MongoDB Multi-Document ACID Transactions

May 2020

Table of Contents

Introduction

Support for multi-document ACID transactions debuted in the MongoDB 4.0 release in 2018, and were extended in the 2019 4.2 release with the addition of Distributed Transactions for sharded clusters.

MongoDB's existing atomic single-document operations already provided transaction semantics that meet the data integrity needs of the majority of applications. The addition of multi-document ACID transactions now makes it even easier for developers to address the full spectrum of use cases with MongoDB. Offering snapshot isolation and all-or-nothing execution, your applications maintain transactional data integrity even across highly distributed sharded clusters

For developers with a history of transactions in relational databases, MongoDB's multi-document transactions are very familiar, making it straightforward to add them to any application that requires them.

In this whitepaper, we will explore why MongoDB added multi-document ACID transactions, their design goals and implementation, best practices for developers, and the engineering investments made over the past 5+ years to lay the foundations for them. You can get started with MongoDB now by spinning up your own fully managed, on-demand MongoDB Atlas cluster, or downloading it to run on your own infrastructure.

Why Multi-Document ACID Transactions?

Since its first release in 2009, MongoDB has continuously innovated around a new approach to database design, freeing developers from the constraints of legacy relational databases. A design founded on rich, natural, and flexible documents accessed by idiomatic programming language APIs, enabling developers to build apps 2-3x faster. And a distributed systems architecture to handle more data, place it where users need it, and maintain always-on availability. This approach has enabled developers to create powerful and sophisticated applications in all industries, across a tremendously wide range of modern use cases.

Figure 1: Organizations innovating with MongoDB

With subdocuments and arrays, documents allow related data to be modeled in a single, rich and natural data structure, rather than spread across separate, related tables composed of flat rows and columns. As a result, MongoDB's existing single document atomicity guarantees can meet the data integrity needs of most applications. In fact, when leveraging the richness and power of the document model, we estimate 80%-90% of applications don't need multi-document transactions at all.

However some developers and DBAs have been conditioned by 40 years of relational data modeling to assume multi-record transactions are a requirement for any database, irrespective of the data model they are built upon. Some are concerned that while multi-document transactions aren't needed by their apps today, they might be in the future. And for some workloads, support for ACID transactions across multiple records is required.

As a result, the addition of multi-document transactions makes it easier than ever for developers to address a complete range of use cases on MongoDB. For some, simply knowing that they are available assures them that they can evolve their application as needed, and the database will support them.

> *"ACID transactions are a key capability for business critical transactional systems, specifically around commerce processing. No other database has both the power of NoSQL and cross-collection ACID transaction support. This combination will make it easy for developers to write mission critical applications leveraging the power of MongoDB."*

Dharmesh Panchmatia, Director of E-commerce, Cisco Systems

Data Models and Transactions

Before looking at multi-document transactions in MongoDB, we want to explore why the data model used by a database impacts the scope of a transaction.

Relational Data Model

Relational databases model an entity's data across multiple records and parent-child tables, and so a transaction needs to be scoped to span those records and tables. The example in Figure 2 shows a contact in our customer database, modeled in a relational schema. Data is normalized across multiple tables: customer, address, city, country, phone number, topics and associated interests.

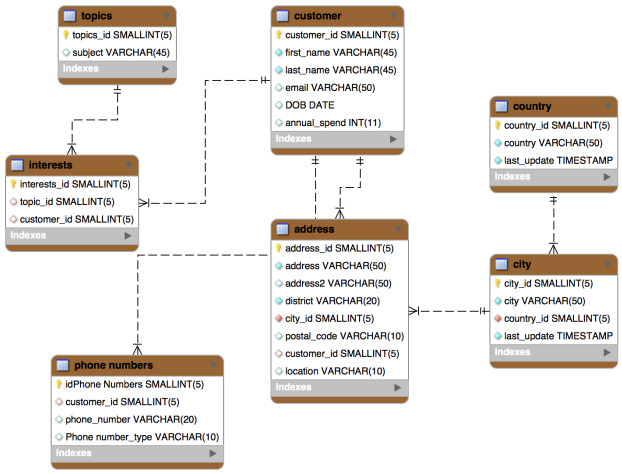

Figure 2: Customer data modeled across separate tables in a relational database

In the event of the customer data changing in any way, for example if our contact moves to a new job, then multiple tables will need to be updated in an "all-or-nothing" transaction that has to touch multiple tables, as illustrated in Figure 3.

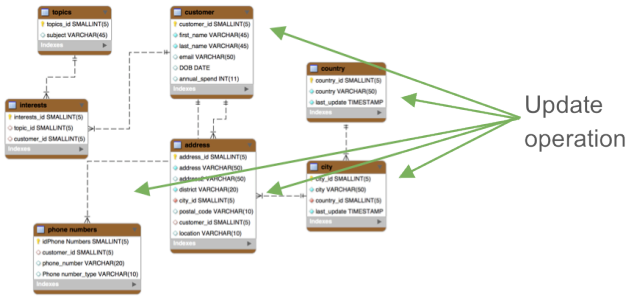

Figure 3: Updating customer data in a relational database

Document Data Model

Document databases are different. Rather than break related data apart and spread it across multiple parent-child tables, documents can store related data together in a rich, typed, hierarchical structure, including subdocuments and arrays, as illustrated in Figure 4.

```
  _id: 12345678
> name: Object
> address: Array
> phone: Array
  email: "john.doe@mongodb.com"
  dob: 1966-07-30 01:00:00.000
v interests: Array
    0: "Cycling"
    1: "IoT"
```

