TODO: Embedded pointers.

4.1.2. Levels

Each sorted run is assigned to a "level". Normally, a level consists of a single sorted run. However, a level may also consist of a set of sorted runs being incrementally merged into a single run.

Figure 1 depicts a database that consists of 4 levels (L0, L1, L2 and L3). L3 contains the data most recently written to the database file. Each rectangular box in the diagram represents a single sorted run. Arrows represent embedded pointers. For example, the sorted run in L0 contains embedded pointers to the sorted run in L1.

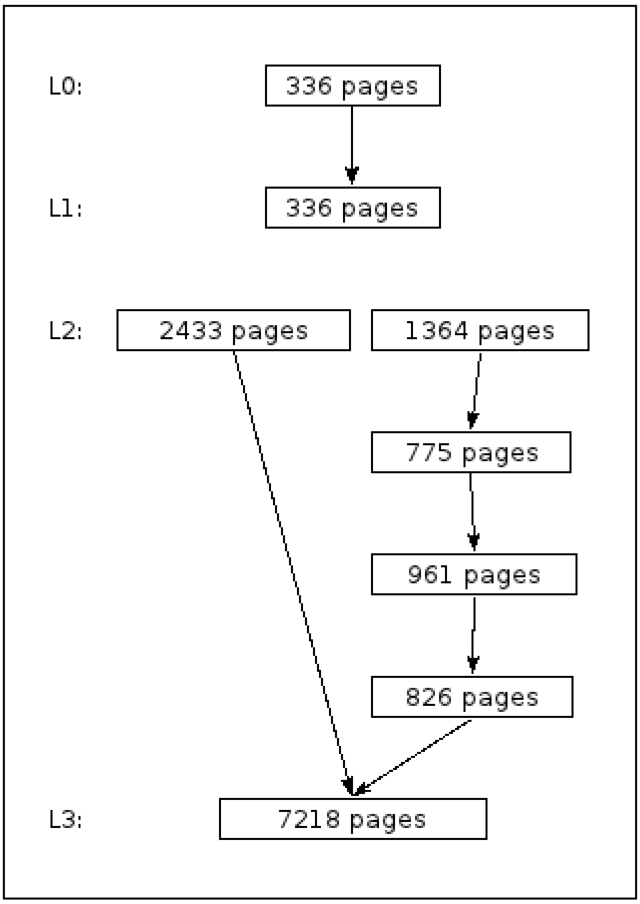

Figure 1.

Level L2 is an example of a level that consists of four sorted runs being merged into a single run. Once all the data in the runs to the right of the figure has been merged into the new run on the left hand side, the four original runs will be no longer required and the space they consume in the database file recycled. In the meantime, the system knows the value of the key most recently written to the left hand side. This key is known as the levels "split-key". When querying L2, if the key being queried for is less than or equal to the split-key, the left hand side of the level is searched. Otherwise, if the key being sought is larger than the split-key, the right hand side is searched.

Querying the database in figure 1 for a single key (K) therefore proceeds as follows:

1. A b+-tree search for the key is run on the sorted run in L0.

2. Use the embedded pointer (page number) located by the search of L0 to go directly to the page in the sorted run in L1 that may contain the key.

3. Since L2 is a split-level, compare K to the split-key. If K is less than or equal to the split-key, perform a b+-tree search of the sorted run on the left hand side of L2.

   Otherwise, perform a b+-tree search of the 1364 page run on the right hand side of L2, then use the embedded pointers to search each subsequent run on the right hand side without accessing any b+-tree pages.

4. Regardless of whether the left or right side of L2 was searched, the search also locates a pointer to an L3 page. This pointer is used to load the page of the L3 sorted run that may contain K.

If, at any point in the search an instance of K is located, the remaining steps can be omitted and the results returned immediately. This means querying for recently written data may be significantly faster than the average query time for all entries.

4.1.3. Snapshots

Each meta page may contain a database **snapshot**. A snapshot contains all the information required to interpret the remainder of the database file (the sorted runs and free space). Specifically, it contains:

- A 64-bit checksum of the snapshot contents.
- A 64-bit snapshot id.
- The total number of blocks in the database file.
- The block size in bytes.
- The page size in bytes.
- A description of each of the levels in the database and the sorted runs they are made up of.
- A subset of the free-block list (see below).
- The log file offset (and initial checksum values) at which to begin log file recovery if this snapshot is loaded from disk (see "Database Recovery and Shutdown" below).

A more detailed description is available in the header comments in source code file lsm_ckpt.c

4.2. In-Memory Tree

The in-memory tree is an append-only b-tree of order 4 (each node may have up to 4 children), which is more or less equivalent to a red-black tree. An append-only tree is convenient, as it naturally supports the single-writer/many-readers MVCC concurrency model.

The implementation includes some optimizations to reduce the number of interior nodes that are updated when a leaf node is written that are not described here. See header comments in source code file lsm_tree.c for details.

