Causal Consistency Guarantees in MongoDB – Lamport Clock, Cluster Time, Operation Time, and Causally Consistent Sessions

In Distributed Systems, MongoDB

Tags causal consistency, causally consistent sessions, cluster time, distributed systems, lamport clock, MongoDB, operation time

December 13, 2021 Vasil Kosturski

Table of Contents

Intro

In the last article, we explored Majority Reads and Majority Writes in Mongo in an attempt to fix a "Read Your Write" consistency violation. That was insufficient due to the replication delay for updating the Majority Snapshot on the Secondary servers.

In this post, we'll finally see how to solve our consistency issue using Causally Consistent Sessions introduced in Mongo 3.6 *(*)*.

(*) It was still possible to achieve Causal Consistency before Mongo 3.6, but it involved forcing the reads to the Primary, which is quite limiting in terms of scalability.

We'll first say a few words about Lamport Clocks (or Logical Clocks) and how we model causal relations in a Distributed System.

Then I'll describe the actual Mongo-specific implementation by investigating the topics of **Cluster Time, Operation Time, and Causally Consistent Sessions**.

Lastly, I'll present the final code sample that guarantees Causal Consistency.

Let's get going!

> **Useful Resources**
>
> This article is very much influenced by the Designing Data-Intensive Applications book. I strongly recommend it if you want to learn more about the nuts and bolts of Data-Driven Systems.

"Read Your Write" Violation – Recap

I would strongly suggest visiting the Causal Consistency intro article to see much more in-depth use cases. Still, at this point, it's enough to understand the workflow from the diagram below:

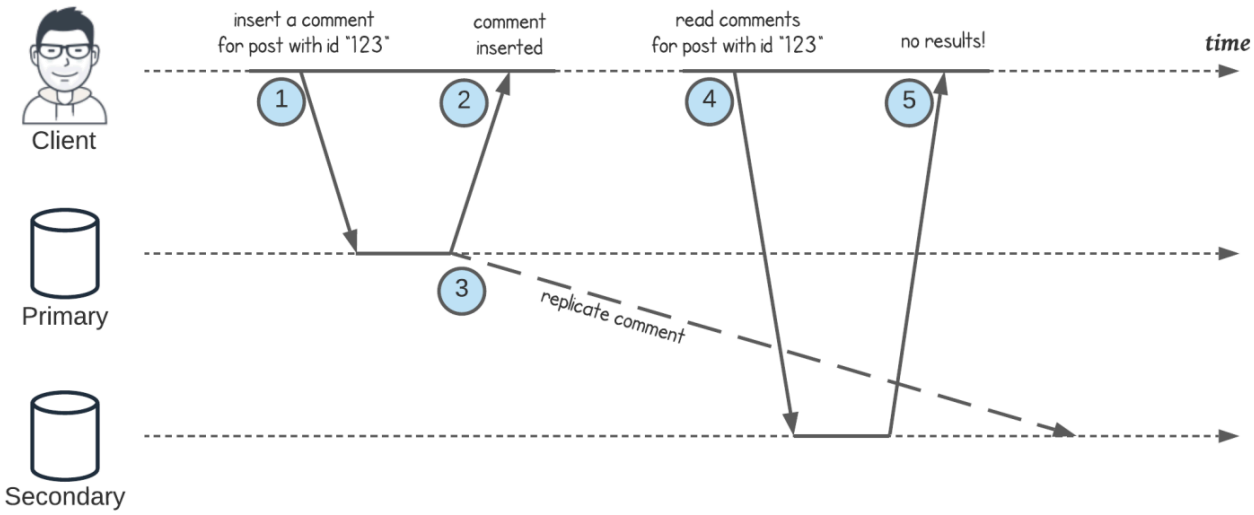


Fig. 1 – Violation of "Read Your Write" consistency

In essence, the Client performs a Write operation to the Primary server. Then he tries to read the same Write, but the Read request goes to the Secondary. The Write is still not replicated to the Secondary, so the Client **cannot read his write**.

Let's see how to solve this!

Solution – High-Level Idea

Imagine the **Primary server keeps a counter** and attaches an auto-incrementing number to every Write. This number is returned to the Client when the Write operation completes.