Figure 4: Customer data modeled in a single, rich document structure

MongoDB has always provided existing transactional properties scoped to the level of a document. As shown in Figure 5, one or more fields may be atomically written in a single operation, with updates to multiple subdocuments and elements of any array, including nested arrays. The guarantees provided by MongoDB ensure isolation as a document is updated; any errors cause the operation to roll back so that clients receive a consistent view of the document. With this design, application owners get the same data integrity guarantees as those provided by traditional relational databases.

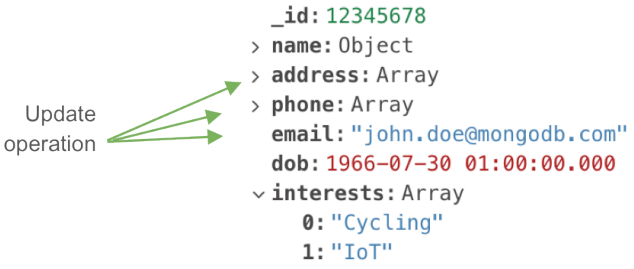

Figure 5: Updating customer data in a document database

Where are Multi-Document Transactions Useful?

There are use cases where transactional ACID guarantees need to be applied to a set of operations that span multiple documents, most commonly with apps that deal with the exchange of value between different parties and require "all-or-nothing" execution. Back office "System of Record" or "Line of Business" (LoB) applications are the typical class of workload where multi-document transactions are useful. Examples include:

- Moving funds between bank accounts, payment processing systems, or trading platforms – for instance where new trades are added to a tick store, while simultaneously updating the risk system and market data dashboards.
- Transferring ownership of goods and services through supply chains and booking systems – for example, inserting a new order in the orders collection and decrementing available inventory in the inventories collection.
- Billing systems – for example adding a new Call Detail Record and then updating the monthly call plan when a cell phone subscriber completes a call.
- Redistributing data across a sharded cluster by modifying a document's shard key value.

MongoDB already served many of these use cases before the addition of multi-document transactions. Now it is even easier for developers as the database automatically handles multi-document transactions for them. Before their availability, the developer would need to programmatically implement transaction controls in their application. To ensure data integrity, they would need to ensure that all stages of the operation succeed before committing

updates to the database, and if not, roll back any changes. This adds complexity that slows down the rate of application development. MongoDB customers in the financial services industry have reported they were able to cut 1,000+ lines of code from their application by using multi-document transactions.

In addition, implementing client side transactions can impose performance overhead on the application. For example, after moving from its existing client-side transactional logic to multi-document transactions, a global enterprise data management and integration ISV experienced improved MongoDB performance in its Master Data Management solution: throughput increased by 90%, and latency was reduced by over 60% for transactions that performed six updates across two collections. Beyond client side code, a major factor in the performance gains came from write guarantees. For each individual write operation in the transaction, the app previously had to wait for acknowledgement from the majority write concern that the operation had been propagated across a majority of replicas. Only once this acknowledgement was received could it then progress to the next write in the transaction. With multi-document transactions, the app only has to wait for the majority acknowledgement when it comes to commit the transaction.

Multi-Document ACID Transactions in MongoDB

Transactions in MongoDB feel just like transactions developers are used to in relational databases. They are multi-statement, with similar syntax, making them familiar to anyone with prior transaction experience.

The Python code snippet below shows a sample of the transactions API.

```
with client.start_session() as s:
       s.start_transaction()
       collection_one.insert_one
            (doc_one, session=s)
       collection_two.insert_one
            (doc_two, session=s)
       s.commit_transaction()
```

The following snippet shows the transactions API for Java.

```
try (ClientSession clientSession
           = client.startSession()) {
        clientSession.startTransaction();
        collection.insertOne(clientSession, docOne);
        collection.insertOne(clientSession, docTwo);
        clientSession.commitTransaction();
    }
```

As these examples show, transactions use regular MongoDB query language syntax, and are implemented consistently whether the transaction is executed across documents and collections in a replica set, and with MongoDB 4.2's Distributed Transactions, across a sharded cluster.

The transaction block code snippets below compare the MongoDB syntax with that used by MySQL. It shows how multi-document transactions feel familiar to anyone who has used traditional relational databases in the past.

MySQL

```
db.start_transaction()
    cursor.execute(orderInsert, orderData)
    cursor.execute(stockUpdate, stockData)
db.commit()
```

MongoDB

```
s.start_transaction()
    orders.insert_one(order, session=s)
    stock.update_one(item, stockUpdate,
        session=s)
s.commit_transaction()
```

Through snapshot isolation, transactions provide a consistent view of data, and enforce all-or-nothing execution to maintain data integrity. Transactions can apply to operations against multiple documents contained in one, or in many, collections and databases. The changes to MongoDB that enable multi-document transactions do not impact performance for workloads that don't require them.

