Causal Guarantees Are Anything but Casual

Alyson Cabral
October 24, 2018 | Updated: September 20, 2022
#Engineering #EngineeringBlog

Traditional databases, because they service reads and writes from a single node, naturally provide sequential ordering guarantees for read and write operations known as "causal consistency". A distributed system can provide these guarantees, but in order to do so, it must coordinate and order related events across all of its nodes, and limit how fast certain operations can complete. While causal consistency is easiest to understand when all data ordering guarantees are preserved – mimicking a vertically scaled database, even when the system encounters failures like node crashes or network partitions – there exist many legitimate consistency and durability tradeoffs that all systems need to make.

MongoDB has been continuously running – and passing – Jepsen tests for years. Recently, we have been working with the Jepsen team to test for causal consistency. With their help, we learned how complex the failure modes become if you trade consistency guarantees for data throughput and recency.

Causal consistency defined

To maintain causal consistency, the following guarantees must be satisfied:

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.

To show how causal guarantees provide value to applications, let's review an example where no causal ordering is enforced. The distributed system depicted in Diagram 1 is a replica set. This replica set has a primary (or leader) that accepts all incoming client writes and two secondaries (or followers) that replicate those writes. Either the primary or secondaries may service client reads.

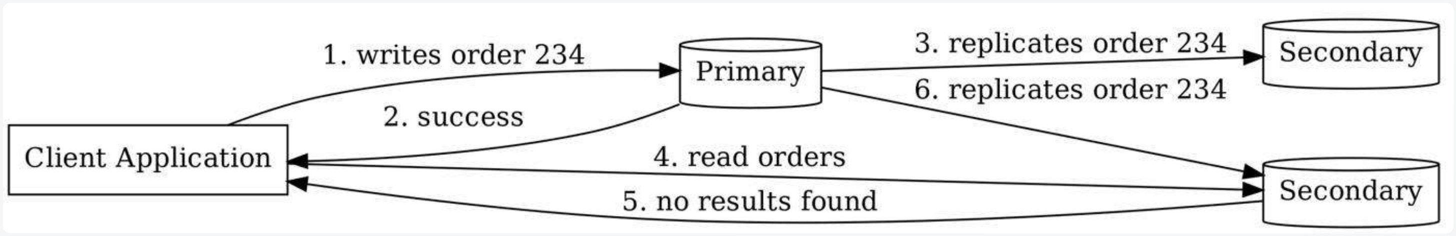


Diagram 1: Flow of Operations in a Replica Set without Enforced Causal Consistency

1. The client application writes order 234 to the primary

2. The primary responds that it has successfully applied the write

3. Order 234 is replicated from the primary to one of the secondaries

5. The targeted secondary hasn't seen order 234, so it responds with no results

6. Order 234 is replicated from the primary to the other secondary

The client makes an order through the application. The application writes the order to the primary and reads from a secondary. If the read operation targets a secondary that has yet to receive the replicated write, the application fails to read its own write. To ensure the application can read its own writes, we must extend the sequential ordering of operations on a single node to a global partial ordering for all nodes in the system.

Implementation

So far, this post has only discussed replica sets. However, to establish a global partial ordering of events across distributed systems, MongoDB has to account for not only replica sets but also sharded clusters, where each shard is a replica set that contains a partition of data.

To establish a global partial ordering of events for replica sets and sharded clusters, MongoDB implemented a hybrid logical clock based on the Lamport logical clock. Every write or event that changes state in the system is assigned a time when it is applied to the primary. This time can be compared across all members of the deployment. Every participant in a sharded cluster, from drivers to query routers to data bearing nodes, must track and send their value of latest time in every message, allowing each node across shards to converge in their notion of the latest time. The primaries use the latest logical time to assign new times to subsequent writes. This creates a causal ordering for any series of related operations. A node can use the causal ordering to wait before performing a needed read or write, ensuring it happens after another operation.

For a deeper dive on implementing cluster-wide causal consistency, review Misha Tyulenev's talk.

