Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB

Misha Tyulenev
MongoDB, Inc.
misha@mongodb.com

Andy Schwerin
MongoDB, Inc.
schwerin@mongodb.com

Asya Kamsky
MongoDB, Inc.
asya@mongodb.com

Randolph Tan
MongoDB, Inc.
randolph@mongodb.com

Alyson Cabral
MongoDB, Inc.
alyson.cabral@mongodb.com

Jack Mulrow
MongoDB, Inc.
jack.mulrow@ mongodb.com

ABSTRACT

MongoDB is a distributed database that supports replication and horizontal partitioning (sharding). MongoDB replica sets consist of a primary that accepts all client writes and then propagates those writes to the secondaries. Each member of the replica set contains the same set of data. For horizontal partitioning, each shard (or partition) is a replica set.

This paper discusses the design and rationale behind MongoDB's implementation of a cluster-wide logical clock and causal consistency. The design leveraged ideas from across the research community to ensure that the implementation adds minimal processing overhead, tolerates possible operator errors, and gives protection against non-trusted client attacks.

While the goal of the team was not to discover or test new algorithms, the practical implementation necessitated a novel combination of ideas from the research community on causal consistency, security, and minimal performance overhead at scale. This paper describes a large scale, practical implementation of causal consistency using a hybrid logical clock, adding the signing of logical time ranges to the protocol, and introducing performance optimizations necessary for systems at scale. The implementation seeks to define an event as a state change and as such must make forward progress guarantees even during periods of no state changes for a partition of data.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06.
https://doi.org/10.1145/3299869.3314049

KEYWORDS: Causal Consistency, Eventual Consistency

ACM Reference Format
Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, Jack Mulrow. 2019. Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB. In 2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3299869.3314049

1 Introduction

Many applications require observing events in the causal order, whereby the order of operations logically follows a sequence of client events. Here are just a few applied examples:

- An ATM cash deposit (a write) followed by a balance check (a read) must maintain logical ordering. If the balance check request is routed to a node that is not caught up with the leader it will deliver the incorrect balance information, potentially allowing overdrafting.
- On social media, a user may update their access control settings to restrict access to a new photo album or conversation topic before adding content. If the updates to access control settings and content are viewed out of order, the security policy may be violated.
- Consider a currency trading application. The price of Mexican Peso/Russian Ruble (MXN/RUB) is derived from USD/RUB and USD/MXN currency pairs. The spot prices are highly dynamic. Delivering the prices for MXN/RUB and USD/RUB out of order can introduce an arbitrage opportunity which will cause material losses for the market maker.

Single node systems naturally provide these sequential ordering guarantees for client read and write operations, as all operations are routed to the same node. Extending this behavior to distributed systems has been a part of

distributed systems research for decades, starting with Lamport's seminal paper on the Lamport Clock [23]. There are hundreds of papers about causal consistency. At the time of writing this paper the ACM digital library shows 235 references, illustrating the community interest in the subject. A contributing factor to this interest is that causal consistency is the strongest consistency model for always-available one-way convergent systems that tolerate partitions [26], making it very attractive for modern distributed applications where availability is one of the highest priorities.

Despite the commercial and research interest in features of causal consistency, there are very few industry implementations. As an analysis of MongoDB 3.6.4 by the Jepsen team [37] denotes: *"MongoDB is one of the first commercial databases we know of which provides an implementation [of causal consistency]"*. Additionally, a recent paper [27] notes that: *"Yet, causal consistency has not seen widespread industry adoption"*. There is vast and mature research on this topic but few commercial implementations. There are several reasons for this:

- Causal consistency in many practical cases is highly probable [9] and for some applications performance is more critical than rare cases of incorrectness.
- Causal consistency can be implemented on the client side and many production applications needing this feature already implemented it that way or used stronger consistency models provided by databases.
- Causal consistency implementations based on dependency graph tracking introduce performance overhead that often cannot be tolerated by production application demands.

As a commercial product, MongoDB's implementation of causal consistency needs to address and balance the entire spectrum of requirements starting with performance and scalability to database security so malicious attacks cannot break or compromise the system. This paper will discuss MongoDB's multi-dimensional requirements for an applied solution, including performance, scalability, and security. These requirements were used as a guide for implementation decisions and ultimately lead to the need to combine aspects from several research areas. This paper will also describe how changes made while implementing this feature significantly improved performance and simplified implementation across all system components.

One of the underlying contributors to the implementation direction revolves around the understanding that the consistency model is a fundamental property of a distributed database and is used as the basis for building distributed transactions. As such, it must be easy to debug and diagnose. The practical issues, such as code simplicity that improve reliability and maintainability also play an important role in a commercial product development. In practice, code simplicity means fewer moving parts.

Another important requirement is backward compatibility. For instance, newer versions of a database must work with older versions of client drivers. In MongoDB's case, while there are officially supported drivers, there are also several community-supported drivers that may not be able to support breaking changes in the protocol. As a modern database for mission-critical workloads, MongoDB provides rolling upgrades – i.e. upgrades to a new version without downtime. The system must be capable of operating in a state where some nodes have been upgraded to a newer version and some are operating with the old version. Any solution must provide the ability to abort an upgrade and safely rollback to the prior version. Additionally, the implementation must be an extension to an existing client protocol that will not break older clients.

MongoDB 3.6, announced in November 2017, introduced causal consistency and has been successfully used by customers for more than a year without the need to make any significant changes.

The rest of the paper is organized as follows: In the "Related Work" section we will overview the relevant papers, then in the "Design Choices" section we will present arguments on how our requirements and priorities affected the choices. In the "System Model" we describe MongoDB's distributed architecture. In the "Using Causal Consistency" section we will describe how adding Causal Consistency helps applications to improve functionality and service. Then in "Performance Optimization" we will present the challenges that arise when implementing vanilla causal consistency and our approach to solve them. Finally, in the "Performance Evaluation" we will present an evaluation of performance cost of using Causal Consistency under various conditions and conclude. The "Appendix" section presents detailed description of algorithms, data structures and protocols of the implementation.

