JEPSEN

MongoDB 4.2.6

Kyle Kingsbury

2020-05-15

MongoDB is a distributed document database which claims to offer "among the strongest data consistency, correctness, and safety guarantees of any database available today", with "full ACID transactions". Jepsen evaluated MongoDB version 4.2.6, and found that even at the strongest levels of read and write concern, it failed to preserve snapshot isolation. Instead, Jepsen observed read skew, cyclic information flow, duplicate writes, and internal consistency violations. Weak defaults meant that transactions could lose writes and allow dirty reads, even downgrading requested safety levels at the database and collection level. Moreover, the `snapshot` read concern did not guarantee snapshot unless paired with write concern `majority`—even for read-only transactions. These design choices complicate the safe use of MongoDB transactions. This work was performed independently, without compensation, and conducted in accordance with the Jepsen ethics policy. MongoDB, Fauna, and YugaByte, all mentioned in this report, have previously engaged Jepsen for paid analyses.

1 Updates

2020-05-26: MongoDB identified a bug in the transaction retry mechanism which they believe was responsible for the anomalies observed in this report; a patch is scheduled for 4.2.8.

2 Background

MongoDB is a popular distributed document database. It offers replication via a homegrown consensus protocol which draws inspiration from Raft, and can distribute data across shards via mongos. We previously evaluated MongoDB at versions 2.4.3, 2.6.7, 3.4.0-rc3, and 3.6.4.

Our most recent report on MongoDB 3.6.4 focused on causal consistency and linearizability in sharded collections. We found that sharded clusters appeared to offer linearizable reads, writes, and compare-and-set operations against single documents, so long as users ran with read concern `linearizable` and write concern `majority`. However, any weaker level of write concern resulted in the loss of committed writes. MongoDB's default level of write concern was (and remains) acknowledgement by a single node, which means MongoDB may lose data by default. Although the write concern documentation does not make this clear, the rollback documentation states:

> With the default write concern, data may be rolled back if the primary steps down before the write operations have replicated to any of the secondaries.

Similarly, MongoDB's default level of read concern allows aborted reads: readers can observe state that is not fully committed, and could be discarded in the future. As the read isolation consistency docs note, "Read uncommitted is the default isolation level".

We found that due to these weak defaults, MongoDB's causal sessions did not preserve causal consistency by default: users needed to specify both write and read concern `majority` (or higher) to actually get causal consistency. MongoDB closed the issue, saying it was working as designed, and updated their isolation documentation to note that even though MongoDB offers "causal consistency in client sessions", that guarantee does not hold unless users take care to use both read and write concern `majority`. A detailed table now shows the properties offered by weaker read and write concerns.

Curiously, MongoDB omitted any mention of these findings in their MongoDB and Jepsen page. Instead, that page discusses only passing results, makes no mention of read or write concern, buries the actual report in a footnote, and goes on to claim:

> MongoDB offers among the strongest data

> consistency, correctness, and safety guarantees of any database available today.

We encourage MongoDB to report Jepsen findings in context: while MongoDB *did* appear to offer per-document linearizability and causal consistency with the strongest settings, it also *failed* to offer those properties in most configurations. We think users might want to be aware that their database could lose data by default, but MongoDB's summary of our work omits any mention of this behavior.

2.1 Transactional Consistency

So, does MongoDB offer "among the strongest data consistency, correctness, and safety guarantees"? Past work suggests that for individual document operations, the answer is, "yes; MongoDB offers per-document linearizability with the strongest settings, but not by default". However in 2018, MongoDB introduced *multi-document transactions*—limited to within a shard—and in 2019, extended those transactions across shards. What safety properties do these transactions provide?

The MongoDB home page proudly advertises "full ACID transactions". The transactions page states that MongoDB is "the only database that fully combines the power of the document model and a distributed systems architecture with ACID guarantees," a combination also claimed by CosmosDB, DynamoDB, FaunaDB, Oracle NoSQL, OrientDB, RavenDB, SAP HANA, YugaByte DB, et al.

The MongoDB architecture guide promises ACID transactions "maintain the same data integrity guarantees you are used to in traditional databases…", and that MongoDB offers "strong consistency by design", but offers no more specific claims. The ACID whitepaper clarifies that MongoDB transactions offer snapshot isolation: a reasonably strong model which constitutes the baseline level of consistency for systems like PostgreSQL. The whitepaper states:

> … 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.