Let's revisit our example from Diagram 1 but now enforcing causal consistency:

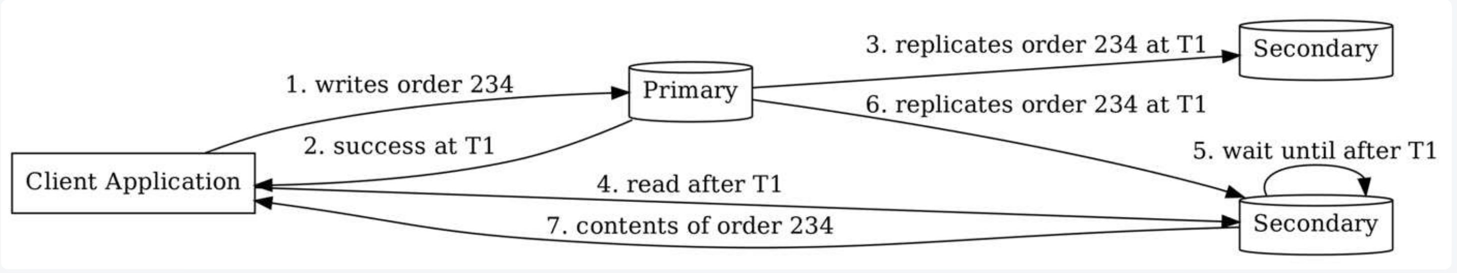


Diagram 2: Flow of Operations in a Replica Set with Enforced Causal Consistency

1. The client application writes order 234 to the primary
2. The primary responds that it has successfully recorded the write at time T1
3. Order 234 is replicated from the primary to one of the secondaries
4. The client application reads after time T1 on a secondary
5. The targeted secondary hasn't seen time T1, so must wait to respond
6. Order 234 is replicated from the primary to the other secondary

7. The secondary is able to respond with the contents of order 234

Write and read concerns

Write concern and read concern are settings that can be applied to each operation, even those within a causally consistent set of operations. Write concern offers a choice between latency and durability. Read concern is a bit more subtle; it trades stricter isolation levels for recency. These settings affect the guarantees preserved during system failures

Write concerns

Write concern, or write acknowledgement, specifies the durability requirements of writes that must be met before returning a success message to the client. Write concern options are:

`1`	Write returns a success once it has been applied to the primary
`n`	Write returns a success once it has been applied to n number of nodes
`majority`	Write returns a success once it has been applied to a majority of nodes

Only a successful write with write concern `majority` is guaranteed to be durable for any system failure and never roll back.

During a network partition, two nodes can temporarily believe they are the primary for the replica set, but only the true primary can see and commit to a majority of nodes. A write with write concern `1` can be successfully applied to either primary, whereas a write with write concern `majority` can succeed only on the true primary. However, this durability has a performance cost. Every write that uses write concern `majority` must wait for a majority of nodes to commit before the client receives a response from the primary. Only then is that thread freed up to do other application work. In MongoDB, you can choose to pay this cost as needed at an operation level.

Read concern

Read concern specifies the isolation level of reads. Read concern `local` returns locally committed data whereas read concern `majority` returns data that has been reflected in the majority committed snapshot that each node maintains. The majority committed snapshot contains data that has been committed to a majority of nodes and will never roll back in the face of a primary election. However, these reads can return stale data more often than read concern `local`. The majority snapshot may lack the most recent writes that have not yet been majority committed. This tradeoff could leave an application acting off old data. Just as with write concern, the appropriate read concern can be chosen at an operation level.

Effect of write and read concerns

With the rollout of causal consistency, we engaged the Jepsen team to help us explore how causal consistency interacts with read and write concerns. While we were all satisfied with the feature's behavior under read/write concern majority, the Jepsen team did find some anomalies under other permutations. While less strict permutations may be more appropriate for some applications, it is important to understand the exact tradeoffs that apply to any database, distributed or not.

