facebook

Transactions in RocksDB

Anthony Giardullo
Software Engineer
July 1, 2015

RocksDB

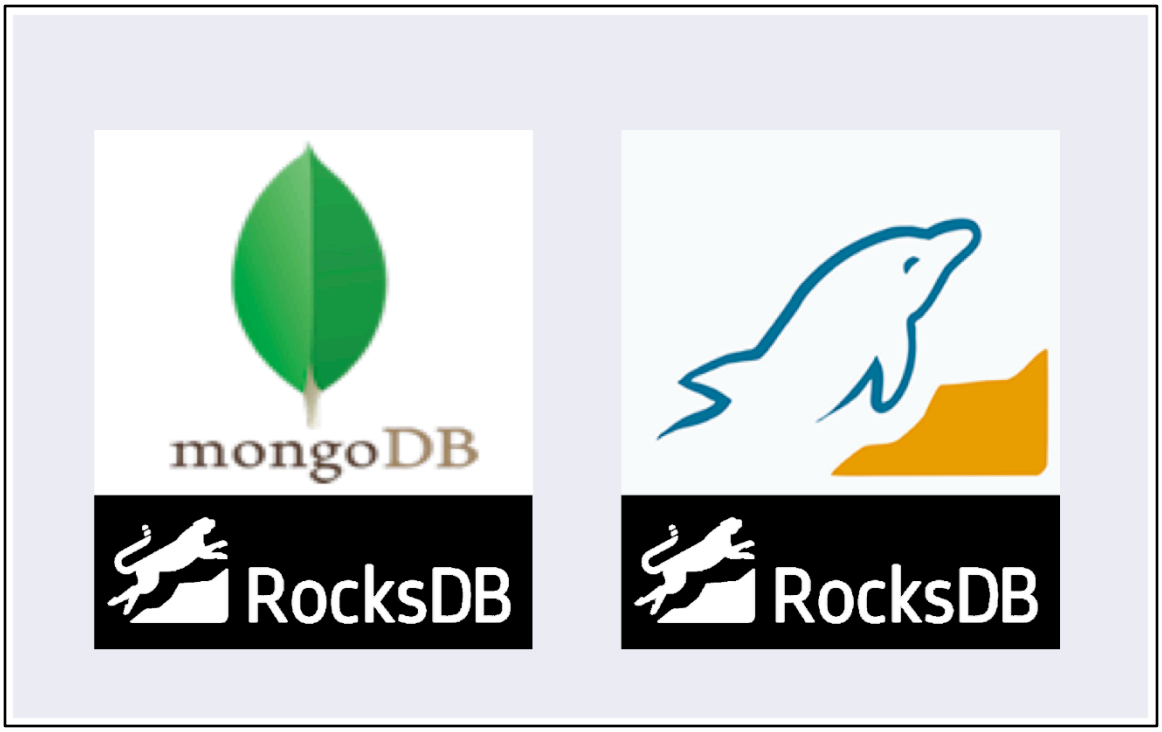
Why transactions?

Rocksdb already has several features that help you deal with concurrency: atomic batches writes, snapshot reads, and merge operators.
*However, in order to support more complicated application logic, many users need support for transactions with a full BEGIN/COMMIT/ROLLBACK –style API.

Here's a simple example- Assume the balance of my checking account is stored as a key in rocksdb.

*How could you safely increment the balance of my checking account? You could use a <Merge operator> to do an atomic read-modify-write.
*What if you want to transfer money from one account to another? You could use an <atomic write> to guarantee that the value of one key only gets decremented if the value of the other key gets incremented.
*What if at the same time as all these operations are going on, you needto sum up all users bank accounts? You could do a <snapshot read> to get a consistent view of the database at a particular point in time.

So RocksDB's features are pretty powerful. Why do we need transactions?



At Facebook we have many use-cases for transactions.

In order to run MongoDB on RocksDB, Igor had to write a ton of custom code to support MongoDB transactions. If we had transaction support in rocksdb, we wouldn't need to do all this extra work above rocksdb.

Another big project we have planned is using rocksdb as a storage engine for MySQL. Facebook relies on MySQL pretty heavily to store core user data, so this is a big goal for us.

Current Status

Optimistic Transactions (in master)

- Lightweight, less overhead
- Best for workloads with low contention
- Commit() can fail

Pessimistic Transactions (in development)

- Contention resolved by taking locks
- Commit() should not fail*

*Currently, we have support for Optimistic Transaction in the master branch (and will be in the next release of Rocksdb).
*We are working on support for Pessimistic Transactions.

How many people here know the difference between optimistic and pessimistic concurrency control?

Optimistic Transactions are “optimistic” in the sense that they assume there won’t be any conflicts and will wait until commit time to verify that we’re conflict free.
-This is lightweight as very little work is done for most transaction operations.
-This work well for workloads where write conflicts are rare. If you have too many transactions trying to update the same keys, you can get a lot of transaction failures.
-Some users want to know that a transaction will succeed before they call commit. They don’t like having to deal with commit() failures.