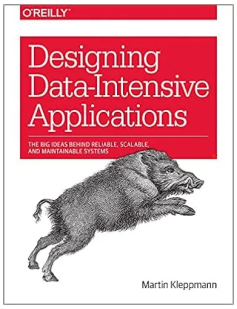
Also, **the counter value gets propagated to the secondaries** during standard replication. In other words, every Secondary knows the latest value of the counter it has seen from the Primary.

So, how does this help with achieving Causal Consistency Guarantees?

The counter represents the ordering of the events in the replica set. **If WriteX has a lower counter value than WriteY, then WriteX happened before WriteY.**

When the Client receives the counter value of his Write operation, he then passes this value in a subsequent Read request to a (potentially) Secondary server. This instructs the **Secondary to wait** until the Write with the specified number is replicated.

In Mongo, this happens automatically when your Write and Read requests are part of a Causally Consistent Session. You'll see a concrete example at the end of the article.



Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems [Kleppmann, Martin] on Amazon.com. *FREE* shipping on qualifying offers. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Conceptual Diagram

The following diagram demonstrates the workflow:

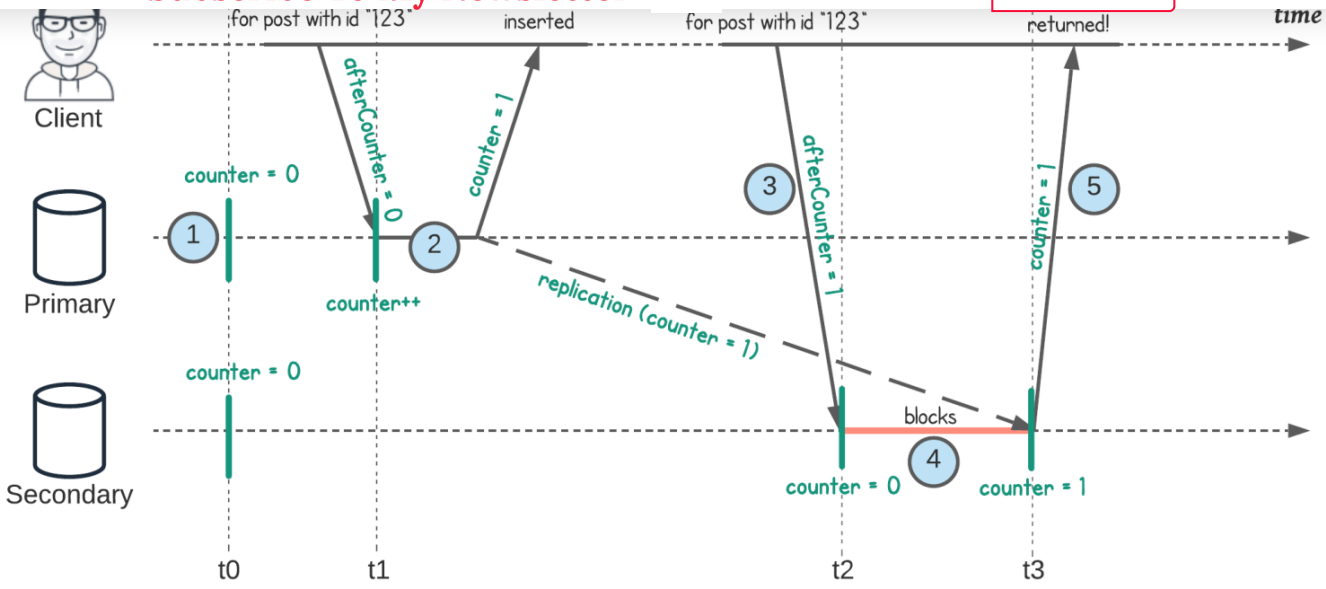


Fig. 2 – Causal Consistency Conceptual Diagram

1. At t0, the `counter` is 0 on both the Primary and the Secondary.
2. At t1, the Client initiates a Write request to the Primary. As with any other operation in a Causally Consistent Session, he passes the latest known `counter` value in the `afterCounter` parameter. Then the Primary performs the Write and **increments the** `counter`, which is returned to the Client as part of the "successful operation" message. Also, at this point, the Write (with the counter attached) is scheduled for replication.
3. On a subsequent Read, the **Client passes the** `counter` received from the previous Write. This instructs the server to wait until the Write with `counter = 1` gets replicated.
4. The Read request on the Secondary **is blocked**, waiting for the Write.
5. When the Write is propagated, the `counter` **on the Secondary is updated**, the query is executed, and returns the correct result.