Failure scenario examples

Consider the behavior of different combinations of read and write concerns during a network partition where P1 has been partitioned from a majority of nodes and P2 has been elected as the new primary. Because P1 does not yet know it is no longer the primary, it can continue to accept writes. Once P1 is reconnected to a majority of nodes, all of its writes since the timeline diverged are rolled back.

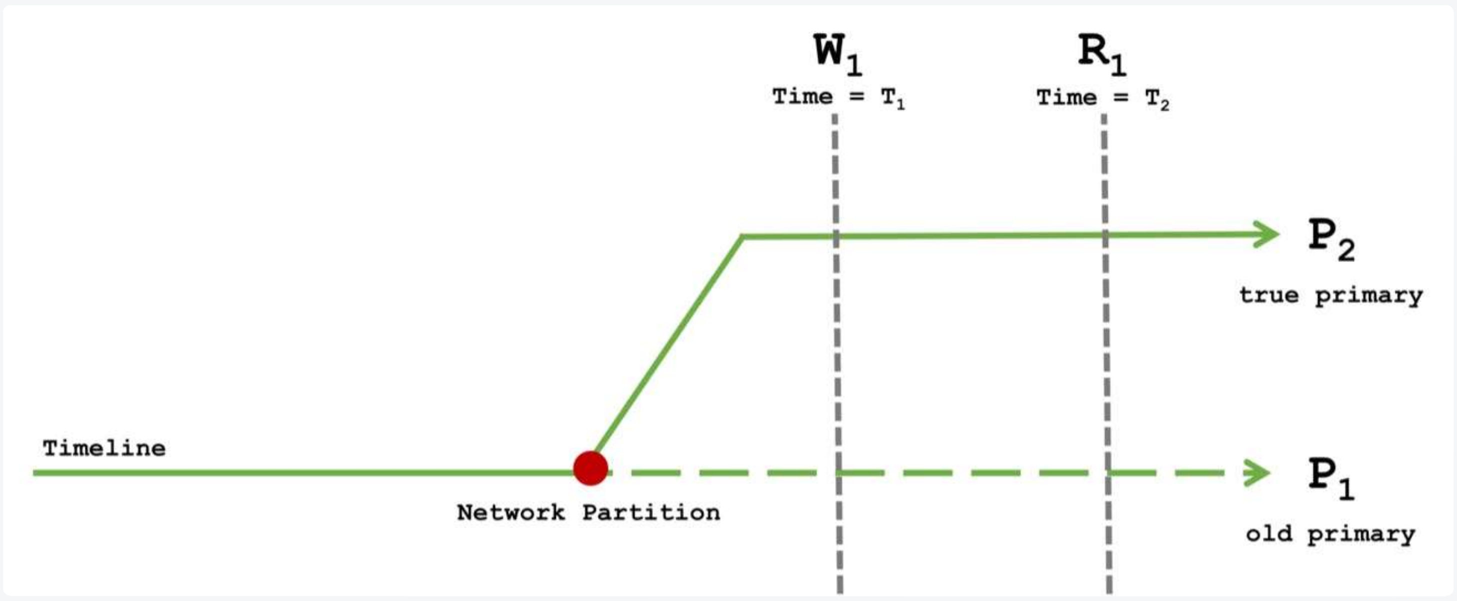


Diagram 3: Network Partition Timeline

During this time, a client issues a causal sequence of operations as follows:

1. At Time T1 perform a write W1
2. At Time T2 perform a read R1

The following four scenarios discuss the different read and write concern permutations and their tradeoffs.