Pessimistic transactions are the kind of transactions most SQL relational database users are familiar with.
-Write operations try to lock keys, and if they can’t the write fails.
-Once we are ready to commit, we know that there are no conflicts since we successfully locked all the keys we care about.
**The only way a commit can fail is if you are unable to write to your db (say your out of space or have a hardware failure.)

API

```
// Open a TransactionDB or OptimisticTransactionDB
TransactionDB::Open(options, path, &txn_db);
OptimisticTransactionDB::Open(options, path, &txn_db);

// Start a Transaction
Transaction* txn = txn_db->BeginTransaction(write_options);

// Write to the Transaction
status = txn->Put(key, value);
status = txn->Delete(key);
status = txn->Merge(key, value);


// Read from Transaction
status = txn->Get(read_options, key, &value);
status = txn->GetForUpdate(read_options, key, &value);

// Commit or Rollback
Status = txn->Commit();
txn->Rollback();
```

The API for transactions is flexible and pretty similar to existing RocksDB APIs.

-OptimisticTransactionDB or TransactionDB can be opened similarly to opening a normal db.

-To start a txn, you call BeginTransaction and get back a txn handle.
**all operations you do on this handle are part of this transaction. Any operation you do on the db directly, is not a part of this transaction.

You can write data similar to the way you write to the db (or to a WriteBatch)
-If you're using a OptimisticTransactionDB, these writes will always return Success(optimistic transactions don't do conflict checking until commit time).
-If you're using a TransactionDB, these writes could fail if we cannot lock the key.

Get shows you the state of the db if this transaction were to be committed.
-It essentially merges the pending changes in the transaction with the data in the db.

You can also do a GetForUpdate, which behaves similar to a sql select for update.
-GetForUpdate will make sure that no other transaction can modify these keys.
**For pessimistic transactions, we will take a lock, for optimistic, conflicts are checked at commit time.

DB Isolation level?

- Read Committed
- Repeatable Read
- Almost Serializable
  - (Phantom write detection not currently supported)

LOCK "APPLE"
LOCK "BANANA"
INSERT "GRAPE"
LOCK "ORANGE"
LOCK "PEAR"

RocksDB Transactions have a flexible api and we let the user to decide what level of isolation they want.
If you're not familiar with common transaction isolation levels, don't worry about it, I'll give some examples on the next slides.
TRADEOFF between parallelism and isolation

RocksDB Transacitons support RC, RR, and "almost serializeable.
-The reason we don't yet support fully serializable transaction is because we don't currently support detecting phantom reads.
-What are phantom reads? Suppose I had a table of all the fruits in my kitchen and I locked all of these fruits.
--What if someone then inserts GRAPE into this table? I no longer have all fruits locked since I didn't know there was a such thing as a grape.
--Many databases solve this by supporting GAP LOCKING to lock the gaps between existing rows.
--We decided not to implement any gap locking for now because it turns out we don't need it.

So RocksDB has a key-value API not a SQL api. So we behave a bit differently. Instead of just setting an isolation level, we give users the tools they need to tune their isolation…

Read Committed

```
db->Get(read_options, key, &value1);
db->Get(read_options, key, &value2);

txn->Get(read_options, key, &value1);
txn->Get(read_options, key, &value2);
```

Repeatable Read

```
read_options.snapshot = mySnapshot;
db->Get(read_options, key, &value1);
db->Get(read_options, key, &value2);

read_options.snapshot = mySnapshot;
txn->Get(read_options, key, &value1);
txn->Get(read_options, key, &value2);
```

What is the difference between Read Committed and Repeatable Read?

If you read the same key twice from the db, you may get a different value the second time.
-By default this is the same when using a transaction.

If you wanted repeatable reads, you can simply set a snapshot in your read_options.
-This is the same behavior whether you're reading from a txn or from the db.

Note: neither option is better than the other. The correct solution depends on what the application wants to do.

SetSnapshot()