During its execution, a transaction is able to read its own uncommitted writes, but none of its uncommitted writes will be seen by other operations outside of the transaction. Uncommitted writes are not replicated to secondary nodes until the transaction is committed to the database. Once the transaction has been committed, it is replicated and applied atomically to all secondary replicas.

Read and Write Concerns

To enforce transactional behaviour across replica sets and shared clusters, it is important for developers to configure the read and write concerns recommended in the MongoDB transactions documentation at the start of each transaction.

The majority write concern should be used to ensure write durability in the event of primary elections, and read concerns should be configured with majority or snapshot.

Starting with MongoDB 4.4 read and write concerns can be configured cluster-wide, allowing application owners to set the most appropriate defaults server-side. By configuring these stronger global default read and write concerns, users no longer need to explicitly configure settings for each transaction. Behavior can be standardized across all clients whether they are running MongoDB in Atlas, or managing MongoDB themselves. Users still have the flexibility to override the default on a per-client or per-session basis in the driver if they want to.

> *"We're excited to see MongoDB offer dedicated support for ACID transactions in their data platform and that our collaboration is manifest in the Lovelace release of Spring Data MongoDB. It ships with the well known Spring annotation-driven, synchronous transaction support using the MongoTransactionManager but also bits for reactive transactions built on top of MongoDB's ReactiveStreams driver and Project Reactor datatypes exposed via the ReactiveMongoTemplate."*

Pieter Humphrey - Spring Product Marketing Lead, Pivotal

Global Point in Time Reads

Taking advantage of the transactions infrastructure in MongoDB, snapshot read isolation ensures queries and aggregations executed within a read-only transaction will operate against a globally consistent snapshot of the database across each primary replica of a sharded cluster.

As a result, a consistent view of the data is returned to the client, even when the data is distributed across shards, and as it is being simultaneously modified by concurrent write operations.

Transactions Best Practices

As noted earlier, MongoDB's existing document atomicity guarantees will meet 80-90% of an application's transactional needs. They remain the recommended way of enforcing your app's data integrity requirements. For those operations that do require multi-document transactions, there are several best practices that developers should observe.

Creating long running transactions, or attempting to perform an excessive number of operations in a single ACID transaction can result in high pressure on the WiredTiger storage engine's cache. This is because the cache must maintain state for all subsequent writes since the oldest snapshot was created. As a transaction always uses the same snapshot while it is running, new writes accumulate in the cache throughout the duration of the transaction. These writes cannot be flushed until transactions currently running on old snapshots commit or abort, at which time the transactions release their locks and WiredTiger can evict the snapshot. To maintain predictable levels of database performance, developers should therefore consider the following:

1. By default, MongoDB will automatically abort any multi-document transaction that runs for more than 60 seconds. Note that if write volumes to the server are low, you have the flexibility to tune your transactions for a longer execution time. To address timeouts, the transaction should be broken into smaller parts that allow execution within the configured time limit. You should also ensure your query patterns are properly optimized with the appropriate index coverage to allow fast data access within the transaction.
2. There are no hard limits to the number of documents that can be read within a transaction. As a best practice, no more than 1,000 documents should be modified within a transaction. For operations that need to modify more than 1,000 documents, developers should break the transaction into separate parts that process documents in batches.
3. As noted earlier, configure the appropriate read and write concerns for transactions.
4. When a transaction aborts, an exception is returned to the driver and the transaction is fully rolled back. Developers should add application logic that can catch

and retry a transaction that aborts due to temporary exceptions, such as a transient network failure or a primary replica election. With retryable writes, the MongoDB drivers will automatically retry the commit statement of the transaction.

5. Transactions that affect multiple shards incur a greater performance cost as operations are coordinated across multiple participating nodes over the network.
6. Transactions spanning shards will error and abort if any participating shard contains a replica set arbiter.
7. The performance of chunk migrations – used to rebalance data across shards – will be impacted if a transaction runs against a collection that is subject to rebalancing.

You can review all best practices for both replica set and Distributed Transactions in the MongoDB documentation for multi-document transactions.

The Path to Transactions

Our path to transactions represents a multi-year engineering effort, beginning back in 2015 with the integration of the WiredTiger storage engine. We've laid the groundwork in practically every part of the platform – from the storage layer itself to the replication consensus protocol, to the sharding architecture. We've built out fine-grained consistency and durability guarantees, introduced a global logical clock, refactored cluster metadata management, and more. And we've exposed all of these enhancements through APIs that are fully consumable by our drivers.

Figure 6 presents a timeline of the key engineering projects that have enabled multi-document transactions in MongoDB, with status shown as of June 2019. The key design goal underlying all of these projects is that their implementation does not sacrifice the key benefits of MongoDB – the power of the document model and the advantages of distributed systems, while imposing no performance impact to workloads that don't require multi-document transactions.

One of the hardest parts of the engineering effort has been how to balance two priorities. Firstly, building the stepping stones we needed to get to multi-document transactions starting in MongoDB 4.0. And secondly, immediately exposing useful features to our users, to make MongoDB the best way to work with their data. Wherever we could, we built components that satisfied both priorities:

- The acquisition of WiredTiger Inc. and integration of its storage engine back in MongoDB 3.0 brought massive scalability gains through document level concurrency control and compression. And with MVCC support, it also provided the storage layer foundations for multi-document transactions.
- In MongoDB 3.2, the enhanced consensus protocol allowed for faster and more deterministic recovery from the failure of the primary replica set member or network partitioning, along with stricter durability guarantees for writes. These enhancements were immediately useful to MongoDB users at the time, and they are also essential capabilities for transactions.
- The introduction of the readConcern query option in 3.2 allowed applications to specify the read isolation level on a per operation basis, providing powerful and granular consistency controls for users then, and the isolation levels required for multi-document transactions in MongoDB 4.0 and beyond.