MongoDB repeatedly summarizes snapshot isolation as "transactions provide a consistent view of data, and enforce all-or-nothing execution to maintain data integrity". This is a concise and intuitive summary of snapshot isolation, but we should note that a "consistent view" under snapshot isolation may still be surprising: as Fekete, O'Neil, and O'Neil wrote in 2004, read-only transactions can observe nonserializable behavior under snapshot isolation. Nor does snapshot isolation necessarily maintain data integrity: in their 1995 paper defining snapshot isolation, Berenson, Bernstein, et al. provided examples of applications whose integrity constraints could be violated under snapshot isolation—for instance, due to write skew.

3 Test Design

We designed a test suite using the Jepsen distributed systems testing library, and used it to evaluate transactional safety in MongoDB 4.2.6. Our tests installed MongoDB's official Debian packages on clusters of nine Debian 9 nodes. We tested both in LXC and EC2; both exhibited similar behavior. As per the deployment guidelines, we built two-shard clusters, with both shards and the `configsvr` metadata system running as three-node replica sets. All nine nodes ran an instance of `mongos`, which serves as the frontend to sharded MongoDB clusters.

Our test workload involved transactions over a rotating pool of documents in a single MongoDB collection, each document containing a single array of integers. Each transaction performed one to four operations over those documents: either reading a single document by primary key `_id`, or appending (using `$push`) a unique integer to a single document's `value` array, again by `_id`. We sharded the collection by `_id`.

Using Elle, a new transaction analysis tool developed in collaboration with UC Santa Cruz's Peter Alvaro, Jepsen automatically inferred dependencies between these transactions, and searched for cycles in that graph to identify isolation anomalies. Additional techniques checked for aborted and intermediate reads, as well as other non-cyclic anomalies.

During some of these tests, we introduced network partitions, targeted to isolate MongoDB primary nodes.

4 Results

4.1 Sometimes, Programs That Use Transactions… Are Worse

We began by running our tests without transactions to get a baseline: each Jepsen "transaction" performed only a single read or append operation without using the session or transaction APIs. Since we know MongoDB loses updates with any setting less than `majority`, and exhibits stale reads without read concern `linearizable`, we set write concern `majority` and read concern `linearizable` on the `client's database handle`. The resulting histories appeared consistent with snapshot isolation.

We then wrapped those single operations in transactions, and a surprising behavior appeared: with network partitions, transactions appeared to lose acknowledged writes. For instance, consider this history, where in 30 seconds, updates to eight documents were successfully acknowledged, then disappeared. Here is the history of reads for document 555:

```
21  :ok :txn    [[:r 555 [1]]]
23  :ok :txn    [[:r 555 [1]]]
14  :ok :txn    [[:r 555 [1]]]
6   :ok :txn    [[:r 555 [1]]]
5   :ok :txn    [[:r 555 [1]]]
21  :ok :txn    [[:r 555 [1 2]]]
0   :ok :txn    [[:r 555 [1 2 3 5 4 6 7]]]
6   :ok :txn    [[:r 555 [1 2 3 5 4 6 7]]]
19  :ok :txn    [[:r 555 []]]
0   :ok :txn    [[:r 555 [8]]]
```

Clients observed a monotonically growing list of elements until `[1 2 3 5 4 6 7]`, at which point the list reset to `[]`, and started afresh with `[8]`. This could be an example of MongoDB rollbacks, which is a fancy way of saying "data loss".

This is bad, but a more subtle question arises: *why were we able to read these values at all*? After all, read concern `linearizable` is supposed to show only majority-acknowledged (i.e. durable) writes. The answer is a surprising—but documented—MongoDB design choice:

> Operations in a transaction use the transaction-level read concern. That is, any read concern set at the collection and database level is ignored inside the transaction.

The received wisdom in the database community is that MongoDB has historically resisted raising the default level of read and write safety because doing so would impact production users who have become accustomed to faster, occasionally unsafe defaults, and because data loss might not be significant enough to warrant the increased latency, throughput, and capital expenditure entailed by stronger safety settings. MongoDB's researchers reported in VLDB that:

> … users would prefer, of course, to use readConcern level "majority" and writeConcern w:"majority", since everyone wants safety. However, when users find stronger consistency levels to be too slow, they will switch to using weaker consistency levels. These decisions are often based on business requirements and SLAs rather than granular developer needs. As we argue throughout this paper, the decision to use weaker consistency levels often works in practice because failovers are infrequent and data loss from failovers is usually small.