Read Concern majority with Write Concern majority

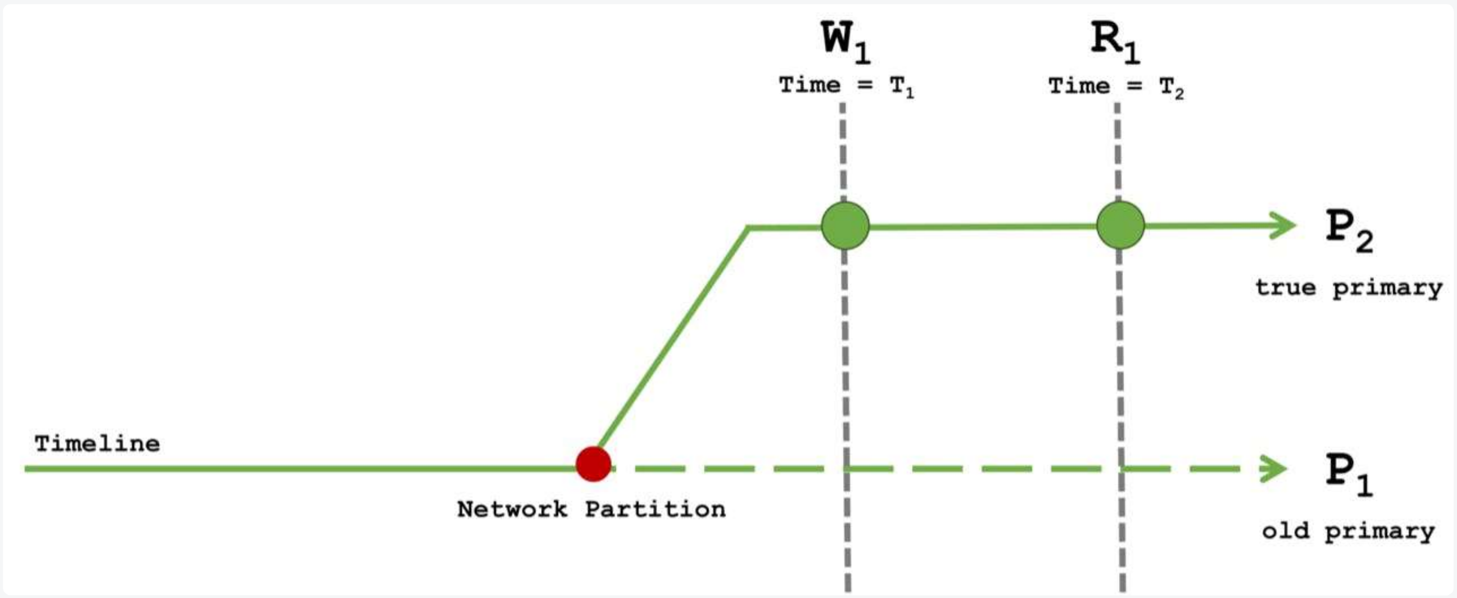


Diagram 4: Read Concern majority with Write Concern majority

The write W1 with write concern `majority` can only succeed when applied to a majority of nodes. This means that W1 must have executed on the true primary's timeline and cannot be rolled back.

The causal read R1 with read concern `majority` waits to see T1 majority committed before returning success. Because P1, partitioned from a majority of nodes, cannot progress its majority commit point, R1 can only succeed on the true primary's timeline. R1 sees the definitive result of W1.

All the causal guarantees are maintained when any failure occurs. All writes with write concern `majority` prevent unexpected behavior in failure scenarios at the cost of slower writes. For their most critical data, like orders and trades in a financial application, developers can trade performance for durability and consistency.