MongoDB 3.6 brought a host of new functionality, much of which was underpinned by the introduction of a global logical clock and storage layer timestamps, based on an implementation of the earlier academic concepts of Lamport clocks and timestamps. This functionality enforces consistent time across every operation in a distributed cluster, enabling multi-document transactions to provide snapshot isolation guarantees. And it also allowed us to expose additional capabilities that further improved developer productivity as soon as MongoDB 3.6 was released:

- Change streams enable developers to build reactive applications that can view, filter, and act on data changes as they occur in the database, in real time. The logical clock and timestamps give change streams resumability – automatically recovering from transient node failures, so consuming apps can just pick up applying changes from the point when the node failure occurred.
- Logical sessions are the foundation for both causal

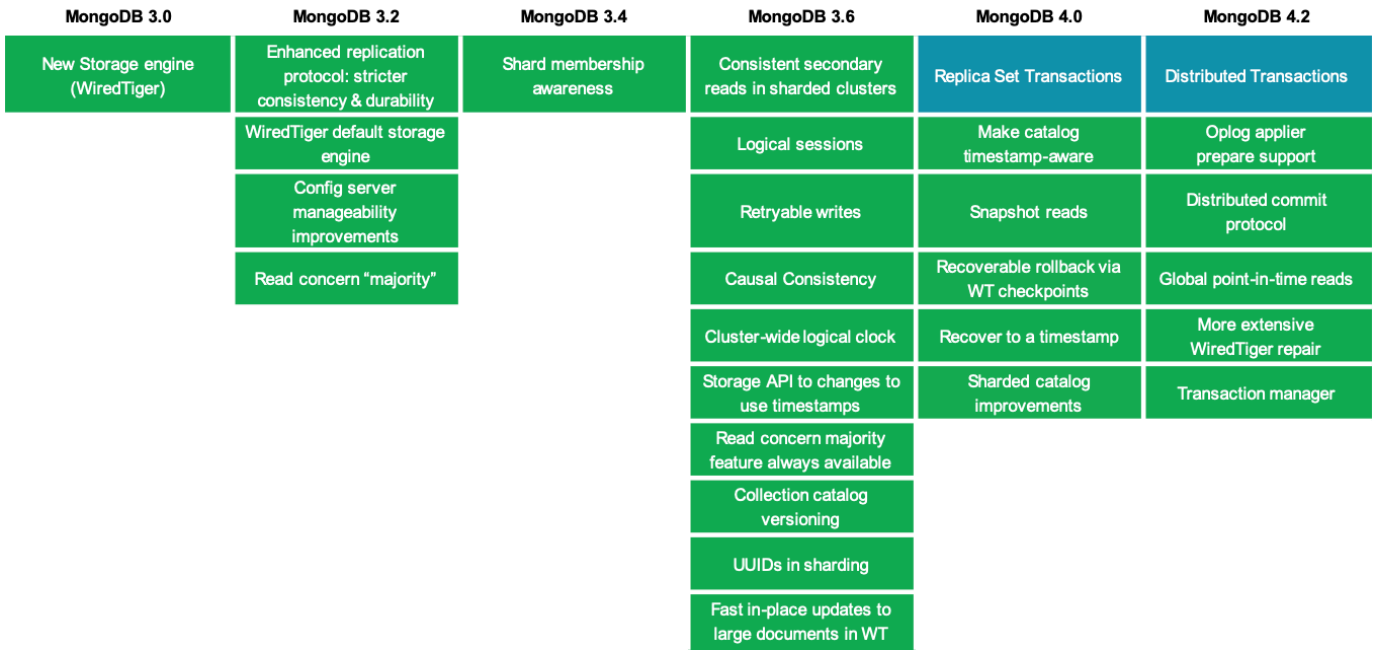

Figure 6: The path to transactions – multi-year engineering investment, delivered across multiple releases

consistency and retryable writes, discussed below. From the perspective of multi-document transactions, their value is the ability to coordinate client and server operations across distributed nodes, managing the execution context for each statement in a transaction.

- Causal consistency, enabled by logical sessions and cluster time, allows developers to maintain the benefits of strong data consistency with "read your own write" guarantees, while taking advantage of the scalability and availability properties of our intelligent distributed data platform.
- Retryable writes simplify the development of applications in the face of elections (or other transient failures), while the server enforces exactly-once processing semantics.

MongoDB 4.0 saw the first of release of multi-document ACID transactions, scoped at that time to a replica set. MongoDB 4.2 added support for Distributed Transactions – enforcing ACID guarantees across sharded clusters. Two critical projects in the delivery of Distributed Transactions in 4.2 were the Oplog Applier Prepare Support and Distributed Commit Protocol:

- Prepare Support introduces a new state for a document that prevents threads from reading it while it is being committed atomically by a cross-shard transaction. This ensures MongoDB's consistency and isolation guarantees are enforced.
- The Distributed Commit Protocol selects a coordinator node which is responsible for executing the transaction across shards. The new protocol determines whether the transaction commits or aborts, enforcing MongoDB's all-or-nothing ACID guarantees.

