MongoDB 3.6.4

Kit Patella

2018-10-23

In February 2017, we discussed data loss and fixes in MongoDB 3.4.0-rc3's v0 and v1 replication protocols. In this Jepsen report, we will verify that MongoDB 3.6.4's sharded clusters offer comparable safety to non-sharded deployments. We'll also discuss MongoDB's new support for causal consistency (CC) in version 3.6.4 and 4.0.0-rc1, and show that sessions prevent anomalies so long as user stick to majority reads and writes. However, with MongoDB's default consistency levels, CC sessions fail to provide the claimed invariants. This work was funded by MongoDB, and conducted in accordance with the Jepsen ethics policy.

1 Background

MongoDB is a long-time user of Jepsen. Over the past three years, Jepsen has performed several analyses for MongoDB, and MongoDB has integrated an expansive Jepsen test suite into their CI system. In March, MongoDB requested Jepsen perform an analysis on a previously untested configuration: sharded clusters. We also pursued new research into modeling and verifying causal consistency, a new safety feature in MongoDB 3.6. First, we will verify that sharded clusters offer comparable safety to individual replica sets in version 3.6.4, and then we'll discuss the anomalies we uncovered in versions 3.6.4 and 4.0.0-rc1 with our new causal consistency tests.

2 Sharded Clusters

Sharded clusters split a collection of documents into parts, called *shards*, based on a single field in each document: the *shard key*. Each shard is stored on an independent MongoDB replica set. A router process, called `mongos`, routes client requests to the appropriate replica set. A dedicated MongoDB replica set, called `configsvr`, maintains the cluster state including the mapping of shards to replica sets.

Since the size of shards may change over time, MongoDB further divides shards into *chunks*. If a shard grows too large, a balancer process, driven by the config server, can produce a more even distribution by splitting and moving chunks to other shards.

2.1 Methods

We repeated our tests from prior MongoDB analyses with sharded clusters. Our cas-register test verifies that single documents support linearizable reads, writes, and compare-and-set operations. Because this test is computationally expensive, we supplement it with a set test, which verifies that compare-and-set operations are sequentially consistent by adding elements to a single document, and verifying that all acknowledged elements are present in a final read. This test is faster and cheaper to verify, because, we just care that each op shows up in final read. This allows us to check more events and catch rare cases of lost updates.

We set up a cluster of 5 nodes with a configurable number of routers and shards, each shard being a replica set. Each node runs one mongod `configsvr` process, a `mongod` process per shard, and at most one `mongos` router to accept requests. The number of routers in the cluster can be limited as well.

Sharded clusters have the added complexity of chunk migrations, which offers significant potential for data loss if mishandled. We therefore designed a balancer nemesis, which combines network partitions and `moveChunk` commands to move acknowledged data under adverse conditions. We choose chunks containing recently written-to document IDs, move them to new, randomly selected shards, then immediately partition the network and heal it some time later.

