Tokutek®

Transactions and MongoDB:
ACID Reliability and MVCC with TokuMX

Technology Whitepaper

December 2013

"The real end-game for Big Data is to have transactional and analytic data on the same database." -- David Floyer, Wikibon

Executive Overview

MongoDB has quickly become the leading NoSQL database, and it is no wonder. The language interface makes it drop-dead easy to adopt MongoDB, and the ability to change schema on the fly allows development teams to build and modify their applications without unnecessary constraints. With Big Data applications and analytics requirements growing at an explosive rate, MongoDB is quickly stepping up to meet these needs.

One feature that is not in the MongoDB codebase is support for multi-document transactions or MVCC. Without transaction support, reads and writes can collide and applications get served inconsistent data – i.e. part of the read is pre-transaction data, part of the read is post-transaction data and the onus is on the developer to work around this behavior. If you are analyzing terabytes of clickstream data it may not matter if a few bits are off, but as MongoDB is being adopted in broader application areas, and as DBAs are taking more of an interest in MongoDB, support for transactions is of growing importance.

Currently there are three options when considering transactions:

- **Live without** – this, being the most popular approach to date, dictates that you avoid applications where transactions are important. If you need them, go elsewhere.
- **Build your own** – for minimal support, many have chosen to build support for transactions into their application code. This is a slippery slope and chips away at the ease-of-use appeal of MongoDB.
- **Use TokuMX** – The root cause for the lack of transactions is the implementation of the storage code in MongoDB. TokuMX – which does not require any changes to your application - gives you transactions with MVCC and ACID reliability.

TokuMX is an open source distribution of MongoDB that makes MongoDB more performant in large applications with demanding requirements. In addition to providing support for transactions, it also drastically improves performance (20x) and compression (90%).

MongoDB Transaction Capabilities with TokuMX

Tokutek's TokuMX, a performance engine for MongoDB, takes MongoDB's basic transactional behavior and extends it. MongoDB is transactional with respect to one, and only one, document. MongoDB guarantees single document atomicity. Journaling provides durability for that document. The database level reader/writer lock provides consistency and isolation.

TokuMX, on non-sharded clusters, extends these ACID properties to multiple documents and statements in a concurrent manner. Transactions take document-level locks instead of database-level locks, leading to more concurrency for both in memory and out of memory read/write workloads.

The behavioral differences include:

- **Each statement is a transaction** - For each statement that tries to modify a TokuMX collection, either the entire statement is applied or none of the statement is applied. A statement is never partially applied.
- **There are new commands for transactions** - Commands “beginTransaction”, “commitTransaction” and “rollbackTransaction” allow users to perform multi-statement transactions.
- **Queries use multi-version concurrency control (MVCC**) – With TokuMX, queries operate on a snapshot of the system that does not change for the duration of the query. Concurrent inserts, updates and deletes do not affect query results. Note that this does not include file operations like removing a collection.

We will explore each of these points in greater depth.

Each Statement is a Transaction

With MongoDB, if a statement fails (e.g. a batched insert) then that statement may be partially applied to the database. What part of the statement succeeds and fails depends on the scenario.

With TokuMX, either the entire statement succeeds or fails. There is no in-between.

Let’s look at an example. Take the following commands run in a shell:

```
> db.createCollection("foo")
{ "ok" : 1 }
> db.foo.insert([ {_id : 1 }, { _id : 2 } , { _id : 1 } , { _id : 3 } ])
```

With both TokuMX and MongoDB, the insertion statement will fail because the statement is inserting a value of 1 for the unique _id field in two separate documents.

Now let’s look at the state of the collection after both MongoDB and TokuMX try and fail to execute the insertion statement. With MongoDB, querying the collection shows the insert statement is partially applied. The first two documents were inserted, but when the third hit a duplicate key error, the first two remained in the collection:

```
> db.foo.find()
{ "_id" : 1 }
{ "_id" : 2 }
```

TokuMX, on the other hand, keeps the collection empty, acting as though the original statement never executed:

```
> db.foo.find()
```

Hence, each statement is transactional.