Combined with Distributed Transactions, MongoDB 4.2 introduced additional enhancements that simplify application development:

- Large Transactions: By representing transactions across multiple oplog entries, you can now write more than 16MB of data in a single ACID transaction (subject to the existing 60-second default runtime limit). This means you no longer need to experiment to see whether you would hit the 16MB limit, especially for those use cases where you are atomically inserting multiple large documents into the database.
- Deep Transactions Diagnostics: Transactions metrics are now written to the mongod log to expose telemetry on execution time, locks, numbers of documents touched, time spent in the storage engine, and more, helping you tune application performance. In addition, the currentOp and serverStatus commands have been expanded with transaction-specific sections, enabling

developers to get instant oversight of all running transactions in the database.

- Transaction Error Handling: New driver-side helpers with a callback API make it easier for you to develop retry logic in your app that handles any transactions aborts.

MongoDB 4.4 introduces the ability to create new collections and indexes within a transaction.

Transactions Under the Covers

The addition of multi-document ACID transactions has required foundational enhancements in the underlying MongoDB server and storage engine. In the following part of the paper, we present those key enhancements. You can also get more detail on each of these from the Engineering Chalk and Talks videos on our transactions page.

Low-level timestamps in MongoDB & WiredTiger

Timestamps from MongoDB write operations now appear in the WiredTiger storage layer as additional metadata. This allows MongoDB's concept of time and order to be queried so that only data from or before a particular time is retrieved. It allows MongoDB snapshots to be created so that database operations and transactions can work from a common point in time.

Background: To enable replication MongoDB maintains an operational log, also known as the oplog. The oplog is a specialized collection within the server which lists the most recent operations that have been applied to the database. By replaying those operations on a secondary server, the replica can be brought up to date, maintaining a consistent state with the primary server. The ordering of operations in the oplog is critical to ensuring that the replicas correctly reflect the contents of the primary server.

MongoDB manages that ordering of the oplog and how replicas can access the oplog in the correct order. The switch to WiredTiger 's more powerful storage engine with its own underlying idea of order meant that to fully exploit WiredTiger's capabilities, the two ideas of order in the server and the storage layer would need to be reconciled.

WiredTiger Storage: WiredTiger stores all data in a tree of keys and values. When used as MongoDB's storage layer, that data can be a document or a part of an index. Both are stored in the WiredTiger tree. When any updates are made to a key's value, WiredTiger creates an update structure. This structure contains information about the transaction, the data that has changed and a pointer on to any later changes. WiredTiger then attaches that to the original value. Later update structures will append themselves to the previous update structure creating a chain of different versions of the value over time.

This is the multi-version component of the concurrency control implemented by WiredTiger. WiredTiger has its own rules for how to read the update structures to get the "current" state of a value. This order that WiredTiger applies updates is different from MongoDB's oplog order. That difference in ordering comes from WiredTiger, parallelizing multiple writes to secondaries when possible. As the primary can accept many parallel writes, secondaries need to be able to match that throughput with their own parallel writes for replication.

Timestamps: To preserve the MongoDB order within the WiredTiger storage engine, the update structure was extended with a "timestamp" field. The value for this field is passed into WiredTiger by MongoDB and is treated as a significant item of meta information by WiredTiger. When queries are made of WiredTiger, a timestamp can be specified with the query to get the exact state of the data at that particular moment. This provides a way to map between MongoDB order and WiredTiger storage order.

Secondary Reads: When a secondary replica synchronizes with the primary, it does so by reading batches of updates from the oplog. It then attempts to apply those changes to its own storage. Without timestamps, that applying of the operations would block read queries until a batch of updates were completed to ensure that no out of order writes are seen by users. With the addition of timestamps, it is now possible to continue read queries using the timestamp from the start of the current batch. That timestamp will ensure a consistent response to the queries. This means that secondary reads will now not be interrupted by replication updates.

Replication Rollback: When multiple secondary servers in a MongoDB cluster are updated through replication, they

will find themselves at different stages of synchronization with the primary. This fact means we also have the "majority commit point": the point in time where a majority of the secondary servers have caught up to. When a primary fails, only data up to that majority commit point is guaranteed to be available on all servers, and that's what the secondary's work with as one of them is elected to be a new primary with our RAFT-based consensus protocol.

When the former primary returns to the cluster, the process of synchronizing that server with the rest of the cluster was quite complex. As it may have had data from beyond the common point, it had to work out what changes it had done which the cluster no longer knew about and retrieve old versions of records that it had changed.

With timestamps, this process is radically simplified. By taking the timestamp of the majority commit point and applying it to the former primaries storage, changes that happened after that timestamp can be dropped. Once done, the node can rejoin the cluster and start replicating the primary.

Timestamps and Transactions: By pushing the timestamp information into the heart of the WiredTiger trees, it has enabled using WiredTiger's multi-version concurrency control to reduce locks and streamline resynchronization processes. The ability to snapshot a point in time also gives the server the ability to roll back to that point in time, a capability that is fundamental to the correctness guarantees of multi-document ACID transactions.