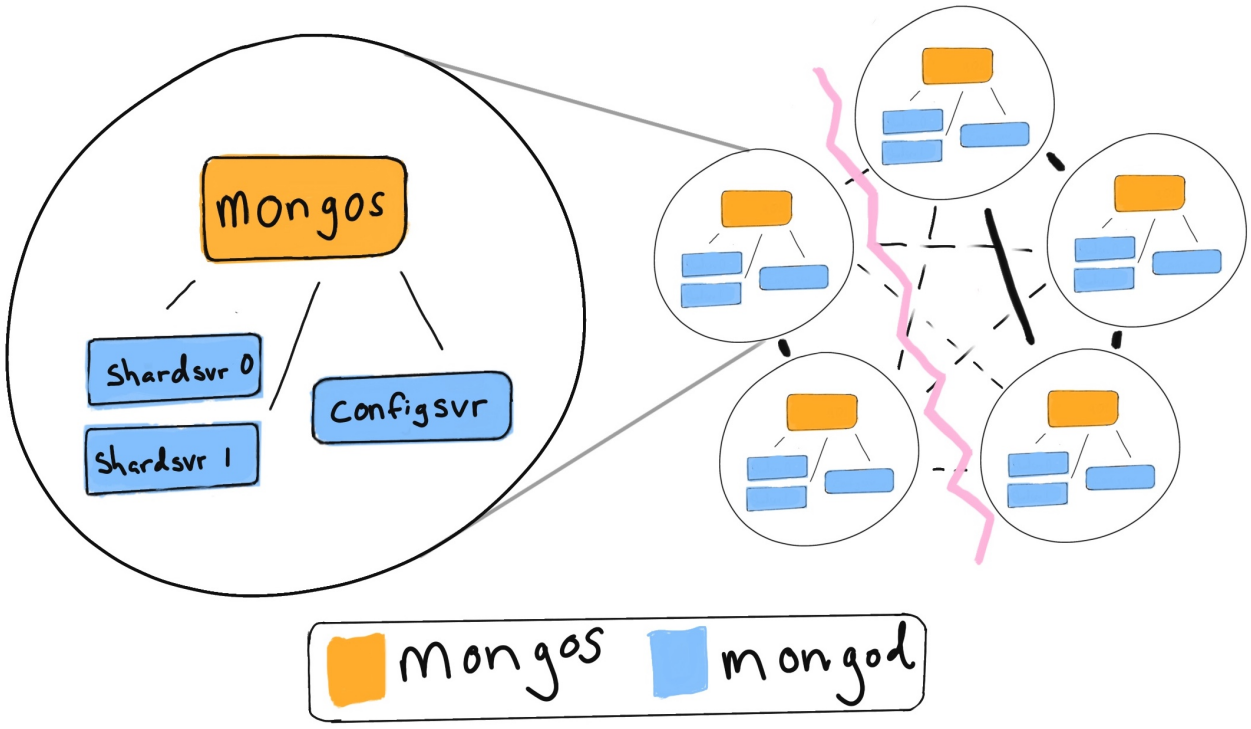

Figure 1: Topology of a sharded cluster with two shards, including a majority-halves partition splitting nodes into two isolated groups: a majority component with three nodes, and a minority component with two.

2.2 Results

We know that MongoDB can lose committed writes for all levels of write concern less than `majority`, since those writes could be rolled back by newer primaries, regardless of whether the `journaled` flag is enabled. This is a documented behavior for MongoDB replica sets, and sharded clusters are no different. In this set test against a sharded cluster, with write concern `journaled`, MongoDB lost 543 out of 6095 successfully acknowledged writes.

Lost	Recovered	OK
181/2032	1/3048	229/254

This result is unsurprising, as sharded clusters only affect where data is located not the replication protocol for the data within a shard.

What about `majority` writes and `linearizable` reads? After weeks of testing both insert-only and update-heavy workloads against sharded clusters, we've found that MongoDB's v1 replication protocol appears to provide linearizable single-document reads, writes, and compare-and-set, through shard rebalances and network partitions.

Running fewer `mongos` routers than nodes in the test lowers the throughput of the test, possibly masking uncommon concurrency errors. However, we do not have any evidence that `mongos` introduces linearizability violations.

3 Causal Consistency

Causal consistency (CC) is a consistency model for distributed databases, which guarantees that operations that are causally related are always observed in the same order. For instance, replies to a question should appear only if the question is also visible, never on their own. In this model, operations that have no dependency relationship are said to be concurrent, and concurrent operations may have no apparent fixed order.

Causal is one of many middle grounds between eventually consistent and linearizable systems. In eventually consistent systems, operations can be observed in any order so long as they eventually converge. And in linearizable systems, operations must appear to occur in the same order to every single observer, with hard real-time bounds. Causal consistency allows clients to only wait for a subset of dependent operations rather than

waiting for a total order of all operations. This is especially useful when a total order is too expensive or impossible to provide and allows implementations to offer improved availability.

Thus far, causal consistency has generally been limited to research projects, like COPS, Bolt-on Causal Consistency, and AntidoteDB; MongoDB is one of the first commercial databases we know of which provides an implementation.

So how does MongoDB approach causal consistency? If we identify a MongoDB collection as a set of read-write registers, there are four guarantees that MongoDB claims for causal consistency[1]:

> *Read your writes:* Read operations reflect the results of write operations that precede them.
>
> *Monotonic reads:* Read operations do not return results that correspond to an earlier state of the data than a preceding read operation.
>
> *Monotonic writes:* Write operations that must precede other writes are executed before those other writes.
>
> *Writes follow reads:* Write operations that must occur after read operations are executed after those read operations.