However, transactions are an entirely new feature, and users presumably *expect* to trade off some speed in exchange for improved safety guarantees. A reasonable user might expect that the default safety levels for transactions provide, as promised, snapshot isolation—or, at the very least, the same level of read concern that the user has already requested from the databases or collections involved. Instead, transactions without an explicit read concern downgrade any requested read concern at the database or collection level to a default level of `local`, which offers "no guarantee that the data has been written to a majority of replicas (i.e. may be rolled back)."

So: users should be careful to use the `snapshot` level of read concern with every transaction which requires snapshot isolation. The documentation confirms: "Read concern 'snapshot' returns data from a snapshot of majority committed data…" which makes sense, and then continues: "**if** the transaction commits with write concern 'majority'", which does not. Specifically:

> If the transaction does not use write concern "majority" for the commit, the "snapshot" read concern provides no guarantee that read operations used a snapshot of majority-committed data.

"What is the point," an astute reader might ask, "of having a `snapshot` read concern which does not provide snapshot isolated reads?" An even more astute reader, having observed a pattern, might inquire about the *default* level of write concern.

> If the transaction-level write concern and the session-level write concern are unset, transaction-level write concern defaults to the client-level write concern. By default, client-level write concern is w: 1.

In order to obtain snapshot isolation, users must be careful not only to set the read concern to `snapshot` for each transaction, but *also* to set write concern for each transaction to `majority`.

Astonishingly, this applies even to *read-only* transactions. In this test run, we set read concern `snapshot` and write concern `majority` on single-operation write transactions, and for single-operation read transactions, set only read concern `snapshot`. When a network partition occurred, reads observed divergent timelines:

```
11  :ok :txn    [[:r 77 []]]
12  :ok :txn    [[:r 77 [1 5]]]
12  :ok :txn    [[:r 77 [1 5 6]]]
23  :ok :txn    [[:r 77 []]]
4   :ok :txn    [[:r 77 [3]]]
13  :ok :txn    [[:r 77 [3 7]]]
```

These updates weren't *lost* exactly—the transactions which wrote 1, 5, and 6 timed out, but their effects being both visible and not visible to reads implies that at least one of these timelines was an instance of aborted read.

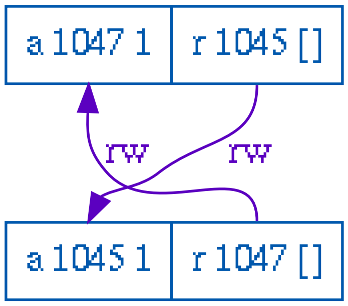
This behavior might be surprising, but to MongoDB's credit, most of this behavior is clearly laid out in the transactions documentation. The question is whether users are closely reading that documentation, versus relying on marketing claims, or assumptions like "using read concern `snapshot` means reading a committed snapshot". We might also ask whether users can be expected to remember to apply these settings to every transaction which requires them. After all, MongoDB offers database and collection-level safety settings precisely so users can *assume* all operations interacting with those databases or collections use those settings; ignoring read and write concern settings when users perform (presumably) safety-critical operations is surprising!