4.2.1. Memory Allocation

More than one in-memory tree may exist in shared-memory at any time. For example in the following scenario:

1. Reader process opens and starts reading in-memory tree.
2. Writer process flushes the contents of the current in-memory tree to disk.
3. Writer process begins inserting data into a second in-memory tree, while the reader process is still reading from the first.

In this case, the memory that was used for the first in-memory tree may not be reused until after the reader process has closed its read-transaction.

Within the shared memory region, space for the in-memory tree is allocated in 32KB chunks. Each time a new chunk is allocated from the end of the file, or an existing chunk reused, it is assigned a 32-bit monotonically increasing sequence value. This value is written to the first 4 bytes of the chunk. Additionally, all chunks are linked into a linked list in ascending order of sequence number using a 32-bit pointer value stored in the second 4-bytes of each chunk. The pointer to the first chunk in the list is stored in the tree-header (see below).

Rolling sequence ids

It is conceivable that the 32-bit sequence ids assigned to shared memory chunks may overflow. However the approach described in this section assume that increasing sequence ids may be allocated indefinitely. This is reconciled using the following argument:

Because of the way chunks are reused, it is not possible for chunk (I+1) to be recycled while there chunk I exists. The set of chunks that are present in shared memory always have a contiguous set of sequence ids. Therefore, assuming that there are not 2^30 or more chunks in shared-memory, it is possible to compare any two 32-bit unsigned sequence ids using the following:

```
/* Return true if "a" is larger than or equal to "b" */
#define sequence_gt(a, b) (((u32)a-(u32)b) < (1<<30))
```

When a new 32KB chunk is required as part of inserting data into an append-only b-tree, LSM either reuses the first chunk in the linked list (if it is not part of the current in-memory tree or part of one that is still being used by a client), or else allocates a new chunk by extending the size of the shared-memory region.

If an application failure occurs while writing to the in-memory tree, the next writer detects that this has happened (see below). In this event, the sequence ids attached to shared-memory chunks are considered trustworthy, but the values that connect the linked list together are not. The writer that detects the failure must scan the entire shared-memory region to reconstruct the linked list. Any sequence ids assigned by the failed writer are reverted (perhaps not to their original values, but to values that put them at the start of the linked list - before those chunks that may still be in use by existing readers).

4.2.2. Header Fields

As well as the in-memory tree data, the following fixed-size fields stored in well-known locations in shared-memory are part of the in-memory tree. Like the in-memory tree data, outside of recovery these fields are only ever written to by clients holding the WRITER lock.

- Two copies of a data structure called a "tree-header". Tree-header-1 and tree-header 2. A tree-header structure contains all the information required to read or write to a particular version of the append only b-tree. It also contains a 64-bit checksum.
- A boolean flag set to true at the beginning of every write transaction and cleared after that transaction is successfully concluded - the "writer flag". This is used to detect failures that occur mid-transaction. It is only ever read (or written) by clients that hold the WRITER lock.

4.3. Other Shared-Memory Fields

- Snapshot 1.
- Snapshot 2.
- The meta-page pointer. This value is either 1 or 2. It indicates which of the two meta-pages contains the most recent database snapshot.
- READER lock values.

4.4. Log file

lsm_log.c.

5. Database Operations

5.1. Reading

Opening a read transaction:

1. Load the current tree-header from shared-memory.

2. Load the current snapshot from shared-memory.

Steps 1 and 2 are similar. In both cases, there are two copies of the data structure being read in shared memory. No lock is held to prevent another client updating them while the read is taking place. Updaters use the following pattern:

i. Update copy 2.
ii. Invoke xShmBarrier().
iii. Update copy 1.

Because writers always use the pattern above, either copy 1 or copy 2 of the data structure being read should be valid at all times (i.e. not half-way through being updated). A valid read can be identified by checking that the 64-bit checksum embedded in both data structures matches the rest of the content. So steps 1 and 2 proceed as follows:

i. Attempt to read copy 1 of the data structure.
ii. If step 1 is unsuccessful, attempt to read copy 2.
iii. If neither read attempt succeeded, invoke xShmBarrier() and re-attempt the operation.
iv. If the data structure cannot be read after a number of attempts, return the equivalent of SQLITE_PROTOCOL.

In the above, "attempting to read" a data structure means taking a private copy of the data and then attempting to verify its checksum. If the checksum matches the content of the tree-header or snapshot, then the attempted read is successful.

This is slightly different from the way SQLite WAL mode does things. SQLite also verifies that copy 1 and copy 2 of the data structure are identical. LSM could add an option to do that. However, to deal with the case where a writer/worker fails while updating one copy of the data structure, a reader would have to take the WRITER or WORKER lock as appropriate and "repair" the data structures so that both copies were consistent. See below under "Writing" and "Working" for details. This would mean we were only depending on the 64-bit checksums for recovery following an unluckily timed application failure, not for reader/writer collision detection.