2 Related Work

The CAP theorem is a conjecture presented by Eric Brewer in 2000 with a formal proof published by Seth Gilbert and Nancy Lynch in 2002 [19]. The theorem states that in the presence of network partitions, a system cannot provide availability and atomic consistency guarantees at the same time. Since network partitions are a reality for large-scale distributed systems [7], distributed systems must tradeoff between consistency and availability. While the CAP

conjecture played an important role in understanding distributed systems, in practice most of the systems do not satisfy all of the theorem's conditions. [20]. The linearizability requirement is expensive and, while supported by MongoDB, used less often than durable reads due to latency cost. Database consistency for systems used in practice is wider than the class of systems that match CAP theorem limitations. The distributed database movement has helped to understand consistency vs. availability tradeoffs in more detail, leading to research of the architectures supporting consistency models that are weaker than atomic consistency but still satisfy application requirements [8].

2.1 Eventual Consistency Behavior

Eventual consistency is the guarantee that all nodes of any given data item in the absence of data changes will converge to the same value at some point in the future. This is a liveness property for distributed databases and should not be violated. While eventual consistency is a weak property, it is supported by many modern distributed data stores (Cassandra [22], DynamoDB [15], Solr [4], Riak [11]). The reasons for such wide adoption are:

1. Eventual consistency does not affect system availability and performance, which are major requirements for production distributed systems [6].
2. In practice users often can specify additional requirements on operations that provide safety guarantees. For example, a user may choose to read only data committed to the majority of replicas, to avoid the risk of a rollback in the case of partition (implemented in MongoDB as readConcern level "majority" [31])

However, eventual consistency does not always satisfy the needs of application developers. Therefore, in some cases the application requirements in an eventually consistent system force developers to add complex application logic that often introduces unnecessary latency, degrading performance and user experience.

2.2 Causal Consistency Behavior

In contrast with the behavior of eventual consistency, causal consistency guarantees manifest to clients by the following properties:

- Client can read its own writes
- Client can write data that follows its reads
- Monotonic reads
- Monotonic writes

These properties can be helpful when developing applications and eliminate the need to write complicated workarounds in the application logic.

Causal Consistency [1] is defined as a model that preserves a partial order of events in a distributed system. If an event A causes another event B, causal consistency provides an assurance that every other process in the system observes the event A before observing event B. Causal order is transitive: if A causes B and B causes C then A causes C. Non causally ordered events are *concurrent*.

The concept of causal consistency is closely connected to the more general topic of ordering events in a distributed system. There are numerous research papers on this subject, but the most influential is the paper published by Leslie Lamport in 1978 [23]. This work defines a total order of events across nodes in a distributed system, establishing the "*happened-before*" relation between operations.

An event X *happened-before* event Y if

1. X has occurred before Y in the same process that processes events sequentially
2. X is the sending of a message and event Y is the reception of the message sent in event X

The Lamport Clock (denoted as LC()) is a scalar value that captures this *happened-before* relation. If the event X occurred at the LC(X) and event Y occurred at LC(Y) and X *happened-before* Y than LC(X) < LC(Y). The converse is not always true, i.e. we could have LC(X) < LC(Y) but X and Y are *concurrent*. Before sending a message, each process increases its LC value by 1. The messages carry the LC value of the sending process at the moment they are sent. When a new message arrives to a process, that process updates its current LC value to the greater of its old LC value, or the LC value carried with the incoming message, and then immediately increases this value by 1.

2.3 Causal Consistency in Practice

Despite being proven as the strongest consistency model that is available in the presence of partitions [26], causal consistency is not widely supported by production distributed systems. Because of the lack of real-world testing, the issues that may arise in causal consistency implementations in production distributed data stores are not well known.

Classical causal consistency models attach the writes that subsequent writes depend on, making the dependency graph potentially large and its processing expensive. A design approach [12] where a communication channel is

responsible for providing message delivery in the causal order in the worst case requires the buffering requirements to grow quadratically in the number of processes in the system.

Since it is too expensive for most modern applications to track all possible causal dependencies in production systems, Balis et al proposed narrowing down the dependencies to the events that mattered, passing the responsibility of defining causal events to the application [10]. In this approach, the data store focuses on enforcing causal relationships on the events explicitly requested by the application instead of enforcing it on the global history of events that ever occurred. This approach is further developed in the "Bolt-on Causal Consistency" paper [5]. The solution describes a layered approach to causality tracking and data store layers that work with eventually consistent storage systems. However, the layered approach with a dedicated agent leads to performance overhead. The overhead includes the need for all to all replication, local tracking of causally dependent messages, and expensive processing of the dependency graph.

Additionally, the per-record timeline consistency implemented by Yahoo Pnuts [13] provides an API for a variety of levels from eventual consistency to serializability. The implementation guarantees that all changes for the record are applied in the same order on all replicas. The API allows addressing a specific version, therefore, allowing implementation of causally consistent reads and writes per record on the client. However, MongoDB's approach needs to extend these causal guarantees across records.

One of the less studied issues is the impact causal ordering may have on the system's vulnerability. MongoDB's operational model permits communication with the clients that are outside of the system authentication perimeter and, therefore, cannot be trusted. In other words, MongoDB needs to assume that messages from clients can be produced by malicious users that are trying to break the system. "Practical Fault Tolerance" [24] presents an approach which builds on top of the crash-fault tolerant replication core, such as Paxos or Raft, and protocol-provided data reliability to tolerate Byzantine faults. In the view-change XPaxos protocol, the participating nodes sign their messages with their own private keys and verify incoming messages using the corresponding public keys. The signature-verification mechanism helps to detect non-crash faults and isolate the malicious nodes. MongoDB's implementation takes this approach, optimizes the performance for high throughput systems, and applies it to global causal ordering.

Another aspect explored in a recent analysis of MongoDB [37] is whether causality and durability should be considered different properties that can be satisfied independently. As a MongoDB write can be acknowledged when locally committed to a node, before it is majority committed and safe from rollback, it is possible the ultimate result of a write is that the write failed. In this case, the causal ordering of committed writes is maintained, but some sub-majority writes can fail because they are unable to commit to a majority of nodes. For example, some writes may ultimately fail due to a partition, but the order of the committed writes will remain causal. There is an argument that this behavior is still causally consistent even if some of the writes are not durable. This allows us to separate durability guarantees from causality guarantees. However, the authors of the Jepsen analysis [37] recommend using majority writes to ensure durability as well. Additionally, the analysis observed that MongoDB provides sequential consistency that forces a total order for individual keys when used with majority reads and writes.