Logical Sessions

By creating logical sessions, it is now possible for the resources being used in a single operation or a multiple-operation transaction to be tracked as they are consumed throughout the system. This enables simple, precise cancellation of operations and distributed garbage collection.

Background: Historically, there have been many operations in MongoDB which would have benefitted from being tracked as they progressed from the client, through the mongos query routers and out to the shards and replica sets which make up a cluster. There was, though, no such identifier to track these operations with, and so the system relied on a range of heuristics.

The solution for MongoDB was to create the Logical Session and the Logical Session Identifier. This is a small unique identifier, referred to as the lsid, that can be attached by a client to its communications with the MongoDB cluster which will, in turn, attach the lsid to any resources being used by that client.

The lsid is automatically generated at the client by the MongoDB drivers. This eliminates the need to call out to central id service. It is composed of an id, which is a GUID (Globally Unique ID) generated by the client, and a uid, which is a SHA256 digest of the username.

From MongoDB 3.6 onwards, any client operation is associated with a Logical Session. The Logical Session Identifier, lsid, will then be associated with the operation of a command across the cluster.

Logical Sessions and Cancellation: Any operation consumes resources. For example, a find() operation will create cursors in all the relevant shards in a cluster. Each cursor will start acquiring results for its first batch to return. Before logical sessions existed, to cancel an operation like this would mean traversing all the shards with administration privileges, working out which activity was associated with your operation and then stopping it.

It also introduced additional complexities. Where say the mongos process associated with the issuing of the command went down, that cancellation process would be even harder and you would have to wait for the cursors to build their first batch and then timeout waiting to return it.

With logical sessions, the entire process is simpler. It is possible to issue a kill command for a specific logical session to the cluster. As all resources, including the cursors, are tagged with a logical session identifier, it becomes a relatively simple operation to stop and free those resources with a particular lsid. It is also possible, because the user id is part of the lsid, to issue a command that removes all session resources for a particular user.

Logical Sessions and Distributed Garbage Collection: Timeouts on resources in MongoDB have previously been a local affair; the node where the resource resided would decide whether or not the resource had timed out and

needed garbage collecting. With plans for future MongoDB features requiring that timeouts and garbage collection are aware of the cluster, changes were made, using lsids to enable them.

In MongoDB 3.6 mongod and mongos processes began to do two things. First, run a controller process to manage sessions, and secondly, to maintain a list of lsids that connected to the process in the controller. Every 5 minutes, controllers would then synchronize that list with a sessions collection, updating the time the session was last used. That last use time becomes the basis for a TTL index which triggers after 30 minutes, that is indicating that this is a session not being used by any controller for 30 minutes. This makes the session, and the resources it uses, eligible to be cleaned up.

Logical Sessions and Transactions: By tagging all the operations and resources with a logical session id, it has become easier to manage long-lived and widely distributed operations in MongoDB. The logical session id has immediate utility in cancellation and garbage collection scenarios and is a foundation for other MongoDB 4.0+ features. By ensuring a transaction take place within a session, it enables the storing and cleaning up of that transaction whether it successfully commits or aborts.

Local Snapshot Reads

For transactions to be effective it needs to be possible to look at the database at a particular point in time. That point in time consistency is made possible by low-level timestamps which identify data as part of a snapshot. The resources are tracked using logical sessions.

Background: The traditional way for a long-running query to operate on the server was to keep acquiring data from a single database snapshot until the cursor was told to yield. In the process of yielding, the cursor state was saved and the locks and snapshot of the database were released.

On returning the lock was reacquired, a new snapshot was obtained, the cursor restored and the long-running query would carry on. This meant that the snapshot time could move forward with each yield.

Retaining the Snapshot: Local Snapshot Reads simply do not release the snapshot or locks when yielding. The logical session from the client is used to record the locks and snapshot in the server and to track those resources to be released when an operation or transaction is complete. The use of the session also allows resource usage to be associated across all servers.

Using Local Snapshot Reads: To get the Local Snapshot Read behavior, a multi-document transaction has to be created within the client session with a ReadConcern of “snapshot”. With “snapshot” ReadConcern, all operations will take place against a consistent snapshot regardless of the number of distinct statements, network trips, or yield points . If the session the transaction is in is causally consistent, it’ll also be a snapshot that’s consistent with the previous operation transaction in the session.

Local Snapshot Reads and Transactions: Local Snapshot Reads offer transactions the ability to work from a point in time across the cluster. This means it can perform multi-document operations with an expectation that the snapshot time isn’t moving forward and is consistent.

The Global Logical Clock

Time synchronization within MongoDB has been enhanced by a move to a global logical clock. Implemented as a hybrid logical clock and protected by encryption, the global logical time now spans an entire sharded cluster. A consistent global view of time is essential for multi-document ACID transactions as it allows for global snapshots to be taken.

Background: MongoDB’s clock is tracked in the oplog of the primary in a MongoDB shard. Each operation, that is everything that creates a durable change in the data, is recorded in the oplog. As each operation is recorded, it ‘ticks’ the logical clock for the primary and the clock “time” is recorded with the operation in the oplog. This provides an ordering to the oplog which can be used to synchronize all nodes within the shard.