MongoDB clients capture causality with the concept of a *session*: a single-threaded context in which each database operation is causally subsequent to the previous operation. Sessions exist alongside client connections, and are associated with a single client. Sessions are passed to the call site with each read and write, providing the server with the highest server times that the client has seen.

Replica sets use a log of operations (the *oplog*) in which each operation is identified by an *optime*. Optimes are a tuple of an *election ID* and a *timestamp*, which uniquely identify every operation. Nodes advance their timestamps monotonically to match the local wall clock, or, like a Lamport clock, the highest observed timestamp from any other node.[2]

Sessions use the timestamp to provide a monotonic ordering relation for operations. When a session asks a server to perform an operation, it includes the last timestamp that session observed; the server must wait until that timestamp has been reached to service the request. This holds even between different shards: all nodes across all replica sets essentially share a single timestamp, whereas election IDs are only meaningful within individual replica sets.

This provides monotonicity because once an operation is majority committed to the oplog, no subsequent operation can be majority committed with a lower timestamp. Likewise, when a majority read is performed with a certain timestamp, that result can never be unobserved by a majority reader with an equal or higher timestamp.

Sessions expose their timestamp as a causal token[3]. That token can be used to force one session to observe the results of another. That is, users can store and pass tokens to other clients, even other nodes, to preserve causal ordering.

MongoDB's sessions are a special case of causal consistency: sessions capture a linear notion of causality in which operations are totally ordered. In general, the causal relationships between operations could be an arbitrary directed acyclic graph (DAG). But in the limiting case where every session performs only a single operation, timestamps can be manually threaded from session to session to construct arbitrary DAGs.

3.1 Methods

We implemented a new kind of Jepsen test to check causal consistency, adapted from Bouajjani, Enea, Guerraoui, and Hamza's techniques On Verifying Causal Consistency. We model the relationship between operations as a causal order (CO), which represents the set of operations visible to each operation we perform. We use independent keys and a single session for each, which should produce a total order per key.

We perform five operations with our client linearly for each CO: an initial read, a write of 1, a read, a write of 2, and a final read. These operations are represented as `:read-init`, `:write`, `:read`, `:write`, and `:read` in our histories. We expect to see nothing in the initial read and mark this nil response as 0 in the history. Then we execute our writes and reads, with each op-

[1] MongoDB's explanation of these properties makes some implicit assumptions about dependencies and causal scope, which may be a bit confusing. For a more thorough discussion of these properties, consider Viotti & Vukolić's formulation of read your writes, monotonic reads, monotonic writes, and writes follow reads.

[2] See also: https://cse.buffalo.edu/tech-reports/2014-04.pdf

[3] These timestamps are cryptographically signed to prevent malicious clients from pushing nodes too far into the causal future, effectively preventing them from servicing any further operations.