Read Concern majority with Write Concern 1

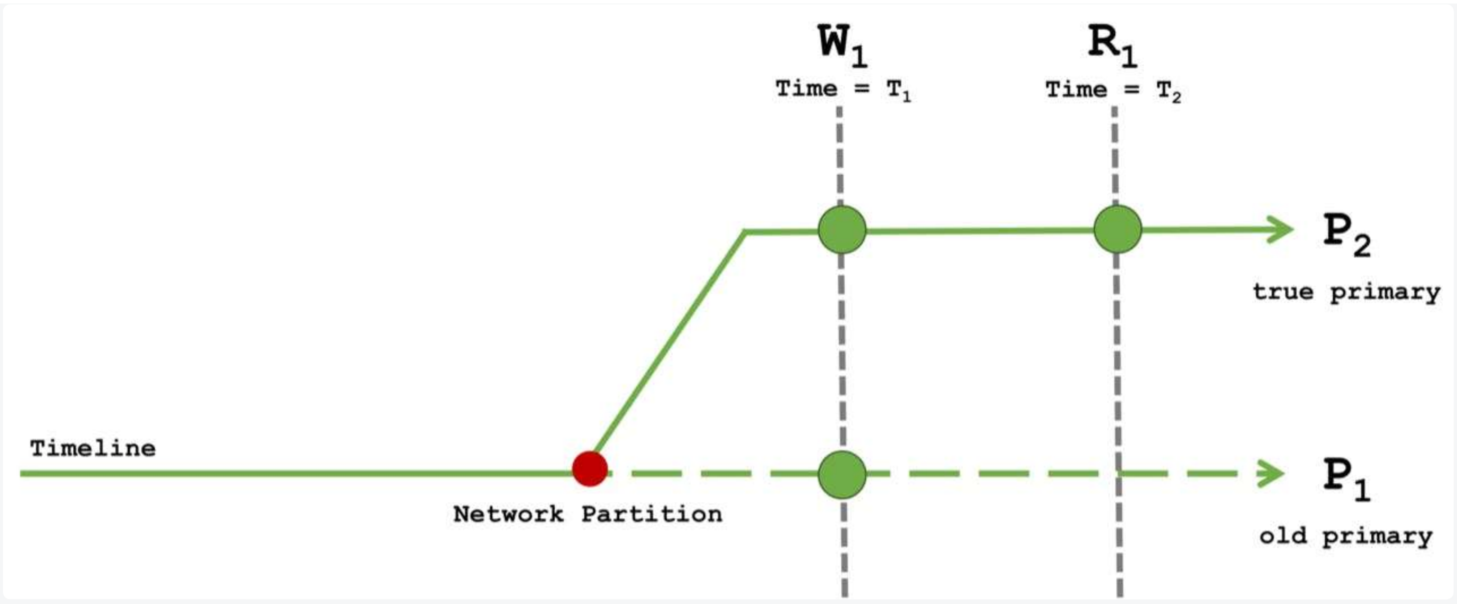


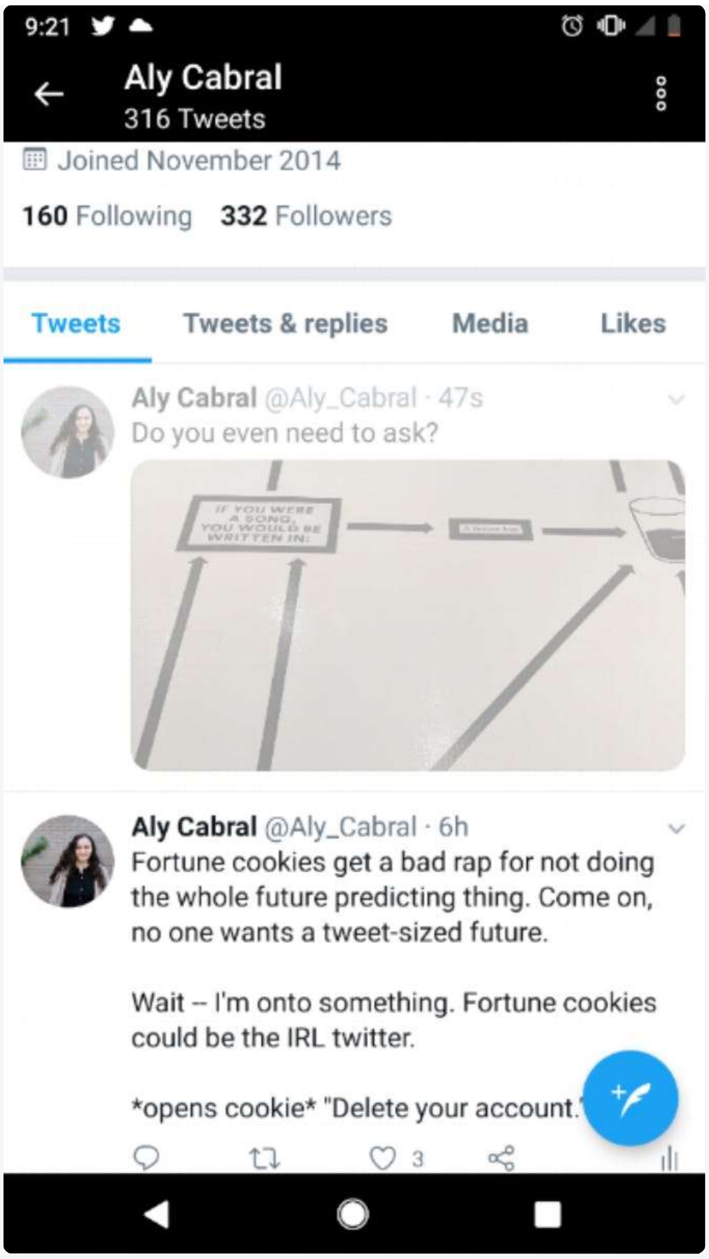
Diagram 5: Read Concern majority with Write Concern 1