Hybrid Logical Clocks: While shards may have their own clock, that doesn’t work for the entire cluster; the clocks will have no common reference and increment on their own time. Enter the hybrid logical clock, as implemented in MongoDB 3.6.

This hybrid clock combines the system time and the counter for operations that happen at the same physical time to create a hybrid timestamp. This new timestamp is gossiped (or spread) by all connections that are established throughout the cluster. When a message with a timestamp is received by a server, if the timestamp is after that node's current timestamp, then it updates their latest seen timestamp with the new value. The latest time seen is immediately incremented when a new modification occurs .

Tamper Prevention: There is always the possibility of an attacker injecting a bad time and count into the system such that it would "run out of time" to increment into for new operations, thus the shard would become unable to accept any new modifications. This issue is mitigated by creating a hash of the timestamp using the primary's own private key. This hash is used to verify the source of the timestamp is within the zone of trust formed by the cluster; if an attacker can get their hands on the private key, then there's a much broader compromise of the system at play.

Global Logical Clocks and Transactions: With a hybrid logical clock, it becomes simpler to coordinate and synchronize multiple nodes in a replica set and across multiple shards in a cluster. As the scope of transactions expands across multiple shards in MongoDB 4.2, having a reliable, synchronized clock that works with that scope is an essential foundation stone. Ensuring the clock cannot be tampered with makes that foundation more reliable.

Safe Secondary Reads

Safe Secondary Reads address a problem that can occur when chunks of documents, or documents within a shard key range, are being migrated between shards. It ensures that not only the primaries of the shards involved in the migration are aware that chunks are in transit or have moved, but also the secondaries. This project made MongoDB's dynamic and automatic balancing, easy to leverage.

Background: Before the introduction of Safe Secondary Reads, only the primaries of the shards held a routing table which detailed which chunks of data were owned by which shards. When a query came into the primary, this routing table would be used to filter out documents that were in the middle of a migration to avoid duplicated results for a query sent to multiple shards.

When chunks of data were in the process of migrating between shards – typically as part of a rebalancing process to evenly distribute data across the cluster – the process would be to copy the chunk from the source shard to destination shard, update the routing table on both to show the chunk's new home, and then delete the chunk from the source shard.

While this is effective for queries routing to the primary in a shard, queries on the secondaries in the shards do not know the migration is happening and return the results they have. For a period of time, while a migration is in progress, there's a copy of the data on both shards' secondaries and so a secondary read could return both copies from either shard.

Replicating Routing: The solution to the problem, in its simplest form, was to replicate the routing table in the shard primary to the secondary nodes in the shard. As the routing table changes during the migration, it is then replicated to the secondary nodes. This makes the secondary node effectively migration aware and able to correctly route queries addressed directly to them. To leverage this routing information with secondary reads, send ReadConcern "local" or "majority" with your queries.

Safe Secondary Reads and Transactions: In combination with other features such as unified timestamps and the global logical clock, safe secondary reads allows transactions to work with a more predictable state.

Retryable Writes

By allowing a driver to retry the exact same write safely, without the danger of the server re-applying the write operation, we make it possible for clients to operate more reliably, while reducing the amount of code needed to implement error handling.

Background: When a driver sends a write to a server and an error is raised, it is not always possible to tell if that error came as the write went to the server or as the acknowledgment was being returned. That means it's also

not possible to tell if the server acted on the write command or not.

Consider, for example, a write which will increment a field. The client's driver could send that request to the server, the server act upon it, but before returning results to the client the connection is lost. On the server, the field has been incremented but the client's driver has no indication of this. If it sends the write again assuming the connection had failed before it reached the server, then the field would be incremented again. That is an assumption that is bound to fail to be true though. The client itself could try and work out, from the database state, if the write had been applied but that too is fraught with numerous ways to fail.

Retryable Writes: On The Client: Enter the retryable write; this is a write which can be sent multiple times and yet only be applied exactly once. This is implemented at the driver level, by using a connection setting of retryWrites, and requires the write operation to be at least acknowledged (writeConcern > 0). There are a specific set of operations which can use retryable writes. Since MongoDB 4.0, the transaction commit and abort operations have also been retryable, but they retry automatically rather than being specifically enabled in the driver. In all cases, there will only be one attempt to retry the write. Drivers will not keep retrying and therefore the retryable write is specifically good for transient network errors or primary elections, rather than persistent connection issues. A write will be sent to the server with a logical session id , and a transaction id.

Retryable Writes: On The Primary: When the write arrives at the server, it is now checked against the transaction table. This is a list of logical sessions ids, last seen transaction ids and pointers into the oplog where that operation was recorded. If there's a match to the logical session id and transaction id, then the logic of retryable writes says this is a retry. A result is compiled that the previous successful write operation would have generated and its this result that is sent back to the client as an acknowledgement.

If there isn't a match in the transaction table, then the write operation is presumed new. It is applied and an entry appears in the oplog. The transaction table is them updated against that session id with the transaction id and a pointer to that new oplog entry. And then the acknowledgement is returned to the client.