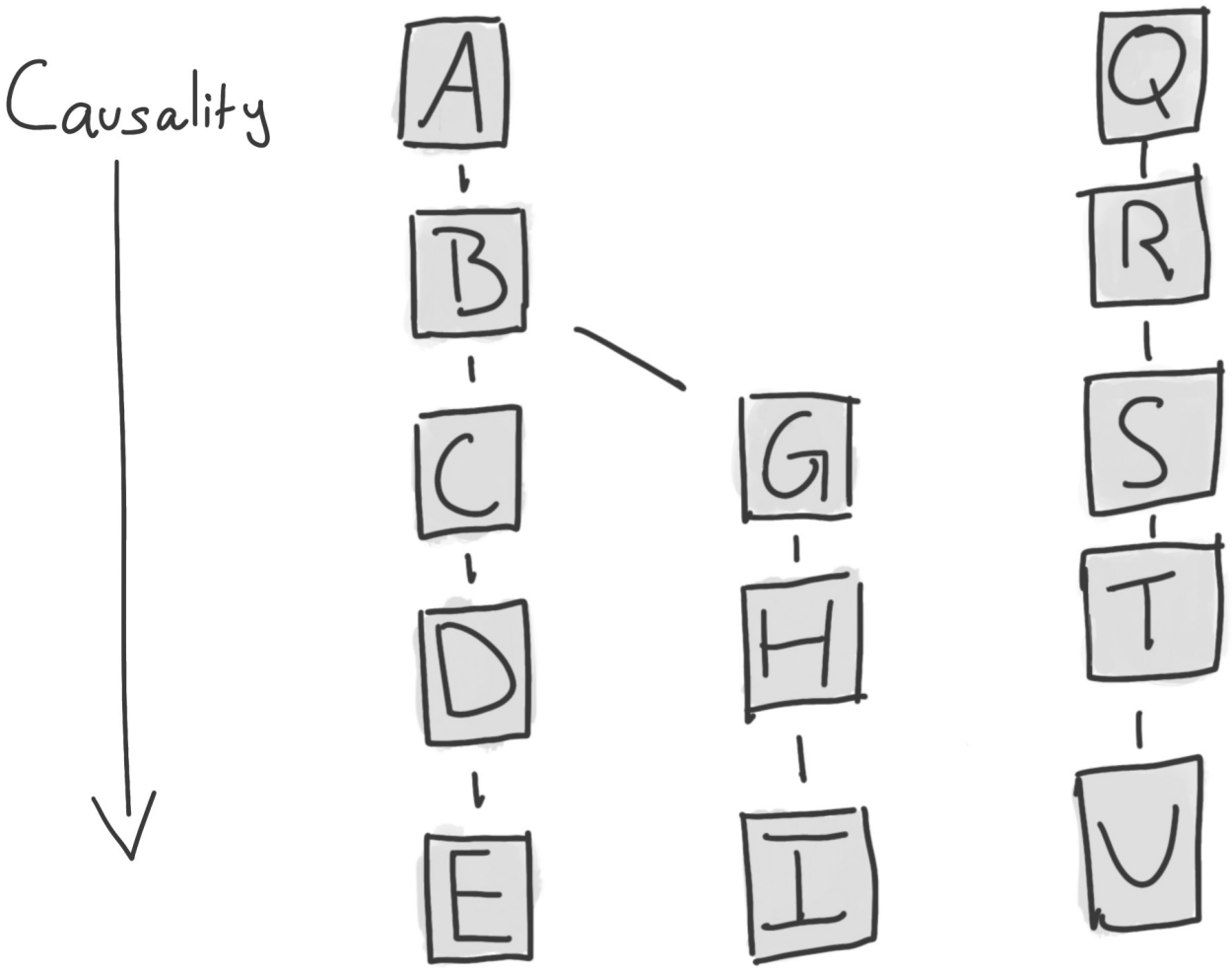

Figure 2: Two concurrent sets of causally related operations. On the left is a graph that forks, and on the right is a linear chain. Note that {G,H,I} is not causally related to {C,D,E}, despite both depending on {A,B}; the two forks are concurrent.

eration depending on the prior one. Our checker expects the register to advance through the states [0, 1, 2]. That is, our history should be: `:read-init 0, :write 1, :read 1, :write 2, :read 2` in every CO. Anomalies become apparent when a CO does not reflect the order that we executed (e.g. [0, 2, 1]) or if dependent operations are rolled back (e.g. [0, 1, 0]). Lastly, if `:read-init` observes a non-zero write (e.g. [n, 1, 2] where n ≠ 0), then we cannot validate the causal order.

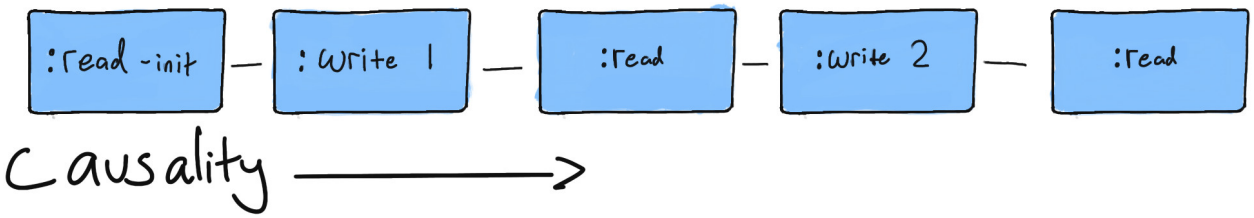

Figure 3: Jepsen operations in a causal order.