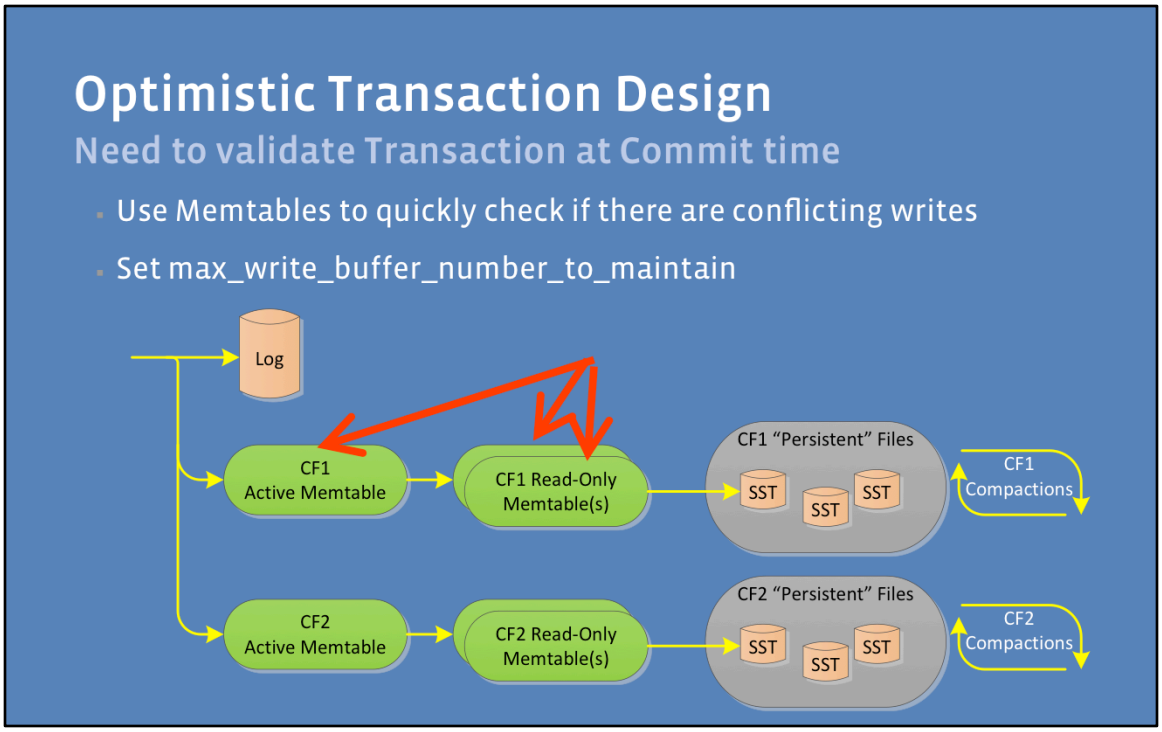
```
status = txn->Get(read_options, key, &value);
new_value = modify(value);
if (new_value != value)
  status = txn->Put(key, new_value);
```

```
status = txn->GetForUpdate(read_options, key, &value);
new_value = modify(value);
if (new_value != value)
  status = txn->Put(key, new_value);
```

```
txn->SetSnapshot();
status = txn->Get(read_options, key, &value);
new_value = modify(value);
if (new_value != value)
  status = txn->Put(key, new_value);
```

We also support an option called SetSnapshot(). Before I explain what this does, let me give an example…

1) Read modify write is dangerous here. It's possible someone else changed out key in between the time we read it and wrote it.
2) GetForUpdate solves the read-modify-write. But this solution isn't always the best. You need to know in advance that you want to write this key.
3) SetSnapshot() guarantees that the Put will only be committed if the key hasn't changed since set snapshot was called. SetSnapshot can be called multiple times throughout a transaction.

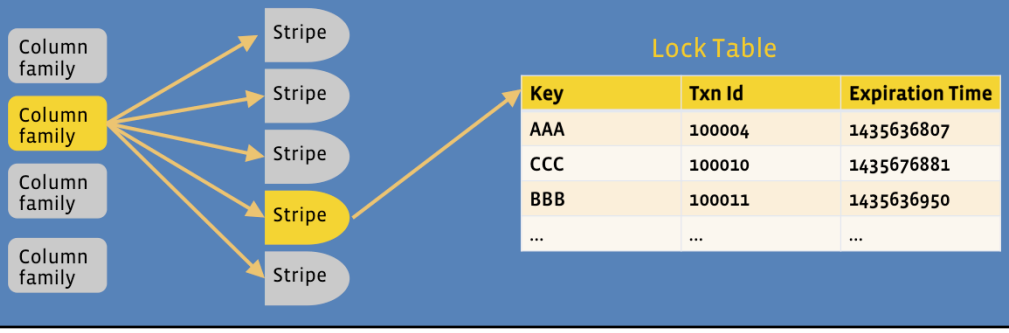


RocksDB is Log Structured database. For each column family, writes are written to in-memory memtables before being flushed to the LSM Tree persistent files.

Pessimistic Transaction Design

Store lock map of keys that are currently locked

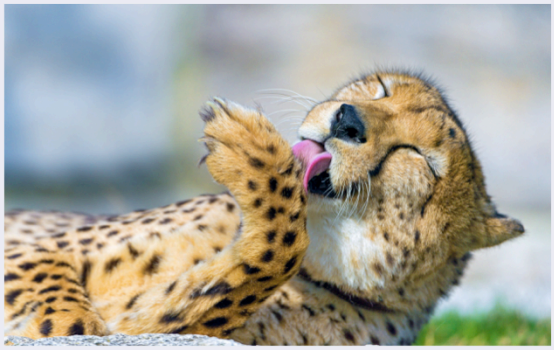
- No deadlock detection. Use lock timeouts.
- SetSnapshot() still uses memtable history.
- Txn Expiration, SavePoints



No deadlock detection
SetSnapshot uses Memtable history
Expiration, savepoints

Future work

- Shared Locking
- Better Comparator support
- Support for DBWithTTL
- Long running transactions



----- Meeting Notes (6/30/15 16:59) -----
long running transactions


RocksDB
Questions?