The critical difference with the "failing" version (Fig. 1) of this workflow is in point 4, where the Secondary blocks the request until its `counter` catches up.

This is how **Causal Consistency is guaranteed** conceptually.

Let's dive into the details.

Lamport Clocks

a Distributed System by Leslie Lamport. It is one of the most-cited publications in Distributed Systems theory.

Simply put, the main advantage of a Lamport Clock is it's just a simple scalar value that represents the ordering of events with *happened-before* relations. By receiving and passing the latest timestamp he's aware of, the Client guarantees to **read/write causally related events in the correct order**.

Also, being a simple integer, a Lamport Clock overcomes a lot of the limitations of physical time that you'll see in the section below.

Bear in mind that the use case we cover in this article is quite simpler than the ones described in the Lamport Clocks paper. The reason is that Mongo is a Single Leader database – you can write to a single Primary server per replica set. In general, Lamport Clocks are described in terms of a Multi Leader setup which makes the problem more complex.

Still, the main logic stays the same. Feel free to explore the topic at a deeper level on your own.

You can read more about Lamport Clocks in Designing Data-Intensive Applications, *Chapter 9 – Consistency and Consensus, Section – Ordering Guarantees -> Lamport Timestamps.*

Why Not Using Physical Time (Wall Clocks)?

You might be wondering, can't we just use the physical time (wall clock) for events ordering. For example, if Event A on Server X happened at 12:00:01 and Event B on Server Y at 12:00:02, that clearly means Event B happened after Event A, right?

Not really.

Physical clocks are one of the most unreliable things in a Distributed System.

Each machine has its own clock – a quartz crystal oscillator. This is not super accurate and has a tendency to drift – it might go slightly faster or slower than other machines.

To deal with the *drift*, a common approach is to synchronize the clocks using NTP (Network Time Protocol). This is essentially asking a group of servers for the correct time.

A whole set of articles can be written on the problems of physical clocks in a Distributed System, but this should be enough to get a sense of the issues they can cause.

Lamport Clocks in Mongo

MongoDB implements Lamport Clocks via Causally Consistent Sessions that I'll demonstrate with a code sample in the next section.

Before that, as you're now familiar with the concept of Lamport Clocks and the example from Fig. 2, it's just a matter of changing a few terms to understand the implementation in Mongo.

The diagram below should make a lot of sense:

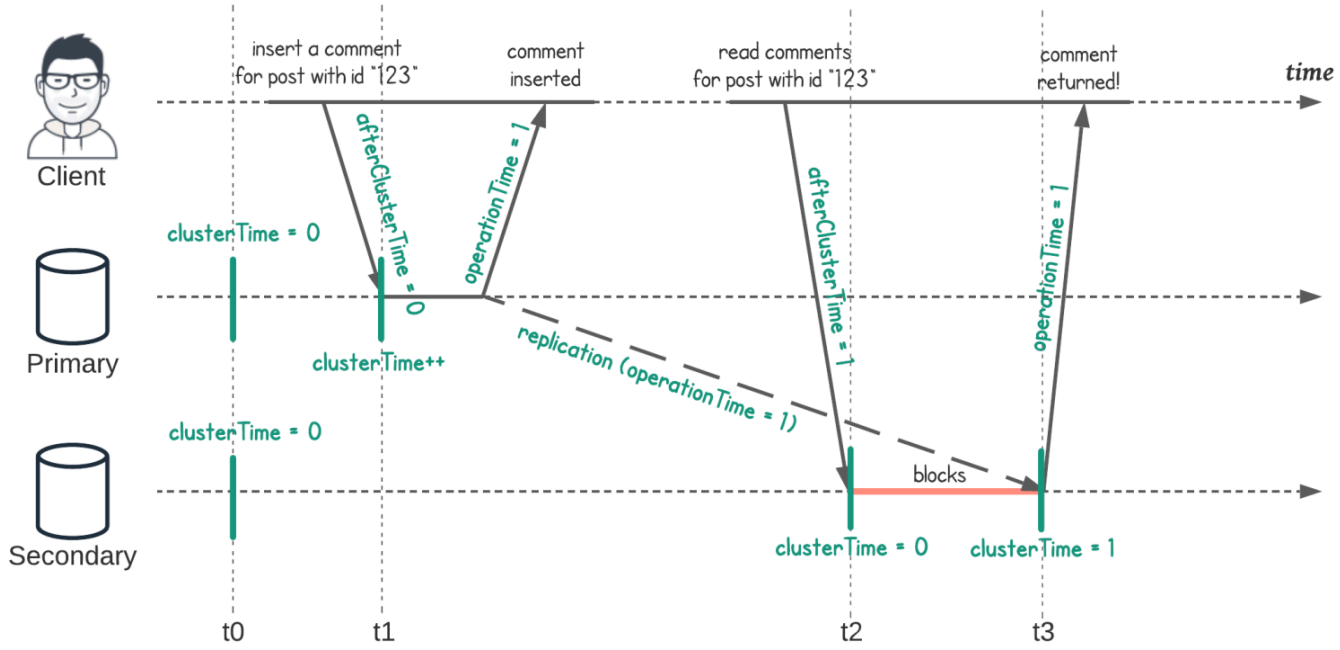


Fig. 3 – Causal Consistency in Mongo

The `counter` from before is pretty much replaced with `clusterTime`. Also, the `clusterTime` at the time of the operation, returned to the Client, is called `operationTime`.

Of course, this description is still a little simplified. The implementation in Mongo contains many more details behind the scenes that are required for a production system with millions of users.

You can read more about the specifics of the Mongo implementation in the paper Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB.

Causally Consistent Sessions in Mongo

Simply put, in order to take advantage of Lamport Clocks and guarantee Casual Consistency in Mongo, you need to use Causally Consistent Sessions.

Here is the final version of the code that fixes our original "Read Your Write" issue.

```
1.  static void Main(string[] args)
2.  {
3.      var client = new MongoClient("mongodb://localhost:27018,localhost:27019,localhost:27020/?replicaSet=rs0");
4.
5.      using var session = client.StartSession(new ClientSessionOptions { CausalConsistency = true });
6.
7.      var collection = client.GetDatabase("test-db")
8.          .GetCollection<BsonDocument>("test-collection")
9.          .WithWriteConcern(WriteConcern.WMajority)
10.         .WithReadConcern(ReadConcern.Majority)
11.         .WithReadPreference(ReadPreference.Secondary);
12.
13.     var sw = new Stopwatch();
14.     sw.Start();
15.     while (sw.ElapsedMilliseconds < 5000)
16.     {
17.         var newDocument = new BsonDocument();
18.
19.         collection.InsertOne(session, newDocument);
20.
21.         var foundDocument = collection.Find(session, Builders<BsonDocument>.Filter.Eq(x => x["_id"], newDocument["_id"]))
22.             .FirstOrDefault();
23.
24.         if (foundDocument == null)
25.             throw new Exception("Document not found");
26.     }
27.
28.     Console.WriteLine("Success!");
29. }
```

This piece of code finally outputs the "Success!" message.

The important part is that we initialize a Causally Consistent Session (line 5) and use it when making the Insert (line 19) and Find (line 21) calls.

Summary

In this article, we reviewed the concept of Lamport Clocks and how they help preserve causal relations.

We've examined the concrete implementation in Mongo by digging into Causally Consistent Sessions.

In the end, I presented the final code sample, which fixes our "Read Your Write" consistency violation.

Thanks for reading, and see you next time!

Resources

1. Designing Data-Intensive Applications
2. Introduction to Database Engineering, Udemy
3. Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB
4. Time, Clocks, and the Ordering of Events in a Distributed System

Vasil Kosturski

« Previous Post:
Causal Consistency Guarantees in MongoDB – "majority" Read and Write Concerns

Next Post:
Should You Migrate Your Reporting Queries From a "General Purpose" DB (MongoDB) to a Data Warehouse (ClickHouse)? (Performance Overview) »