3 Contributions

The implementation of causal consistency in a large-scale commercial data store requires multi-dimensional evaluation criteria. These criteria were used to evaluate existing approaches. No single approach satisfied every need. This paper contributes a simple, reliable, and extendable design that combines several ideas from the separate fields. Specifically, the application combined a hybrid logical clock with gossiping, signed ranges of time, and protection from jumping too far into the future due to operator error.

Additionally, MongoDB's implementation defines events solely as state changes and as such implemented a solution that still has forward logical time progression guarantees even in the face of no state changes. This forward progress guarantee is satisfied by the implementation of a noop (no-operation) writer discussed in Appendix A2: *Progressing Time in the Presence of No State Changes* section. The design choices are also presented in detail as it illustrates the requirements specific to production databases.

4 Design Choices

This section discusses the implementation choices and design challenges of delivering causal ordering. It outlines the evaluation criteria the MongoDB engineering team applied to varying existing approaches. Ultimately this evaluation leads to the necessity of combining aspects of various approaches to deliver a practical implementation.

The implementation choices are the following:

- Type of clock
- Dependency tracking
- Clock synchronization
- System security

4.1 Type of Clock

There are several variations of Lamport's idea, which can be categorized as follows:

a. Scalar logical clock. The Lamport Clock discussed above.
b. Vector logical clock. While the Lamport Clock can assign the same value to independent events on different nodes, a vector clock can distinguish between these independent events across nodes. A vector clock of a system consisting of N processes is an array of N logical clock counters. The events on the processes P and Q are tracked in separate counters. However, a vector clock requires O(N) space for each message [18]
c. Synchronized clocks. Synchronized wall clocks are implemented on nodes using custom hardware. [14]
d. Hybrid Logical Clock (HLC). The Lamport Clock does not provide a reference to physical time, making it impossible to search events by physical time. HLC is a scalar value with a distribution algorithm that allows that value to be bounded to a physical time. [21]

We considered pros and cons of the (a-d) using the following product and user requirements to guide the design decisions:

1. Performance must remain the same for operations that don't need Logical Time.
2. Minimal performance overhead for the events that do leverage Logical Time.
3. The existing tools around MongoDB should remain forward compatible.
4. External clients must be untrusted, must be unable to break the system, or affect its availability.
5. The solution must allow scaling to thousands of shards without noticeable performance degradations.
6. The solution must be resilient to operator errors.
7. The solution must run on off-the-shelf hardware or popular commercial cloud services our users have access to.

Evaluation of option (a): A disadvantage of a scalar logical clock (the Lamport Clock) is that it's disconnected from physical time. With that constraint, popular MongoDB utilities like backup and recovery could not use physical time as a reference. We rejected this option because it goes against our third requirement and would cause breaking changes to tools.

Evaluation of option (b): A Vector clock can distinguish events on different nodes, which improves efficiency in establishing a snapshot, or consistent point-in-time, across multiple shards. However, the size of each message becomes proportional to the number of shards. This does not scale well across numerous shards, breaking our fifth requirement.

Evaluation of option (c): TrueTime is not appropriate for an open source database that can be deployed anywhere, as it requires atomic clocks / GPS to implement tight guaranteed boundaries of clocks. The approaches based on clock synchronization introduce mandatory waits for read or write operations that may become significant. The need for special hardware does not satisfy requirement 7.

Evaluation of option (d): A Hybrid Logical Clock has the same space and computational requirements as scalar logical time. Even prior to causal consistency, MongoDB used ordering similar to hybrid logical time for maintaining order in the operation log used for replication of a single shard. So, we already had some of the foundation needed to synchronize the events across the cluster. This approach was best aligned with our product requirements.

Additionally, we made an important deviation from prior work in our definition of an event. In most papers, starting with [23], an event is the arrival of the message to a node. This helps to abstract the underlying system and consider that any event may have consequences. In MongoDB we can take advantage of being able to detect which messages actually change the system's state and which are simple observers. The logical clock increments only when there are new operation log entries, representing system state change. This makes a connection between the system state and logical time. This approach also significantly simplifies implementation shown below by indexing oplog entries with logical time.

4.2 Dependency Tracking

For dependency tracking, the choices for the engineering team were:

a. Full dependency tracking
b. Explicit dependency tracking
c. Monolithic or Bolt-on
d. Dependency tracking on the client (driver)

We have rejected the approach (a) due to performance overhead caused by the need to attach and process the dependency graph on all events.

For (c), The "Bolt-on Causal Consistency" paper [5] describes an approach to implement a shim that does not keep a durable state and resolves the causal dependencies. That allows existing eventually consistent systems to address causal ordering without changing their implementation. We had two options: "Implement the causal consistency as a separate agent or make it the part of the core functionality of a distributed database".

The approach (d) can potentially be applied to the older versions of the database as it assumes that the server and the protocol did not change. However, the client side implementation requires explicit application support across all client participants, and clients may use any number of drivers across any number of languages. An implementation may require the passing of causality tokens by all participants to establish the *happened-before* relationship and therefore cannot be used universally.

While the layered approach separating availability, liveness, partition tolerance, and scalability (ALPS) [25] with causality sounds very appealing, this approach would have added complexity, overhead, and an additional failure point if implemented as a separate agent. While MongoDB did not implement causal consistency agents or shims it still achieved separate components for data replication, sharding, and causal consistency, i.e. we can change replication protocol and it will not affect the causal consistency code. Likewise, changing the way causal consistency is implemented will not affect sharding or replication. The use of the HLC values as a part of the operation log unique ids is very convenient for implementing causal consistency. It is even possible to experiment with vector hybrid clocks (i.e. attach a node id to each time to provide better partition support) as well as clock synchronization to implement external consistency; it can all be done without changing replication or sharding protocols which is very important in order to provide backward compatibility and rolling upgrades to newer versions.

The idea of making users explicitly track dependencies (b) can be presented in a form of causality segments where all data that a customer reads *happened-after* the latest event observed by the client. The team took approach (b). Assume that the client performed a write and the returned data has been added to a replica A with the causal token T. The follow up read from a replica B by the same client must satisfy the condition that the dataset on a replica B has the write that was committed to replica A. The MongoDB engineering team considered assigning a scalar value of the Hybrid logical clock to each value returned from any server node. A similar approach to dependency tracking was discussed in the [2] but with scalar physical clocks and in [16], but with vector hybrid logical clocks.