4.2 How ACID is Snapshot Isolation, Anyway

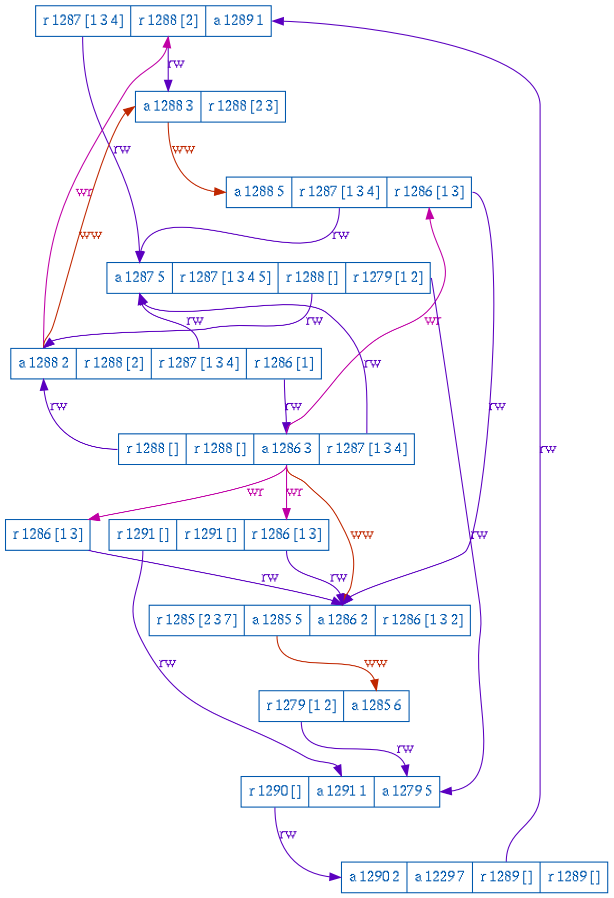
In subsequent tests, we used read concern `linearizable` and write concern `majority` for all single-operation reads and writes, and read concern `snapshot` and write concern `majority` for multi-operation transactions. In healthy clusters (e.g. without faults), cursory testing appeared consistent with snapshot isolation.

However, we note that while MongoDB's home page prominently claims to offer "full ACID transactions", which one might assume means that transactions are fully atomic, isolated, consistent, and durable. This is not exactly the case: transactions under snapshot isolation are not, as previously noted, fully isolated.

For example, consider this test run, which used read concern `snapshot`, write concern `majority`, and did not involve network partitions or other exogenous faults. The resulting history does not appear to violate snapshot isolation, but nonetheless exhibits cyclic transaction dependencies, like the following:

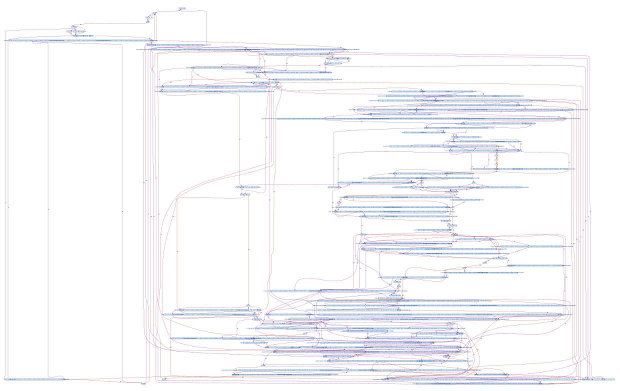


One transaction appends 1 to document 1047, and reads document 1045, finding it empty. The other appends 1 to document 1045, and reads 1047, finding it empty. Lines marked `rw` denote these read-write *anti-dependencies*. These transactions cannot possibly be isolated: if the top transaction executed first, in isolation, its write to 1047 would have been visible to the second—and vice-versa. Since these transactions didn't write to the same documents, they are allowed (under snapshot isolation) to execute concurrently.


r 1287 [1 3 4] | r 1288 [2] | a 1289 1
a 1288 3 | r 1288 [2 3]
a 1288 5 | r 1287 [1 3 4] | r 1286 [1 3]
a 1287 5 | r 1287 [1 3 4 5] | r 1288 [] | r 1279 [1 2]
a 1288 2 | r 1288 [2] | r 1287 [1 3 4] | r 1286 [1]
r 1288 [] | r 1288 [] | a 1286 3 | r 1287 [1 3 4]
r 1286 [1 3]
r 1291 [] | r 1291 [] | r 1286 [1 3]
r 1285 [2 3 7] | a 1285 5 | a 1286 2 | r 1286 [1 3 2]
r 1279 [1 2] | a 1285 6
r 1290 [] | a 1291 1 | a 1279 5
a 1290 2 | a 1229 7 | r 1289 [] | r 1289 []
rw
ww
wr

Other cycles are more complex. Here, for instance, is a cluster of twelve transactions, again executed under MongoDB's strongest possible safety settings. Each of these transactions depends (in various ways) on every other. Arrows labeled `ww` show a write-write dependency, wherein one transaction overwrote another's write. Those with `wr` show a write-read dependency: one transaction read another's write.

Or consider this cluster of 123 transactions, all of which appeared to execute before—but also after!—every other. Is this cluster "full ACID"? Perhaps, but if so, we must accept that the "I" in ACID means only *partial* isolation, or "full" means somewhat less than full.