Commands for multi-statement transactions

The following commands and shell wrappers have been added:

- beginTransaction
- commitTransaction
- rollbackTransaction

The beginTransaction command supports the following isolation levels, passed in the form { isolation : … }:

- mvcc (the default)
- serializable
- readUncommitted

So, if one wants to create a serializable transaction, one would run:

```
db.runCommand( { beginTransaction : 1 , isolation : “serializable” } )
```

Here is an example of a multi-statement transaction being started and rolled back:

```
> db.foo.insert({ a : "inserted before begin" })
> db.foo.find()
{ "_id" : ObjectId("5271a25ef66f424a03b2158b"), "a" : "inserted before
begin" }
> db.runCommand("beginTransaction")
{ "status" : "transaction began", "ok" : 1 }
> db.foo.insert({ a : "inserted during transaction" })
> db.foo.find()
{ "_id" : ObjectId("5271a25ef66f424a03b2158b"), "a" : "inserted before
begin" }
{ "_id" : ObjectId("5271a280f66f424a03b2158c"), "a" : "inserted during
transaction" }
> db.runCommand("rollbackTransaction")
{ "status" : "transaction rolled back", "ok" : 1 }
> db.foo.find()
{ "_id" : ObjectId("5271a25ef66f424a03b2158b"), "a" : "inserted before
begin" }
```

Note that each multi-statement transaction *must* occur over a single connection. The connection stores the multi-statement transaction’s information. Be careful with drivers that use connection pools. If each statement goes through a connection pool to get a connection, then a multi-statement transaction will not work.

MVCC Queries

The best way to demonstrate this is with another example. Suppose we have a collection with only one element:

```
> db.foo.find()
{ "_id" : ObjectId("52702a22f66f424a03b2158a"), "a" : "inserted before
snapshot" }
```

Then, we create a multi-statement MVCC transaction:

```
> db.runCommand("beginTransaction")
{ "status" : "transaction began", "ok" : 1 }
```

The transaction's MVCC snapshot has one, and only one, document in it so all queries should show just that one document.

Suppose, in another connection, we insert a document to the collection and query the collection to show that it is there:

```
> db.foo.insert({ a : "inserted while transaction is live" })
> db.foo.find()
{ "_id" : ObjectId("52702a22f66f424a03b2158a"), "a" : "inserted before
snapshot" }
{ "_id" : ObjectId("52702a6bafef85c2510956e8"), "a" : "inserted while
transaction is live" }
```

If we go back to the first connection that started a multi-statement transaction and performed a query, we see only the one document that was part of the original snapshot:

```
> db.foo.find()
{ "_id" : ObjectId("52702a22f66f424a03b2158a"), "a" : "inserted before
snapshot" }
```

I use a multi-statement transaction here to easily show the effect of concurrent queries and inserts (the same can be said for updates and deletes). But, these properties hold for single statements as well. Any statement executing a find will perform the query on a snapshot of the data. Concurrent inserts, updates and deletes do not impact the query's result.

Durability of TokuMX Transactions

One of the concerns users have with using durable transactions, be it with TokuMX or MongoDB, is the requirement of flushing a recovery log to disk to ensure the transaction will survive a crash. For MongoDB, that "recovery log" is the journal. If each transaction requires a flush of log information to disk or an fsync, and disks may be limited to several hundred fsyncs per second, then throughput is limited by the number of fsyncs. While a form of group commit helps each of these products on multi-threaded workloads, the bottleneck caused by fsyncs may be an issue for some applications.

To give users a way around this limitation, TokuMX and MongoDB use a variable that has transactions not flush the recovery log to disk on commit and instead have a background thread periodically flush the recovery log. That variable is "logFlushPeriod". The intent is to give users an option to have non-durable transactions and instead periodically flush recovery log data to disk in the background. The end effect is that on a crash, the database loses at most a period's worth of data.

By default, MongoDB and TokuMX flush the recovery log to disk once every 100ms, so transactions by default are not durable. On some systems, the default for MongoDB is 30ms. With TokuMX, we decided to emulate MongoDB's default behavior.

To make a MongoDB update/insert/delete be durable, you must subsequently call the "getLastError" command with the option { j : true }. The same works for TokuMX. With TokuMX, to make all transactions durable by default, you can simply set the value of logFlushPeriod to 0. With this setting, you do not need to call getLastError to ensure durability.

Benefits of TokuMX Transactions

From the application's perspective, TokuMX behaves very similar, if not identically, to MongoDB in many ways. But in one subtle yet important way, on non-sharded clusters, TokuMX is different. With MongoDB, operations on each single document are transactional. With TokuMX, as explained above, each *statement* is transactional. There are several reasons for doing this and the top 4 that benefit MongoDB applications are explained below.