Retryable Writes: On The Secondary: Secondary nodes do not handle writes directly from the application until there's a failover and one of them then takes over as primary. If that happens without changes, the secondary has an empty transaction table and at that point it can't tell if an operation has previously happened. The solution to this is to replicate the transaction table. Unlike other replication operations, this is integrated into each write recorded in the oplog. Each gets the logical session id and transaction id sufficient that the secondary can build its own transaction table, correct right up to the particular oplog entry.

Then, when there is a primary failover and the secondary steps in, there is a complete transaction table too as well as a replica of the data, ready to go.

Retryable Writes and Transactions: Retryable writes are a foundation for the same retry mechanism which transactions use. As noted, the commit and abort operations are both retryable, but the retry is for the entire transaction as a single operation, unlike the retryable writes previously mentioned. This provides MongoDB with a way of eliminating transient network faults or primary elections from preventing transactions being committed committing of transactions.

Transactions and Their Impact to Data Modeling in MongoDB

Adding transactions does not make MongoDB a relational database – many developers have already experienced that the document model is superior to the tabular model used by relational databases.

All best practices relating to MongoDB data modeling continue to apply when using features such as multi-document transactions, or fully expressive JOINs (via the $lookup aggregation pipeline stage). Where practical, all data relating to an entity should be stored in a single, rich document structure. Just moving data structured for relational tables into MongoDB will not allow users to take

advantage of MongoDB's natural, fast, and flexible document model, or its distributed systems architecture.

The RDBMS to MongoDB Migration Guide describes the best practices for moving an application from a relational database to MongoDB.

Conclusion

MongoDB has already established itself as the leading database for modern applications. The document data model is rich, natural, and flexible, with documents accessed by idiomatic drivers, enabling developers to build apps 2-3x faster. Its distributed systems architecture enables you to handle more data, place it where users need it, and maintain always-on availability. MongoDB's existing atomic single-document operations provide transaction semantics that meet the data integrity needs of the majority of applications. The addition of multi-document ACID transactions in MongoDB makes it easier than ever for developers to address a complete range of us cases, while for many, simply knowing that they are available will provide critical peace of mind.

Take a look at our multi-document transactions web page where you can hear directly from the MongoDB engineers who have built transactions, review code snippets, and access key resources to get started. You can get started with MongoDB transactions now by spinning up your own fully managed, on-demand MongoDB Atlas cluster, or downloading it to run on your own infrastructure.

> *"Transactional guarantees have been a critical feature for relational databases for decades, but have typically been absent from non-relational alternatives, which has meant that users have been forced to choose between transactions and the flexibility and versatility that non-relational databases offer. With its support for multi-document ACID transactions, MongoDB is built for customers that want to have their cake and eat it too."*

Stephen O'Grady, Principal Analyst, Redmonk

We Can Help

We are the company that builds and runs MongoDB. Over 17,000 organizations rely on our commercial products. We offer cloud services and software to make your life easier:

MongoDB Atlas is the global cloud database service for modern applications. Deploy fully managed MongoDB across AWS, Azure, or Google Cloud with best-in-class automation and proven practices that guarantee availability, scalability, and compliance with security standards.

MongoDB Enterprise Advanced is the best way to run MongoDB on your own infrastructure. It's a finely-tuned package of advanced software, support, certifications, and other services designed for the way you do business.

MongoDB Atlas Data Lake allows you to quickly and easily query data in any format on Amazon S3 using the MongoDB Query Language and tools. You don't have to move data anywhere, you can work with complex data immediately in its native form, and with its fully-managed, serverless architecture, you control costs and remove the operational burden.

MongoDB Charts is the best way to create, share and embed visualizations of MongoDB data. Build visualizations quickly and easily to analyze complex, nested data. Embed individual charts into any web application or assemble them into live dashboards for sharing.

Realm Mobile Database allows developers to store data locally on iOS and Android devices using a rich data model that's intuitive to them. Combined with the MongoDB Realm sync-to-Atlas, Realm makes it simple to build reactive, reliable apps that work even when users are offline.

MongoDB Realm allows developers to validate and build key features quickly. Application development services like Realm Sync for mobile and Realm's GraphQL service, can be used with Realm Functions, Triggers, and Data Access Rules – simplifying the code required to build secure and performant apps.

MongoDB Cloud Manager is a cloud-based tool that helps you manage MongoDB on your own infrastructure. With automated provisioning, fine-grained monitoring, and

continuous backups, you get a full management suite that reduces operational overhead, while maintaining full control over your databases.

MongoDB Consulting packages get you to production faster, help you tune performance in production, help you scale, and free you up to focus on your next release.

MongoDB Training helps you become a MongoDB expert, from design to operating mission-critical systems at scale. Whether you're a developer, DBA, or architect, we can make you better at MongoDB.

Resources

For more information, please visit mongodb.com or contact us at sales@mongodb.com.

Case Studies (mongodb.com/customers)
Presentations (mongodb.com/presentations)
Free Online Training (university.mongodb.com)
Webinars and Events (mongodb.com/events)
Documentation (docs.mongodb.com)
MongoDB Atlas database as a service for MongoDB (mongodb.com/cloud)
MongoDB Enterprise Download (mongodb.com/download)
MongoDB Stitch Serverless Platform (mongodb.com/cloud/stitch)
MongoDB Realm (mongodb.com/realm)

mongoDB

US 866-237-8815 ▪ INTL +1-650-440-4474 ▪ info@mongodb.com