Write Protocol

LMDB uses a single data file for databases. In a LMDB database file, there are two B-tree structures - one containing the actual data and another that contains the IDs of pages that are no longer in use.

A metapage that is found in the beginning of the database file points to the roots of the above two tree structures. There are two such meta pages and transactions alternate between them for header-data updates. The header-data is usually 106 bytes in length. LMDB has only transactional API to interact with the database.

When a new key value pair is inserted, a new data page (or a free one) is used and written at the end of the file. After this, fdatasync() is done on the data file. Corresponding changes to the structure of the B-tree are made in memory and then persisted. After sync-ing the data, the header-data is written and sync-ed.

NOTE : There are 2 file descriptors for the data file - FD1 and FD2. FD2 is a synchronous FD - i.e., it has O_DSYNC flag set in it. So any write operation that uses FD2 will be synchronous. The data region of the file is written using FD1 whereas the metapage updates are done through FD2.

If there is a crash between writing the data and the header-data, metapage would point to the older version of the database and hence be consistent. If the header-data write happened properly, the database state would be updated and it would be in the new state.

Environment is an abstraction of a directory inside which multiple databases can reside.

Assumptions and Vulnerabilities

1. Safe new file flush assumption : LMDB assumes safe new file flush property. LMDB creates 2 files (data.mdb and lock.mdb) when an environment is opened but does not do a sync on the parent directory to ensure that directory entries for the created files are persisted. This is just needed for the first time when the environment is created as all subsequently created DBs and key values go into the same file. In a file system where this property does not hold good, subsequent environment open calls will not show the previously created databases.

2. Process/System crash when creating an environment : As mentioned earlier, LMDB writes two metapages to the beginning of the data file. There is no sync call made after writing these 2 pages. A crash can happen before these pages are persisted to disk. If this crash happens, any subsequent open calls on the same environment will fail with an error saying that the data file is not a valid MDB file.

3. *KEY ASSUMPTION* : Header-data write (106 bytes) is atomic. We experimented breaking this assumption. The header-data is always written using FD2(the one with O_DSYNC flag). But there is a chance that this write can be split and a crash can happen in middle. There are actually two important parts in the header-data written - root of the data B-tree and the root of the free pages list. If a crash happened in such a way that the free page pointer doesn't reach the disk but the root of the data B-tree reaches the disk, then subsequent put operations will fail. We see the following error in this case: ***Assertion 'mp->mp_p.p_pgno != pgno' failed.*** There is an invariant always expected by mdb which is ***free_pages + current_pages + 2 metapages = in_use_pages - 1***. The above crash violates this expectation and hence the problem occurs.

Note: If the free pages section reaches the disk but not the root of the data B-tree, there is no problem.

Workloads and Checkers

The workload suite creation was a two step (manual) process:

1. A very simple workload was run and the strace was observed.
2. From the observation, more workloads(which can make the strace more complex) were added.

Following workloads were created and run.

1. Simple put-get tests with single and multi transactions.
2. Nested transactions.
3. Simple value replace.
4. Simple key delete.
5. Opening of environment.
6. Opening of named DB.

The checkers were simple - Verify if the transaction changes happened atomically by retrieving all the key values.

Additional features and configurations

As mentioned earlier, the only way to interact with the data store is through a transactional interface. Usually normal transactional operations are synchronous. i.e., Whenever a transaction is commited, all the data and header-data are synchronized to the disk. But there are few configuration options that can change this behavior.

These configuration options apply at the environment level only and not for a particular transaction:

'metasync' : If False, never explicitly flushes header-data pages to disk. OS will flush at its discretion, or user can flush with sync(). 'sync': If False, never explicitly flush data pages to disk. OS will flush at its discretion, or user can flush with sync(). This optimization means a system crash can corrupt the database or lose the last transactions if buffers are not yet flushed to disk. 'writemap': If True LMDB will use a writeable memory map to update the database. This option is incompatible with nested transactions. 'map_async': When writemap=True, use asynchronous flushes to disk. As with sync=False, a system crash can then corrupt the database or lose the last transactions. Calling sync() ensures on-disk database integrity until next commit.

Default configuration: writemap = False, metasync= True, map_async doesn't matter.

Note that there is a clear mention about the integrity of the DB if some of the options are turned off or on. Also note that there is no proper mention about what happens to integrity when only metasync is set to false in the documentation. These above configurations change the strace collected for the workloads in a significant way.

Perfomance numbers for various sync configurations

Workload details: Insert 500 key value pairs with 500 txns.

Configurations and average elapsed time in seconds:

Default configuration: writemap = False, metasync= True, map_async doesn't matter.

1. Default: 11.0824041367
2. Metasync = False: 10.2657073498
3. Metasync = True, writemap = True, map_async = False: Average: 29.8406929016
4. Metasync = False, writemap = True, map_async = Average: 0.0028694152832

5. Metasync = True, writemap = True, map_async = True: Average: 0.0029531955719
6. Sync = False: Average: 0.00672941207886

More details on vulnerability 3

header-data total size : 106 bytes.

This is the memory layout of the header-data:

- First 42 bytes for free list related data.
- Next 48 bytes for database related data (pad-4,flags-2,depth-2,branch_page_count-8,leaf_count-8,overflow_page_count-8,entries-8,root_of_tree-8)
- Last_page_in_the_tree - 8 bytes.
- Transaction ID - 8 bytes.

Note: 42+48+8+8 = 106.

Initial setup: There were 3 previous transactions on the database. The workload is the 4th txn on the database.

I tried the following crash scenarios with respect to header-data:

1. Only txn_id gets to disk
2. Free list data, txn_id and last_page gets to disk but not main db details
3. Only the last_page field goes to disk
4. only main db - 2 fields go to disk and free pointer is missed
5. only free pointer goes to disk
6. Everything gets in except complete database pointer - it is 48 bytes - but a crash happens such that second 24 bytes (which contains the root) is missed
7. Everything gets in except complete database pointer - it is 48 bytes - but a crash happens such that first 24 bytes (which contains depth, br_pages, leaves count etc) is missed

After crashing, the query was made for all key value pairs and a new txn to insert a new key value pair was tried.

Results:

1. Only txn_id gets into metapage and no other data:

Values obtained state: txn2 Result : No issues.

2. Free list data, txn_id and last_page goes in but not main db details:

Values obtained state: txn2 Result : Error: lib/mdb.c:1921: mdb_page_touch: Assertion ‘mp->mp_p.p_pgno != pgno’ failed.

3. Only the last_page field goes in nothing else:

Values obtained state: txn3 Result : No issues.

4. only main db - 2 fields no free pointer:

Values obtained state: txn4 Result : Error: lib/mdb.c:1921: mdb_page_touch: Assertion ‘mp->mp_p.p_pgno != pgno’ failed.

5. only free pointer goes to disk:

Values obtained state : txn3 Result : No issues.

6. Everything gets in except database pointer - it is 48 bytes - but a crash happens such that second 24 bytes (which contains the root) is missed:

Values obtained state: txn2 Result : Error: lib/mdb.c:1921: mdb_page_touch: Assertion ‘mp->mp_p.p_pgno != pgno’ failed.

7. Everything gets in except database pointer - it is 48 bytes - but a crash happens such that first 24 bytes (which contains depth, br_pages, leaves count etc) is missed:

Values obtained state: txn4 Result : No issues.

Conclusion: As we can see, the database is always consistent with key value pairs. But the internal consistency of the database is not maintained which is revealed by subsequent operations. As already mentioned, the invariant free_pages + current_pages + 2 metapages = in_use_pages -1, does not hold in some cases after crashes and that is the reason for all seen errors.