These anomalies are not rare. This history, for example, contains roughly 100 transactions per second, and we identified 1461 transactions (out of 13914 total) with cyclic dependencies. Roughly 10% of transactions exhibited anomalies during normal operation, without faults.

It's important to remember that just because we *can* detect anomalies in this simple workload doesn't mean those anomalies *matter* to users. Concurrency could be low enough that concurrency control is largely unnecessary. Some sets of transactions can be proven to execute serializably under snapshot isolation—e.g., when their write sets intersect. Others exhibit nonserializable anomalies, but don't violate application-level consistency constraints. Still others *do* violate constraints, but not frequently enough for users to notice or care. For these purposes, snapshot isolation is good enough!

4.3 Indeterminate Errors

When we introduced network failures into our tests, we encountered (as one would expect) a variety of client errors. The MongoDB transaction documentation says:

> When a transaction aborts, all data changes made in the transaction are discarded without ever becoming visible. For example, if any operation in the transaction fails, the transaction aborts and all data changes made in the transaction are discarded without ever becoming visible.[1]

However, the converse is not necessarily true: some transaction error messages seem to indicate a transaction has aborted, but do not. For example, a `TransactionCoordinatorSteppingDown` exception may actually mean the transaction has committed. Likewise, `Command failed with error 6 (HostUnreachable): 'unable to initialize targeter for write op for collection jepsendb.jepsencoll :: caused by :: Connection refused'` also appears to denote an indeterminate failure. We repeatedly encountered these errors in our tests, only to find that their writes were visible to later reads.

This is not necessarily a *bug*—there will always be a class of errors in any distributed system which could indicate either success or failure. However, it *is* helpful when those errors are clearly marked, or offer textual guidance. Error 6, like most of the errors we encountered, is undocumented; only a few codes remain in the official docs, and Google has little to say. Complete error documentation, perhaps with a table of which codes indicate determinate vs indeterminate failures, could offer a pragmatic alternative to error message haruspicy.

4.4 Duplicate Effects

With errors interpreted correctly, we found that network partitions could cause MongoDB to duplicate the effects of transactions. Despite never appending the same value to an array twice, we repeatedly observed arrays with multiple copies of the same element. For example, take this test run, which contained the following transaction:

```
[[:r 436 [2 4 1 6 8 7 6]]
 [:r 456 [3 4 8 1 2]]
 [:append 456 5]
 [:r 456 [3 4 8 1 2 5]]]
```

Here, element 6 appeared twice in a read of document 436. This duplication raises the possibility that the transaction which wrote 6 to key 436 occurred both before *and* after other transactions; depending on which

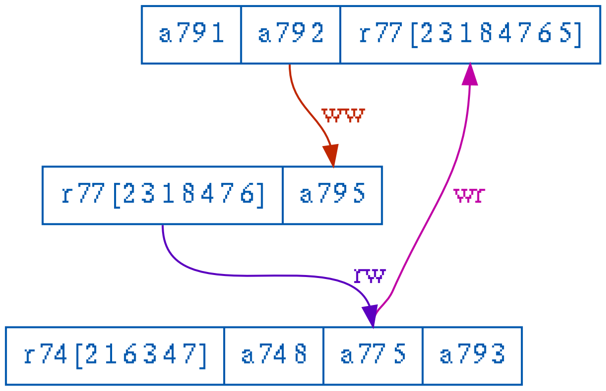
[1]This is, of course, only true for read and write concern `majority` or higher.

(if any) interpretation one chooses, the resulting history also exhibits G-single and G2 anti-dependency cycles. This anomaly occurred even with read concern `snapshot` and write concern `majority`, which suggests that even at the strongest settings, MongoDB transactions do not provide snapshot isolation.

This behavior could point to an improper transaction retry mechanism—MongoDB advertises automatic retries as a feature. To identify whether the retry mechanism might be at fault, we attempted to disable it—only to discover that MongoDB transactions *ignore* the `retryWrites` setting, and retry regardless. We are unsure if users can work around this behavior.