Benefit 1: Cursors represent a true snapshot of the system

In MongoDB, a cursor for a query returns data in batches. If all of the data does not fit in a single batch, then the driver must call getMore to receive more results. If a write interleaves between these calls, then the result of getMore may be affected. One possibility is that if an update moves one document from a location, the cursor has read to a location that the cursor will soon read via getMore and that document will show up twice in the query.

With TokuMX, this is not a concern. Cursors are associated with transactions that have MVCC. As a result, cursors will have a snapshot of the system that is unaffected by concurrent inserts, deletes or updates. So, one does not need to be concerned with interleaving writes affecting query results.

Benefit 2: Simpler to batch inserts together for performance

Batching inserts together is a common method for improving insertion performance over sending insertions one at a time. By batching the insertions together, some overhead work is amortized. For example, only one round trip is taken over the network instead of one per document. Some locks are grabbed only once as opposed to once per document.

While TokuMX and MongoDB can both batch inserts together and both see performance improvements, what makes TokuMX simpler is the error handling. With MongoDB, if you send 1000 inserts in a single batch and the command fails for some reason, then the user does not know if some subset of the 1000 insertions were applied before the command failed. Did 100 of the insertions make it? Did 500 make it? What is the state of the system? With TokuMX, because each statement is its own transaction, the user knows that either the entire statement was applied or none of it was applied. The user does not need to check if the statement was partially applied.

Benefit 3: Simpler for applications to update multiple documents with a single statement

This benefit is very similar to the benefit above. Just as batching multiple insertions together may be problematic with MongoDB, updating multiple documents with a single statement is similarly problematic. If the update fails, the application is responsible for determining what subset of the update statement succeeded and what subset failed. For this reason, some users don’t run with { multi : true } on their update statements.

With TokuMX, because each statement is a transaction, this is not necessary.

Benefit 4: No need to combine documents together for atomicity

MongoDB does not support atomicity with updating multiple documents. One workaround is to combine the documents into a single document to take advantage of MongoDB’s single document atomicity. This requires extra effort and adds code unrelated to the business logic. With TokuMX, this effort is unnecessary.

About TokuMX

TokuMX is an open source performance engine for MongoDB that makes MongoDB more performant in large applications with demanding requirements. TokuMX replaces the storage engine of the database (known as B-tree indexing) with Tokutek’s patented Fractal Tree® indexing.

The more modern indexing technology delivers

- **MongoDB Performance** – With significant improvement in insertions and indexing, TokuMX delivers faster, more complex ad hoc queries in live production systems without rewriting or tuning applications even when tables are too large for memory. Read the performance benchmark.
- **MongoDB Compression**: By leveraging write-optimized compression, TokuMX achieves up to a 90% reduction in HDD and flash storage requirements without

impacting performance, and reduces the number servers needed to host a MongoDB database.

- **MongoDB Transactions**: TokuMX adds transactions with MVCC and ACID reliability to any MongoDB database making it possible to take advantage of the leading NoSQL database in a wider range of applications.

TokuMX is open source, and is available for free download at tokutek.com/download.

Conclusion

If you love MongoDB and also want transaction support, you can have both with Tokutek's distribution of MongoDB, TokuMX. You can adopt TokuMX without making changes to your code (except where you want to add transaction commands) and spend less time developing workarounds. As an added bonus, you will also see sizable improvements in performance and compression. Visit Tokutek.com for benchmark reports.

For more information

- **Download TokuMX** – TokuMX is open source - download it for free and see how it works for you (http://www.tokutek.com/download)
- **Read the analyst report** – CITO Research conducted their own review of TokuMX (http://forms.tokutek.com/acton/form/6118/0011:d-0004/0/index.htm)
- **Watch a webinar** – Tokutek has many live and archived webinars about MongoDB and TokuMX (http://www.tokutek.com/resources/webinars/)

If you have any additional questions about TokuMX, do not hesitate to contact us.

About Tokutek, Inc.

Tokutek is a performance engine company that delivers Big Data to the leading open source data management platforms. Tokutek applies patented Fractal Tree® indexing to increase MySQL performance and MongoDB performance, decrease database size and minimize downtime. Tokutek's breakthrough technology lets customers build a new class of applications that handle unprecedented amounts of incoming data and scale with the data processing needs of tomorrow. The company is headquartered in Lexington, MA, and has offices in New York, NY.

Web: tokutek.com
email: contact@tokutek.com

57 Bedford St., Suite 101
Lexington, MA 02420

Follow us on Twitter, Linkedin, and Facebook