As discussed in the "Type of Clock" sub-section MongoDB uses scalar hybrid logical clocks. Each message from each data node in the causally consistent session – i.e. a thread of execution – receives the `operationTime` - the last known logical time persisted in the oplog on that node. Follow-up read or write operations originated in this session attach the highest known `operationTime` to the request metadata. The data node that receives the request waits for its operation log to catch up to the requested `operationTime`.

4.3 Clock Synchronization

In research papers clock synchronization often means gossiping [19], [23], or heartbeats [17], [16]. However, due to the low latency requirements we need an even faster way to synchronize cluster time on all data nodes participating in an operation. Consider, for example, a system with shard X and shard Y. By definition of horizontal partitioning they contain non-overlapping key domains. Since cluster time in MongoDB is scalar, the system must provide a resolution for the scenario when a client writes to the shard X and then reads from the shard Y requesting the `operationTime(X)` that is greater than the latest persisted cluster time on Y. Clock synchronization can be solved by:

a. using vector clock
b. physical clock synchronization
c. using heartbeats
d. forced advance of Stable Cluster Time (SCT) – the logical time persisted in the oplog

Options (a) and (b) were ruled out earlier, making the decision between sending frequent heartbeats and advancing SCT. To advance SCT MongoDB performs a no-op write as this is the only way to increase Stable Cluster Time. It will not represent conflicts because key domains are independent (there is no multi-master) and hence every sequence of read or write operations will have strictly increasing sequence of SCT. Thus, advancing SCT allows providing low latency for all types of workloads limited only by the node's write throughput. MongoDB took approach (d).

4.4 System Security

MongoDB puts security of data and resistance to malicious attacks at the top of its priorities. While using the HLC in the oplog key helped significantly simplify the code while maintaining a layered approach, it also introduced a

significant security risk. The advance SCT algorithm writes to the oplog the latest known logical time across all system participants, even those outside the security perimeter, i.e. the external clients. As SCT values are strictly monotonic and persisted, a malicious client can simply send the maximum possible logical clock value. Once the max time is written to the operation log, the system will not be able to accept future modifications because time can no longer be incremented. There are several approaches that can be used:

a. Sign the values of logical time so they can be only originated by MongoDB data nodes.
b. Use a separate agent that will track the explicitly attached dependencies – i.e. do not embed causal dependencies in the data nodes.

As discussed earlier the approach (b) does not satisfy the requirements to provide optimal performance and reliability/simplicity. Approach (a) was described in the paper [24], protecting replication protocols against byzantine faults. We used similar approach to prevent users from changing messages in order to advance cluster time to the maximum value. However, we can use the same secure key for generating message signature and validation as MongoDB architecture has a centralized data catalog unavailable to non-privileged users. While this design has the benefit of being a server only implementation it increases the system's complexity. The main engineering difficulty is implementing reliable and secure key generation and synchronization as well as making sure it works during partitions, upgrades, downgrades, and multi-version operations.

Even in scenarios where clients do not comply with causal gossiping, for example by sending older HLC values, only the non-complying session is affected. All causal ordering is enforced by the server. If a modification comes in with an old HLC value, the server will use the greater of that or the latest time it has otherwise seen and increment from there.

5 System Model

MongoDB Deployment is a sharded cluster, replica set, or standalone. A standalone is a storage node that represents a single instance of a data store. A replica set that is part of a sharded cluster is not itself a MongoDB deployment. However, a replica set that is not part of any sharded cluster is a MongoDB deployment. A MongoDB deployment may consist of:

- *Storage nodes* also referred as *mongod* that are typically part of replica sets, and therefore may be *primary*, *secondary* or in some other *follower modes* (i.e., rollback, recovering, initial sync).We will describe a sharded cluster deployment where each shard is a replica set comprised of storage nodes.
- *Clients* are application processes that communicate with one or more MongoDB deployments via a *driver*.
- *Routers* also referred as *mongos* that do not hold data and only route *client* commands. A *cluster node* is any router or storage node within a MongoDB deployment.

MongoDB deployments may contain multiple data shards. Each shard represents a horizontally partitioned set of data. A config shard also exists to keep the metadata describing data distribution and database configuration, as data can migrate between shards elastically in MongoDB. Each shard can be a replica set that contains a primary and one or more secondary nodes. At any given time there can be only one primary node that accepts client writes for a set of data, selected by consensus election. However, it's possible to temporarily have multiple nodes that act as primary (though only one is truly primary) during network partitions, where a previous primary has not yet realized it's been partitioned from a majority. Only the writes that were sent to the new primary, elected by consensus, will be persisted. The primary node receives all the client write operations and records all other changes to its data set in its operation log. The secondary nodes replicate the primary's operation log and apply the operations to their data set such that their data set reflects the primary's data set. If the primary is unavailable, an eligible secondary will call an election to elect the new primary.

Each write request contains or implies a write concern [32]. Write concern specifies when a write can be acknowledged to the client. One possible write concern value is "majority". Majority writes guarantee that the write will be acknowledged only after the data has been persisted by the majority of voting nodes including the primary. Majority writes are durable and even in the case of partitions will not be lost.

Although clients cannot write data to secondaries, clients can read data from secondary members. Clients can specify a read concern [31] to control the consistency and isolation properties of the data read. Among others, the read concern level "majority" allows clients to query only majority committed data. The returned documents are durable even in the event of system failures.

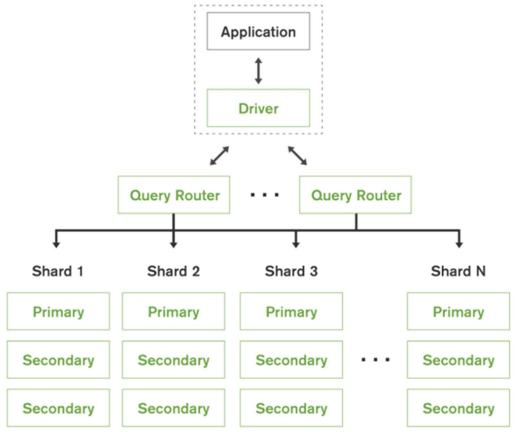

Figure 1. MongoDB architecture.

Every storage node in a replica set is a state machine that applies changes from the operation log. The operation log entries are created on the primary node and replicated to the secondary nodes sometime thereafter. While the secondary node will eventually have up-to-date data, its state may lag behind the primary. The entries in the operation log are ordered by a timestamp-based structure that can be used to determine the order in which events occurred.