3. Take a READER lock. The snapshot-id field of the reader-lock field must be less than or equal to the snapshot-id of the snapshot read in step 2. The shm-sequence-id of the reader lock must be less than or equal to the sequence-id of the first shared-memory chunk used by the in-memory tree according to the tree-header read in step 1.

4. Check that the tree-header read in step 1 is still current.

5. Check that the snapshot read in step 2 is still current.

   Steps 5 and 6 are also similar. If the check in step 5 fails, there is a chance that the shared-memory chunks used to store the in-memory tree indicated by the tree-header read in step 1 had already been reused by a writer client before the lock in step 4 was obtained. And similarly if the check in step 6 fails, the database file blocks occupied by the sorted runs in the snapshot read in step 2 may have already been reused by new content. In either case, if the check fails, return to step 1 and start over.

> Say a reader loads the tree-header for in-memory tree A, version 49 in step 1. Then, a writer writes a transaction to the in-memory tree A (creating version 50) and flushes the whole tree to disk, updating the current snapshot in shared-memory. All before the reader gets to step 2 above and loads the current snapshot. If allowed to proceed, the reader may return bad data or otherwise malfunction.
>
> Step 4 should prevent this. If the above occurs, step 4 should detect that the tree-header loaded in step 1 is no longer current.

Once a read transaction is opened, the reader may continue to read the versions of the in-memory tree and database file for as long as the READER lock is held.

To close a read transaction all that is required is to drop the SHARED lock held on the READER slot.

5.2. Writing

To open a write transaction:

1. Open a read transaction, if one is not already open.
2. Obtain the WRITER lock.
3. Check if the "writer flag" is set. If so, repair the in-memory tree (see below).
4. Check that the tree-header loaded when the read transaction was opened matches the one currently stored in shared-memory to ensure this is not an attempt to write from an old snapshot.
5. Set the writer flag.

To commit a write transaction:

1. Update copy 2 of the tree-header.
2. Invoke xShmBarrier().
3. Update copy 1 of the tree-header.
4. Clear the transaction in progress flag.
5. Drop the WRITER lock.

Starting a new in-memory tree is similar executing a write transaction, except that as well as initializing the tree-header to reflect an empty tree, the snapshot id stored in the tree-header must be updated to correspond to the snapshot created by flushing the contents of the previous tree.

If the writer flag is set after the WRITER lock is obtained, the new writer assumes that the previous must have failed mid-transaction. In this case it performs the following:

1. If the two tree-headers are not identical, copy one over the other. Prefer the data from a tree-header for which the checksum computes. Or, if they both compute, prefer tree-header-1.
2. Sweep the shared-memory area to rebuild the linked list of chunks so that it is consistent with the current tree-header.
3. Clear the writer flag.

5.3. Working

Working is similar to writing. The difference is that a "writer" modifies the in-memory tree. A "worker" modifies the contents of the database file.

1. Take the WORKER lock.

2. Check that the two snapshots in shared memory are identical. If not, and snapshot-1 has a valid checksum, copy snapshot-1 over the top of snapshot-2. Otherwise, copy snapshot-2 over the top of snapshot-1.

3. Modify the contents of the database file.

4. Update snapshot-2 in shared-memory.

5. Invoke xShmBarrier().

6. Update snapshot-1 in shared-memory.

7. Release the WORKER lock.

5.3.1. Free-block list management

Worker clients occasionally need to allocate new database blocks or move existing blocks to the free-block list. Along with the block number of each free block, the free-block list contains the snapshot-id of the first snapshot created after the block was moved to the free list. The free-block list is always stored in order of snapshot-id, so that the first block in the free list is the one that has been free for the longest.

There are two ways to allocate a new block - by extending the database file or by reusing a currently unused block from the head of the free-block list. There are two conditions for reusing a free-block:

- All existing database readers must be reading from snapshots with ids greater than or equal to the id stored in the free list. A worker process can check this by scanning the values associated with the READER locks - similar to the way SQLite does in WAL mode.
- The snapshot identified by the free-block list entry, or one with a more recent snapshot-id, must have been copied into the database file header. This is done by reading (and verifying the checksum) of the snapshot currently stored in the database meta-page indicated by the shared-memory variable.

5.4. Checkpoint Operations

1. Take CHECKPOINTER lock.
2. Load snapshot-1 from shared-memory. If the checksum does not match the content here, release the CHECKPOINTER lock and abandon the attempt to checkpoint the database.
3. The shared-memory region contains a variable indicating the database meta-page that a snapshot was last read from or written to. Check if this page contains the same snapshot as just read from shared-memory.
4. Sync the database file.
5. Write the snapshot into a meta-page other than that (if any) currently identified by the shared-memory variable.
6. Sync the database file again.
7. Update the shared-memory variable to indicate the meta-page written in step 5.
8. Drop the CHECKPOINTER lock.

This page was generated in about 0.004s by Fossil 2.9 [ac199e7a8a] 2019-05-28 12:16:00