4.5 Read Skew

In this case, a test running with read concern `snapshot` and write concern `majority` executed a trio of transactions with the following dependency graph:



The topmost transaction appended 2 to document 79, which was followed by the middle transaction's append of 5. We can infer these writes took place in this order because another transaction (not part of this cycle) observed:

```
[:r 79 [1 2 5 6 7]]
```

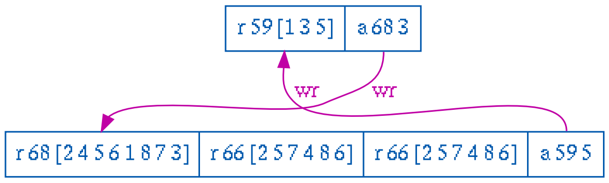
Following these writes, the bottom transaction appended 5 to document 77, which was *not* observed by the middle transaction's read of 77. However, that append of 5 *was* visible to the topmost transaction!

Because this cycle contains exactly one anti-dependency (rw) edge, it's likely[2] an example of Adya's G-single anomaly, also known as *read skew*. In essence, the middle transaction observed some, but not all, effects of logically prior transactions. Read skew is prohibited by snapshot isolation.

Curiously, this anomaly appeared in a history without (observed) duplicate writes. This suggests there could be multiple problems in the MongoDB transaction protocol, but it's difficult to say for sure.

4.6 Cyclic Information Flow

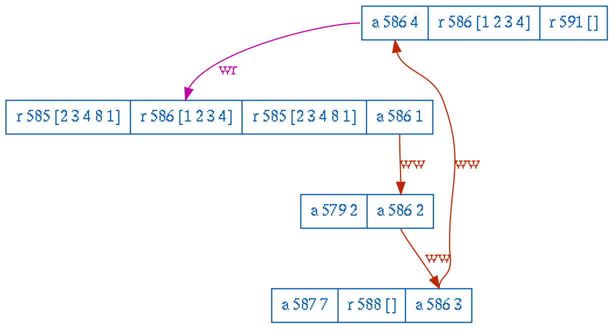
Worse yet, transactions running with the strongest isolation levels can exhibit G1c: cyclic information flow. Take this example, in which two transactions (both running at read concern `snapshot` and write concern `majority`) observe one another's effects:



The top transaction appended 3 to document 68, and the bottom transaction's read of 68 observed that write. However, the bottom transaction also appended 5 to document 59, and the top transaction read that write! These anomalies appear relatively infrequently, but we have collected a dozen-odd examples in a day's worth of testing.

4.7 Read Your (Future) Writes

It's even possible for a single transaction to observe its own future effects. In this test run, four transactions, all executed at read concern `snapshot` and write concern `majority`, append 1, 2, 3, and 4 to key 586—but the transaction which wrote 1 observed [1 2 3 4] *before* it appended 1.



This is, of course, impossible: our test submits each transaction's operations in strict order, and unless MongoDB has built a time machine, it cannot return values which it doesn't yet know will be written. This

[2]It could also be the case that some or all of these transactions observed garbage data, or values from aborted transactions. Since these anomalies are in a sense "worse" than G-single, we choose the more charitable interpretation here.

suggests that the retrocausal transaction actually ran *twice*, and on its second run, observed an effect of its own prior execution. This could be another consequence of an inappropriate retry mechanism.

5 Discussion

MongoDB 4.2.6 claims to offer "full ACID transactions" via snapshot isolation. However, the use of these transactions is complicated by weak defaults, confusing APIs, and undocumented error codes. Snapshot isolation is questionably compatible with the marketing phrase "full ACID". Even at the highest levels of read and write concern, MongoDB's transaction mechanism exhibited various anomalies which violate snapshot isolation.

MongoDB's default read and write concern for single-document operations remains `local`, which can observe uncommitted data, and `w: 1`, which can lose committed writes. Even when users select safer settings in their clients at the database or collection level, transactions ignore these settings and default again to `local` and `w: 1`. The `snapshot` read concern does not actually guarantee snapshot isolation, and must always be used in conjunction with write concern `majority`. This holds even for transactions which perform no writes.

Default behavior has significant impact: MongoDB's user research suggests roughly 80% of users of their hosted MongoDB service use the default write concern, and 99.6% of users use the default read concern. While it might be the case that many of these users *intend* to occasionally lose data or observe uncommitted state, it might also be the case that users are simply unaware of this behavior.