The timestamp is based on the primary's wall-clock time. MongoDB does not provide enforcement of clock synchronization between nodes in the deployment. Nodes use the operating system provided clock synchronization, which is in many cases based on the popular NTP protocol [28]. The accuracy of node's wall clocks is subject to an internet connection latency, strata of the time server chosen for synchronization and the distance from the time-servers.

The detailed description of algorithms, protocols and data structures of Causal Consistency implementation can be found in the Appendix.

6 Using Causal Consistency

MongoDB powered applications have been around for several years before the introduction of causal consistency support. Prior to causal consistency user applications often were using the primary node in a replica set for writes and reads to increase the likelihood of observing data consistently. This approach works in the absence of elections that may cause split-brain because MongoDB totally orders all writes in the operation log. Occasional non-causally consistent reads and commands can be tolerated by most applications because elections are rare: our statistics for Atlas [29] – our managed cloud MongoDB offering - shows 1 election per replica set per 50 days (including maintenance).

However, reading from the primary only affects application scalability especially for workloads that are mostly reads or geo-distributed. While the geo - distributed writes can be addressed with zone sharding [34], the local reads need causal consistency in order to be causally consistent with the writes. To address scalability and performance requirements in the environments that cannot tolerate occasional staleness of data users were implementing application side logic to ensure that the data read from secondaries is not stale.

In this section we will overview how some of those application use MongoDB features and how adding causal consistency allows them to improve and expand functionality and service. MongoDB enables causal consistency in client sessions. Each session where causal consistency is enabled tracks signed `clusterTime` – the highest known logical time and `opearationTime` - the logical time of the last operation's causal snapshot. The `operationTime` must be no greater than the `clusterTime`. MongoDB provides an API to pass those values between sessions to enable clients to extend causally consistent chain of operations across multiple sessions, or even clients. Because causal consistency provides users with guarantees that all operations are causal it allows making causally consistent reads from secondary nodes that could be useful in some scenarios including geo-replicated low-latency reads.

7 Performance Optimization

One of the key tests in the performance measurement by engineering team for a new release is making sure that the new features did not cause unacceptable slowdown of the system performance. Specifically, we want to confirm that performance for legacy workloads and workloads with causal consistency turned off will not be affected by measuring an overhead to message processing for signature generation and validation. To isolate these codepaths we used code instrumentation with instruction counters to quantify the performance impact directly.

7.1 Measuring and Optimizing Signature Generation and Verification Overhead

Generating a signature to protect ClusterTime from being changed by non-authorized users added performance overhead with each message. The measurements were determining the exact value of this overhead. The tests were run on a MongoDB 3.6.0 3-node replica set and measured

cycles to generate the signature via `rdtscp` CPU instruction.

Table 1. Signature generation CPU cost

N inserts	N calls	Min (000)	Max (000)	Average (000)
100	103	13	92	22
10,000	10,003	8	98	8
1,000,000	1,000,003	8	1,284	8

To compare here is the bare insert measurements:

Table 2. Bare insert CPU cost

N inserts	N calls	Min (000)	Max (000)	Average (000)
10,000	10,003	130	351,400	223

Hence the signature generation may represent up to 10% of the workload. To mitigate the overhead, we implemented several optimizations.

The first path was to disable signing and verifying when it was not needed: where there are no untrusted clients - i.e. the implementation is inside an authentication perimeter. For those cases we added a system privilege `advanceCluserTime`. Messages to and from users with this privilege can skip the signing and verification process by using dummy signatures.

The second optimization path was to sign a range of time, so not every increment needs to be verified. As the cluster time grows linearly, we can always mask the least significant bits in the representation to '1', sign it, and cache the signature. Chances are, the next message will have the ClusterTime value incremented by 1, resulting in the same masked value. Then, *mongod* can reuse the cached signature.

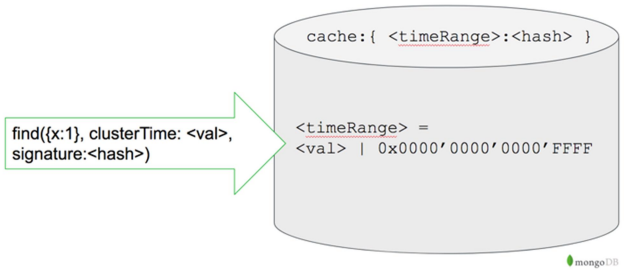

Figure 2. Signing Time Range.

In our experiments, this approach decreased the CPU load needed for signature generation and verification by a few hundred times.

Table 3. Signature generation CPU Cost with caching

N inserts	N calls	Min (000)	Max (000)	Average (000)
100,000	30	25	56	31

7.2 Results Discussion

The results demonstrate that Causal Consistency signature generation and verification overhead has a negligible cost to the existing applications which are dominated by reading and writing from primary host pattern that delivers probable causal order of operations.

8 Functionality Testing

Besides malicious users that can modify the message the system must provide correct results in the presence of failovers, rolling upgrades, data balancing and encryption key rotations that happen automatically. The failure scenarios must be verified on every platform where the product ships and since MongoDB receives plenty of code changes every day the tests must be repeated regularly. Those functionality tests are run with MongoDB open source continuous integration system Evergreen [30].

MongoDB already had implemented thousands of tests to verify functionality developed over the years and running with causal consistency is not expected to change their outputs. Hence, we had run relevant subset of the tests in the causally consistent sessions while reads are forced to be done on the secondary. The test infrastructure allows to run the same test scripts in the different environments. In particular we have a test variant that runs tests during the continuous failover. The new functionality, such as encryption key auto rotations had a separate test script developed and included in the evergreen runs.

One of the benefits of an established test base is to be able to run tests early on: while working on the proof of concept implementation. This way we were able to validate design decisions during the design phase.

9 Performance Evaluation

This section presents an evaluation of performance cost of using Causal Consistency under various conditions. First, we establish a baseline of system performance on a particular workload and then measure the impact of enabling causal consistency in combination with other settings, showing how this impacts performance and scalability. Last, we show how users can increase throughput of their application by increasing the number of

nodes in the cluster and safely reading from secondaries with causal consistency enabled.