The write W1 using write concern `1` may succeed on either the P1 or P2 timeline even though a successful W1 on P1 will ultimately roll back.

The causal read R1 with read concern `majority` waits to see T1 majority committed before returning success. Because P1, partitioned from a majority of nodes, cannot progress its majority commit point, R1 can only succeed on the true primary's timeline. R1 sees the definitive result of W1. In the case where W1 executed on P1, the definitive result of W1 may be that the write did not commit. If R1 sees that W1 did not commit, then W1 will never commit. If R1 sees the successful W1, then W1 successfully committed on P2 and will never roll back.

This combination of read and write concerns gives causal ordering without guaranteeing durability if failures occur.

Consider a large scale platform that needs to quickly service its user base. Applications at scale need to manage high throughput traffic and benefit from low latency requests. When trying to keep up with load, longer response times on every request are not an option. The Twitter posting UI is a good analogy for this combination of read and write concern:


9:21

Aly Cabral
316 Tweets

Joined November 2014

160 Following 332 Followers

Aly Cabral @Aly_Cabral · 47s
Do you even need to ask?

Aly Cabral @Aly_Cabral · 6h
Fortune cookies get a bad rap for not doing the whole future predicting thing. Come on, no one wants a tweet-sized future.

Wait -- I'm onto something. Fortune cookies could be the IRL twitter.

opens cookie "Delete your account."

The pending tweet, shown in grey, can be thought of as a write with write concern `1`. When we do a hard refresh, this workflow could leverage read concern `majority` to tell the user definitively whether the post persisted or not. Read concern `majority` helps the user safely recover. When we hard refresh and the post disappears, we can try again without the risk of double posting. If we see the post after a hard refresh at read concern `majority`, we know there is no risk that post ever disappearing.