3.2 Results

We uncovered clear evidence of causal consistency violations. So far, we've observed two types of anomalies.[4]

Process	Function	Value
nemesis	:start	[:isolated {"n5" #{"n2" ...}, ...}]
0	:read-init	0
0	:write	1
0	:read	1
0	:write	2
:nemesis	:stop	:network-healed
0	:read	0

In this example, we perform two writes which MongoDB acknowledges, during a network partition. Once the partition heals, despite confirming the writes, our final read returns the initial, blank state of the register—as if those writes *had never taken place*.

Process	Function	Value
0	:read-init	0
0	:write	1
0	:read	1
0	:write	2
:nemesis	:start	[:isolated {"n5" #{"n2" ...} ...}]
:nemesis	:stop	:network-healed
0	:read	1

The last read executes during a network partition, and returns after the partition heals. We can see from the final read that the value of the second write appears to have been rolled back, rather than being cached for the client.[5]

In this case, both COs were executed with write concern `w1`, and read level `local`. The first anomaly, where reads observe the uninitialized state of the CO, disappears with read level `majority`. We did not observe any anomalies with write concern `majority`.[6] MongoDB version 4.0.0-rc1 displayed comparable behavior. This issue is further described in SERVER-35316.

4 Discussion

MongoDB and Jepsen have an established history of public analyses and internal tests. This also makes finding new bugs difficult. We have, by now, picked much of the low-hanging fruit. Sharding has known issues, but so far we haven't uncovered any new ones.

We did, however, uncover problems with MongoDB's causal consistency: it doesn't work unless users use both read and write concern `majority`, and the causal consistency documentation made no mention of this. While MongoDB will reject causal requests with the safer `linearizable` read level, and the unsafe write concern `unacknowledged`, it will happily accept intermediate levels, like write concern 2 or read level `local`. Since many users use sub-majority operations for performance reasons, and since causal consistency is typically used for high-performance local operations which

[4] The first history is from key 73, and the second is from key 158.

[5] The example histories above have been truncated for ease of understanding. Full Jepsen histories list both the invocation and completion of client operations. Type `:invoke` marks the start of an operation, and `:ok`, `:fail`, or `:info` types denote the result (or non-termination) of that operation. Here, we show only ops for `:ok` responses and the `:info` ops recording the nemesis' state transitions.

[6] Jepsen and MongoDB suspect that write concern `majority` anomalies should exist, see https://jira.mongodb.org/browse/SERVER-35316.

do not require coordination with other cluster nodes, users could have reasonably assumed that causal sessions would ensure causal safety for their sub-majority operations; they do not.

Mongo has since added numerous warnings to the consistency docs advising users that its guarantees apply only to `majority`/`majority` operations, which should help guide users toward using the feature correctly.

Even with causal sessions, sub-majority reads may fail to observe causally prior successful writes, or fail to observe previously read values, even with `majority` write concern. Conversely, sub-majority writes may be visible, then lost in the event of a leader transition, which means that successfully acknowledged prior writes may not be observed by subsequent reads. We interpret this as a violation of causal consistency.

MongoDB has closed this issue as working as designed, arguing that with sub-majority writes and `majority` reads, sessions actually do preserve causal consistency:

> Even with write concerns less than majority, the causal ordering of the committed writes is maintained. However, durability is not guaranteed.

This interpretation hinges on interpreting successful sub-majority writes as *not necessarily successful*: rather, a successful response is merely a suggestion that the write has *probably* occurred, or might later occur, or perhaps will occur, be visible to some clients, then un-occur, or perhaps nothing will happen whatsoever.[7]

We note that this remains MongoDB's default level of write safety.

If one considers every successful sub-majority write as indeterminate[8], this interpretation is defensible: majority reads will tell you whether or not a prior write succeeded[9] and observe logically monotonic states of the system. Writes are monotonic if observed by majority reads, and so on. This could offer performance advantages where users are unable to make concurrent calls and wish to batch together several writes, all of which will be read later to confirm their success, and no concurrent operation will interfere with those records. However, Jepsen believes users will find this interpretation less intuitive.