9.1 Implementation and Experimental Setup

For our hardware we provisioned a single replica set in AWS us-east-1 region via MongoDB Atlas [29]. Each node had 15.25GBs of RAM, 2 vCPUs and extremely low latency NVMe based SSD storage. While all nodes were in the same geographical region they were in different availability zones. We chose relatively small instance sizes to keep the costs low for anyone who would like to reproduce our results, and to make it easy to see the impact of relatively small variations in workload setup. We varied the number of nodes in the replica set from 3 to 5. For our client we used a single r4.8xlarge instance (32 vCPU and 244GBs of RAM) in the same AWS region.

9.2 Workload

For testing causal consistency we had to create a workload which would perform multiple sequential operations in a single logical session rather than independent point queries, so we used a workload loosely based on the type of commerce related operations used in TPC-C benchmark [38] which involve each client doing a mix of reads and writes (orders, deliveries, payments) or just a sequence of reads (stock level checks, order status check) in a session. Performance tests were done without enabling MongoDB transactions, but using logical sessions, and varying settings for causal consistency, read preference (primary or nearest), write concern, as well as the number of client threads.

9.3 Benchmarks

First the database was pre-populated with data generated by TPC-C benchmark for 100 warehouses which produced around 8.5GBs of compressed data on disk, translating to 12.5GBs of data in memory. We then ran variations of operations both with more typical TPC-C mix of read and write operations which we measured by recording TpmC as throughput, as well as with read only operations where we measured reads per second. Each combination of configuration parameters was run for five minute durations with increasing number of threads till the throughput started decreasing or latency became prohibitively high. Aside from comparing causal consistency on vs off we also wanted to understand the relative performance of the typical settings our users choose for their applications.

9.4 Causal Consistency with Nondurable Writes

Many applications that demand low latency writes cannot wait for acknowledgement of replication of each write and use w:1 write concern, meaning some of the writes may be rolled back if there is a failover. They likewise require low latency reads so sending read request to nearest node, primary or secondary is frequently their desired configuration, even though it may result in application seeing inconsistent view of the data. We compared the impact of turning on causal consistency in such a configuration. Our experiments showed that while overall number of operations was lower, the impact was less pronounced at higher number of threads.

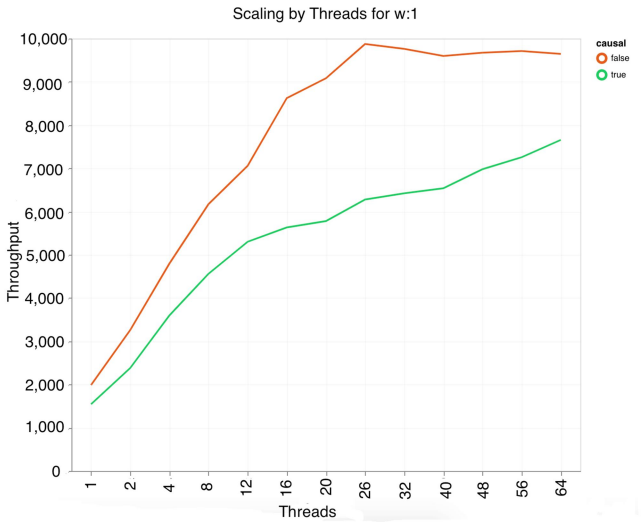

Figure 3. Scaling by Threads Non-durable Writes.

9.5 Causal Consistency with Durable Writes

On the other hand, applications that cannot tolerate inconsistent view of the data are more likely to require that data be durable across majority of the replica set before a write is considered successful. The higher latency for majority write concern setting can be mitigated by increasing threads, but reading from secondaries without causal consistency can still return inconsistent data. Our experiments show that with write concern majority, enabling causal consistency had minimal effect on overall throughput and would likely be acceptable setting to consider for many of the users.

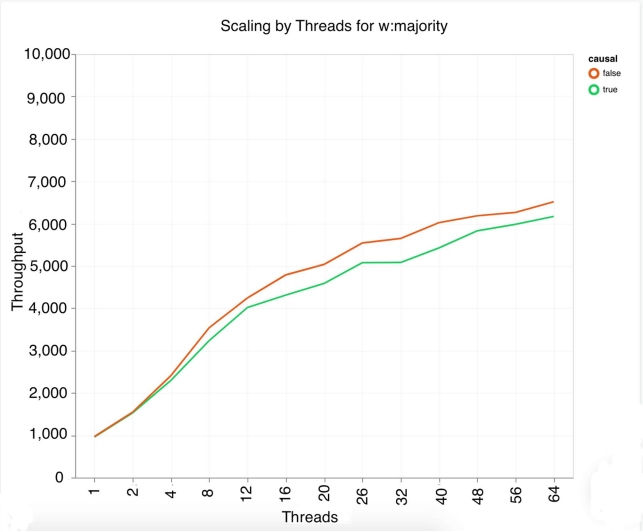

Figure 4. Scaling by Threads Durable Writes.

9.6 Scaling by Number of Nodes

We wanted to test the assumption that the number of reads the system can handle would go up with the number of available nodes to read from in the cluster so we ran similar test for read only workloads and as expected saw the overall reads per second go up almost linearly with the number of total nodes in the replica set and read preference nearest.

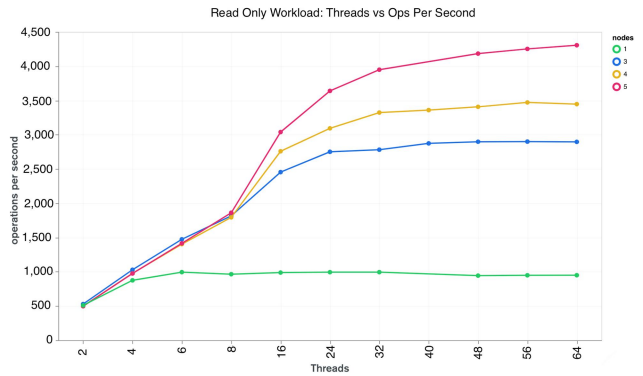

Figure 5. Scaling by Secondary Nodes.

10 Future Work

Implementation of the cluster-wide logical clock provides the basis for many features that require cluster-wide data ordering. For example, change streams [33] resumability is based on ClusterTime.

In the longer term our team works on building the multi-document ACID transactions. The Cluster Time allows to establish a consistent logical time across all cluster operations. This in turn is needed to build cluster-wide snapshot isolation which is an important step towards the distributed transactions delivery.