Read Concern local with Write Concern majority

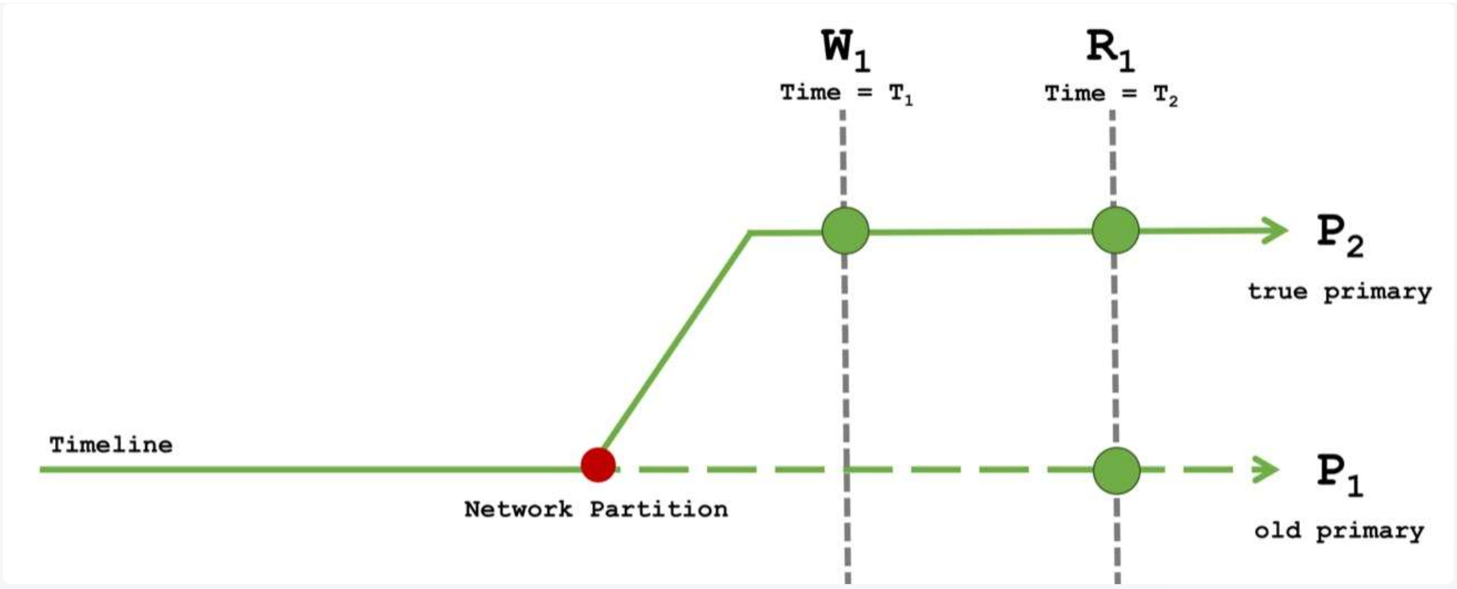


Diagram 6: Read Concern local with Write Concern majority

The write W1 with write concern `majority` can only succeed when applied to a majority of nodes. This means that W1 must have executed on the true primary's timeline and cannot be rolled back.

With read concern `local`, the causal read R1 may occur on either the P1 or P2 timeline. The anomalies occur when R1 executes on P1 where the majority committed write is not seen, breaking the "read your own writes" guarantee. The monotonic reads guarantee is also not satisfied if multiple reads are sequentially executed across the P1 and P2 timelines. Causal guarantees are not maintained if failures occur.

Consider a site with reviews for various products or services where all writes are performed with write concern `majority` and all reads are performed with read concern `local`. Reviews require a lot of user investment, and the application will likely want to confirm they are durable before continuing. Imagine writing a thoughtful two-paragraph review, only to have it disappear. With write concern `majority`, writes are never lost if they are successfully acknowledged. For a site with a read heavy workload, greater latency of rarer `majority` writes may not affect performance. With read concern `loca`, the client reads the most up-to-date reviews for the targeted node. However, the targeted node may be P1 and is not guaranteed to include the client's own writes that have been successfully made durable on the true timeline. In addition, the node's most up-to-date reviews may include other reviewers' writes that have not yet been acknowledged and may be rolled back.