Nor can users rely on examples to demonstrate snapshot isolated behavior. MongoDB's transaction documentation and tutorial blog posts show only write-only transactions, using read concern `local` rather than `snapshot`. Other examples from MongoDB don't specify a read concern or run entirely with defaults. Learn MongoDB The Hard Way uses read concern `snapshot` but write concern `local`, despite performing writes. Tutorials from DZone, Several Nines, Percona, The Code Barbarian, and Spring.io all claim that transactions are either ACID or offer snapshot isolation, but none set *either* read or write concern. There are some examples of MongoDB transactions which *are* snapshot isolated—for instance, from BMC, +N Consulting, and Maciej Zgadzaj, but most uses of MongoDB transactions we found ran—either intentionally or inadvertently—with settings that would (in general) allow write loss and aborted reads.

Snapshot isolation is a reasonably strong consistency model, but claiming that snapshot isolation is "full ACID" is questionable. We routinely observed histories which appeared compatible with snapshot isolation, but also included hundreds of G2 (anti-dependency cycle) anomalies, wherein transactions failed to observe one another's effects. This is normal and allowed under snapshot isolation, but whether these transactions are fully isolated—in the sense of ACID I—seems debatable.

Finally, even with the strongest levels of read and write concern for both single-document and transactional operations, we observed cases of G-single (read skew), G1c (cyclic information flow), duplicated writes, and a sort of retrocausal internal consistency anomaly: within a single transaction, reads could observe that transaction's own writes from the future. MongoDB appears to allow transactions to both observe and not observe prior transactions, and to observe one another's writes. A single write could be applied multiple times, suggesting an error in MongoDB's automatic retry mechanism. All of these behaviors are incompatible with MongoDB's claims of snapshot isolation.

5.1 Recommendations

We continue to recommend that users of MongoDB consider their read and write concerns carefully, both for single-document and transactional operations. Settings applied at the database and collection level do not transfer to transactional contexts, even when those database or collection handles are used within the transaction. Instead, users must be careful to set read and write concern on each transaction directly. Moreover, the `snapshot` read concern does not guarantee snapshot isolation—even for read-only transactions; users must take care to use write concern `majority` whenever snapshot isolated reads are required. We recommend users perform careful review of safety-critical codepaths to look for potential mistakes.

Given this behavior, and the relative scarcity of users who actually set their read and write concern, we continue to recommend that MongoDB select safer defaults for all operations—but especially for transactions: a feature specifically intended to provide stronger safety guarantees! We also question why it's even *possible* for read concern `snapshot` to return non-snapshot-isolated reads, especially for read-only queries. While these behaviors *are* clearly documented, we think MongoDB would be better off with more conservative behavior, at least where transactions are concerned.

MongoDB's claim of "full ACID transactions" which "maintain the same data integrity guarantees you are used to in traditional databases" could be misleading. We recommend that users who are accustomed to serializable behavior evaluate critical transactions carefully, to identify whether running at snapshot isolation could violate application-level constraints. MongoDB may wish to revise their marketing language to use "snapshot isolated" instead of "ACID".

In theory, transactions running on a snapshot isolated system like MongoDB could be lifted, via static analysis, into transactions which execute serializably by materializing selected read conflicts as writes. Some database users perform a similar technique by hand, introducing no-op writes into selected transactions. However, MongoDB may optimize those writes away. MongoDB recommends incrementing a counter to force conflicts where desired.

Our research suggests that users of MongoDB 4.2.6 may experience transactional anomalies during network partitions. We suspect MongoDB may investigate and resolve these issues in future versions, and encourage users to upgrade when appropriate.

5.2 Future Work

This report represents a brief investigation relative to most Jepsen reports; we have not investigated pauses, crashes, or clock skew in depth; nor have we introduced membership changes or shard reallocation. Disk and filesystem-related issues could also prove fruitful.

Going forward, we would like to investigate collection-level transactional behavior with respect to predicates, as well as apply Elle to MongoDB's causal sessions. In addition, we would like to generalize the present tests to those with reads or writes via secondary indices or scans, as well as changing shard keys.

This work was performed independently, without compensation, and conducted in accordance with the Jepsen ethics policy. MongoDB, Fauna, and YugaByte have previously engaged Jepsen for paid analyses. Jepsen wishes to thank C. Scott Andreas, Peter Alvaro, Silvia Botros, Nicole Forsgren, Camille Fournier, Coda Hale, Marc Hedlund, Ben Linsay, Dan McKinley, Kit Patella, Marco Rogers, and James Turnbull for their invaluable feedback on early drafts. Zach Bjornson and Alex ter Weele helped correct a pair of typos in the published report. We also wish to thank MongoDB's Maxime Beugnet for inspiration.