As cloud providers expose higher quality time sources for synchronizing process wall clocks [3], another interesting avenue of future work is to utilize better-synchronized clocks to eliminate stalls and excess writes in the implementation of causal consistency. The extent to which these synchronization services allow third-party application developers to implement systems similar to Google's TrueTime [14] is also an interesting question in practice.

11 Conclusion

MongoDB combined ideas from the research community into a secure production-grade implementation that supports causal consistency. The implementation adds a linear processing overhead, tolerates possible operator errors, and provides partial causal event ordering in a multisharded, replicated, distributed database.

One aspect of our design stands out: the protection against non-trusted clients attacks by incrementing the clusterTime on the server only and signing it with a secure key. The implementation guarantees the liveness of the protocol by generating a noop write on a primary to advance the logical clock persisted value.

ACKNOWLEDGMENTS

Many people have helped. Grigori Melnik, Blake Oler, and Natalie Tsvetkova provided timely and helpful feedback to improve this paper.

Spencer Jackson and Andreas Nilsson shared novel ideas for signing and optimizing the signature generation process of the Cluster Time.

We were working closely with MongoDB engineers implementing the driver and client sessions support. Among those Jason Carey, Samantha Ritter, Jesse Jiryu Davis, Jeff Yemin, Bernie Hackett, David Golden, and Robert Stam. Special thanks to Max Hirschhorn for Mongo shell support and integration tests review and feedback.

During the design and implementation those MongoDB engineers helped to define the scope and provided an important design feedback: Eliot Horowitz, Dan Pasette, Kal Manassiev, Spencer Brody, Eric Milkie, David Storch, Steve Briskin, Esha Maharishi, Dianna Hohensee, William Schultz, Judah Schvimer, Benety Goh, Charlie Swanson, Tess Avitabile, Andrew Morrow, Geert Bosch. Many thanks to Mathias Stearn in particular for sharing constructive ideas on improving the code.

Thanks to Kay Kim for writing the documentation and giving the feedback along the way.

APPENDIX

A1 Implementation of Cluster-Wide Logical Clock

A1.1 Logical Clock Design

The term "ClusterTime" refers to the time value of a node's logical clock. As discussed in the Design Choices section we went with the Hybrid Logical Clock design. Despite similar data structure for the logical clock, there are several differences in the logical clock distribution algorithm that allowed us to mitigate malicious attacks intended to break the database or mis-programmed clients. MongoDB only increments time when state changing "events" occur. Sending and receiving messages are not events as they do not change the state of the nodes, while all writes against MongoDB are time incrementing events.

The difference between events and non-events allows us to separate the concepts of distributing and incrementing ClusterTime. Each shard's primary node's process is always responsible for implementing state changes in MongoDB. Clients only participate in non-events (the sending and receiving of messages), making it unnecessary for clients to increment ClusterTime. That said, clients maintain responsibility for distributing the greatest ClusterTime with each message they send or receive.

MongoDB's ClusterTime adheres to the following rules:

ClusterTime Increment rule:

The ClusterTime is incremented ("ticks") only when there is a write to a primary node's replication operation log (oplog).

ClusterTime Distribution rule:

Cluster nodes (mongod, mongos, config server, clients) always track and include the greatest known ClusterTime when sending a message.

A1.2 ClusterTime Increment

ClusterTime is represented by a `<Time><Increment>` pair: where `<Time>` is a 32 bit count of seconds since the Unix epoch (physical time) and `<Increment>` is a 32 bit integer that allows us to distinguish writes that occurred within the same second. Every node in the cluster has a LogicalClock that keeps an in-memory version of the node's ClusterTime. ClusterTime can be converted to the OpTime, the time stored in MongoDB's replication oplog. OpTime can be used to identify entries and corresponding events in the oplog. OpTime is represented by a `<Time><Increment><ElectionTerm>` triplet. Here, the `<ElectionTerm>` is specific to the MongoDB replication protocol. It is local to the replica set and not a global state that should be included in the ClusterTime.

To associate the ClusterTime with an oplog entry when events occur, MongoDB first computes the next ClusterTime value on the node, and then uses that value to create an OpTime (with the election term). This OpTime is what's written to the oplog. This update did not require the OpTime format to change, remaining tied to a physical time. All the existing tools that use the oplog, such as backup and recovery remain forward compatible.

The ClusterTime "ticks" with this algorithm [35]

```
ClusterTime getNextClusterTime() {
newCounter = 0;
wallClockSecs = now();
// _clusterTime is a current local
// value of node's ClusterTime
currentSecs = _clusterTime.getSecs();

if (currentSecs > wallClockSecs) {
   newSecs = currentSecs;
   newCounter =
        _clusterTime.getCounter() + 1;
}
else {
   newSecs = wallClockSecs;
}

_clusterTime =
    ClusterTime(newSecs, newCounter);
return _clusterTime;
}
```

A1.3 ClusterTime Distribution

Every node keeps track of the maximum value of ClusterTime it has ever seen [36]. Every node adds this value to each message that it sends.

Let's illustrate how ticking and gossiping work together on a multi-shard system. The "ticking" may be done only on the data nodes that can write (the primaries of each shard). Advancing may happen on all nodes, including client nodes,

where advancing is simply updating to a new maximum value of ClusterTime the node is aware of.

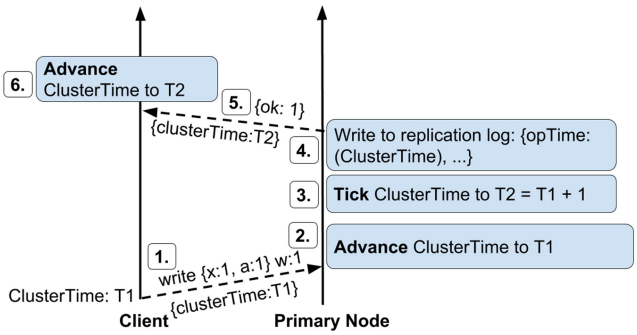

Figure 6. ClusterTime Increment and Distribution.

1. Client sends a write command to the primary, the message includes its current value of the ClusterTime: T1.
2. Primary node receives the message and advances its ClusterTime to T1, if T1 is greater than the primary node's current ClusterTime value.
3. Primary node "ticks" the cluster time to T2 in the process of preparing the OpTime for the write. This is the only time a new value of ClusterTime is generated.
4. Primary node writes to the oplog.
5. Result is returned to the client, it includes the new ClusterTime T2.
6. The client advances its ClusterTime to T2.