Read Concern local with Write Concern 1

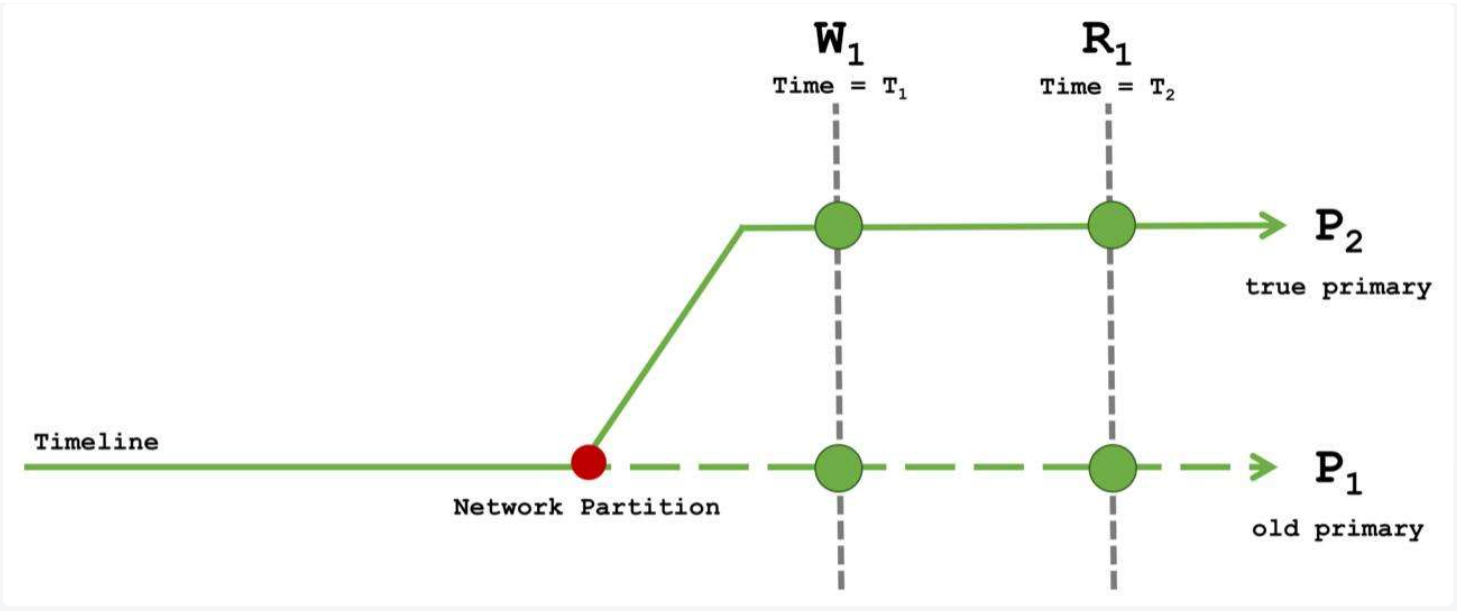


Diagram 7: Read Concern local with Write Concern 1

The combination of read concern `local` and write concern `1` has the same issues as the previous scenario but now the writes lack durability. The write W1 using write concern 1 may succeed on either the P1 or P2 timeline even though a successful W1 on P1 will ultimately roll back. With read concern `local`, the causal read R1 may occur on either the P1 or P2 timeline. The anomalies occur when W1 executes on P2 and R1 executes on P1 where the results of the write is not seen, breaking the "read your own writes" guarantee. The monotonic reads guarantee is also not satisfied if multiple reads are sequentially executed across the P1 and P2 timelines. Causal guarantees are not maintained if failures occur.

Consider a sensor network of smart devices that does not handle failures encountered when reporting event data. These applications can have granular sensor data that drives high write throughput. The ordering of the sensor event data matters to track and analyze data trends over time. The micro view over a small period of time is not critical to the overall trend analysis, as packets can drop. Writing with write concern `1` may be appropriate to keep up with system throughput without strict durability requirements. For high throughput workloads and readers who prefer recency, the combination of read concern `local` and write concern `1` delivers the same behavior of primary only operations across all nodes in the system with the aforementioned tradeoffs.

Conclusion

Each operation in any system, distributed or not, makes a series of tradeoffs that affect application behavior. Working with the Jepsen team pushed us to consider the tradeoffs of read and write concerns when combined with causal consistency. MongoDB now recommends using both read concern `majority` and write concern `majority` to preserve causal guarantees and durability across all failure scenarios. However, other combinations, particularly read concern `majority` and write concern `1`, may be appropriate for some applications.

Offering developers a range of read and write concerns enables them to precisely tune consistency, durability, and performance for their workloads. Our work with Jepsen has helped better characterize system behavior under different failure scenarios, enabling developers to make more informed choices on the guarantees and trade-offs available to them.