Jepsen continues to recommend `majority` writes in all cases, and `majority` reads where `linearizable` is prohibitively expensive. Anything less than `majority` writes can lose data, and anything less than `majority` reads can read dirty data. MongoDB has discussed making servers reject write concerns and read levels below `majority` when using CC sessions, which might help. We recommended MongoDB update their documentation so users are aware of the requirements for using causal consistency, which was completed in September 2018.

We typically choose causally consistent systems because they can be made totally available: even when the network is completely down, every node can independently make progress. In fact, slightly stronger models like causal+ and real-time causal (RTC) are proven to be the among the strongest consistency models achievable in totally available contexts. However, MongoDB's replication architecture is currently incompatible with a totally-available approach. Only leaders can write, so some nodes must refuse some operations when the leader is inaccessible, and refuse all writes when no leader can reach a majority.

So why use MongoDB's causally consistent sessions? Because they offer stronger safety properties than majority/majority alone. For instance, majority/majority allows you to observe a write, then un-observe it. Or you can write something, then try to query it, and it won't be there. It'll show up… eventually! Causal eliminates these anomalies by forcing logical monotonicity. So, if you're a user of MongoDB with write concern `majority` and read level `majority`, we recommend using CC sessions in your applications. Furthermore, users who need to ensure the order of operations between different clients should consider passing causal timestamps between those clients via e.g. side channels.

MongoDB uses monotonic timestamps derived from wall clocks, messages from other servers, and mes-

[7] In a sense, `majority` writes are *also* "best effort" delivery. For example, if a majority of nodes lost their disks concurrently, some majority-committed writes could be lost! However, this failure mode is *detectable*; the system will fail to make progress when it occurs, and operators *know* that data may have been lost. By contrast, write concerns less than `majority` could result in *silent* data loss. Silent write loss can occur even with majority writes. (e.g. byzantine faults, like malicious nodes, or concurrent crashes followed by filesystem-level data loss which reverts a majority of nodes to an earlier consistent snapshot of the log) However, we believe these failures are significantly less likely than leader elections.

[8] This behavior is similar to what happens when a client experiences a network timeout: operations may or may not occur. To impress upon users the fundamental uncertainty involved, perhaps MongoDB clients could throw network timeout errors instead of returning successfully when servers acknowledge a sub-majority write.

[9] Of course, if someone else updates a record between a session's write and read, it may be impossible to ascertain whether the write actually took place.

sages from causal-enabled clients, as the link between dependent operations. Two leaders with different election IDs might have similar, locally monotonic, timestamps. So causal sessions based on timestamps alone could perform sub-majority operations on two independently-evolving leaders while still observing a monotonic timestamp order. That's why we observe anomalies with sub-majority operations: the causal structure of MongoDB operations isn't (in general) captured by timestamps alone.

Because MongoDB's causal model requires majority writes and majority reads, clients essentially pay the latency and availability costs required for sequential consistency, which forces a *total* order, rather than the partial order required by causal. We believe that MongoDB's causal sessions might actually provide sequential consistency on individual keys. Due to limitations in sharding, it's unclear if this extends to multiple keys.

Finally, there are significant limitations to our tests. Our sharded tests assume a relatively uniform cluster topology, where all MongoDB components partition in the same way. Non-homogeneous topologies where we can partition `configsvr` and `mongos` processes separately from `shardsvr` processes may find unique anomalies. Starting and stopping and killing various component processes may also provide interesting results.

Our causal consistency test only measures a very simple case: we test short time frames, on single keys, against single nodes, from single client threads. We suspect that writes to multiple nodes might be required to observe causal violations with majority writes and sub-majority reads. We also don't check how failures and process crashes influence causal orders. Ultimately, there's still a lot we don't know!

This work was funded by MongoDB, and conducted in accordance with the Jepsen ethics policy. My thanks to Kyle Kingsbury for his invaluable contributions and review, and to Christopher Meiklejohn, Peter Alvaro, and the MongoDB team, especially Cristopher Stauffer, Max Hirschhorn, Dan Pasette, and Misha Tyulenev.