A1.4 Protecting Against Malicious Attacks

As shown before, nodes advance their logical clocks to the maximum ClusterTime that they receive in the client messages. The next oplog entry will use this value in the timestamp portion of the OpTime. But a malicious client could modify their maximum ClusterTime sent in a message. For example, it could send the `<greatest possible cluster time - 1>` . This value, once written to the oplogs of replica set nodes, will not be incrementable and the nodes will be unable to accept any changes (writes against the database). The only way to recover from this situation would be to unload the data, clean it, and reload back with the correct OpTime. This malicious attack would take the affected shard offline, affecting the availability of the entire system. To mitigate this risk, MongoDB added a HMAC-SHA1 signature that is used to verify the value of the ClusterTime on the server. ClusterTime values can be read by any node, but only MongoDB processes can sign new values. The signature cannot be generated by clients. Here is an example of the document that distributes ClusterTime:

```
"$clusterTime" : {
      "clusterTime" :
          Timestamp(1495470881, 5),
      "signature" : {
        "hash" : BinData(0,

"7olYjQCLtnfORsI9IAhdsftESR4="),
        "keyId" : "6422998367101517844"
      }
}
```

The `keyId` is used to find the key that generated the hash. The keys are stored and generated only on MongoDB processes. This seals the ClusterTime value, as time can only be incremented on a server that has access to a signing key. Every time the *mongod* or *mongos* (MongoDB server data node or query router) receives a message that includes a ClusterTime that is greater than the value of its logical clock, they will validate it by generating the signature using the key with the `keyId` from the message. If the signature does not match, the message will be rejected.

A1.5 Handling Operator Errors

The risk of malicious clients affecting ClusterTime is mitigated by a signature, but it is still possible to advance the ClusterTime to the "end of time" by changing the wall clock value. This may happen as a result of operator error. Once the data with the OpTime containing the "end of time" timestamp is committed to the majority of nodes it cannot be changed. To mitigate this, we implemented a limit on the rate of change. The ClusterTime on a node cannot be advanced more than the number of seconds defined by the `maxAcceptableLogicalClockDriftSecs` parameter (default value is one year).

A2 Implementation of Causal Consistency

A2.1 Returning the OperationTime from all Operations

When a write event is sent from a client, that client has no idea what time is associated with the write, because the time was assigned after the message was sent. But the node that processes the write does know, as it incremented its ClusterTime and applied the write to the oplog. To make the client aware of the write's ClusterTime, it will be

included in the `operationTime` field of the response. To make sure that the client knows the time of all events, every response (including errors) will include the `operationTime` field, representing the Stable Cluster Time – i.e. the ClusterTime of the latest item added to the oplog at the time the command was executed.

Now, to make the follow up read causally consistent the client will pass the exact time of the data it needs to read - the received `operationTime` - in the `afterClusterTime` field of the request. The data node needs to return data with an associated ClusterTime greater than or equal to the requested `afterClusterTime` value.

Here is an illustration on how "Read your own Write" works in a replica set:

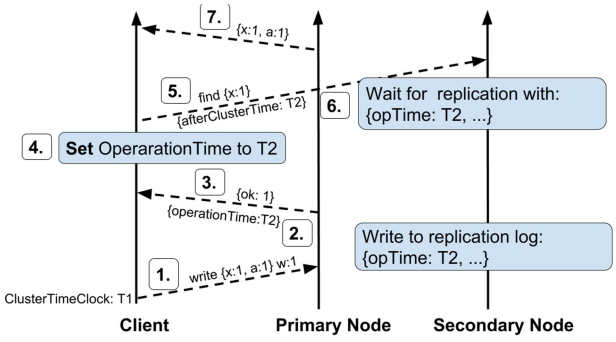

Figure 7. Returning an OperationTime.

1. The client sends a write to the primary. The "w:1" means that the primary will not wait for the data to be replicated to a secondary, and will return as soon as its committed to the oplog.
2. The primary computes the ClusterTime, and ticks it as described in the previous sections.
3. The primary returns the result with the ClusterTime value stored in the OpTime, as the operationTime field of the response.
4. The client conditionally updates its local value of the `lastOperationTime` with the returned operationTime value.
5. The client sends a read to a secondary node. To be sure that it can "read your own write" it includes the `afterClusterTime` field in the request and passes the operationTime value it received from the write.
6. The secondary checks if the data with the requested operationTime is in its oplog. If not, it waits until it's replicated.
7. The result is returned to the client.

A2.2 Progressing Time in the Presence of no State Changes

The scenario in the previous example has not described the behavior when the client writes to one shard and then performs a read that goes to a shard that has its own logical clock behind the requested time. If there are no new writes, the read will stall. In this case, the *mongod* will force a write to ensure that clients will not stall forever.

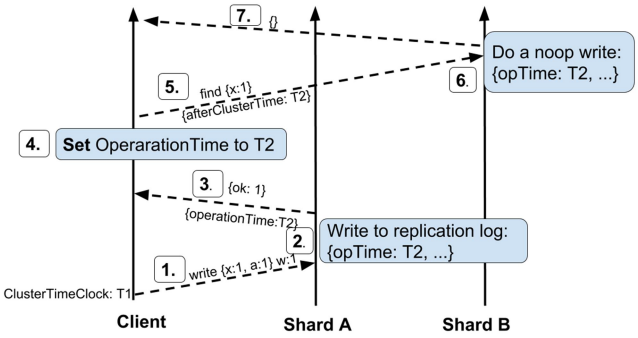

Figure 8. Logical Clock Synchronization in a Cluster.

1. The client sends a write to Shard A.
2. The primary on Shard A computes the ClusterTime, and ticks as described in the previous sections.
3. Shard A returns the result with the operationTime that was written to the oplog.
4. The client conditionally updates its local lastOperationTime value with the returned operationTime value
5. The client sends a read to Shard B. To be sure that it can "read your own write" it includes the `afterClusterTime` field in the request and passes the operationTime value it received from the write
6. Shard B checks if the data with the requested OpTime is in its opog. If not, it performs a noop write.
7. The result is returned to the client (in this example it's empty because the data it searches for is not on Shard B).

REFERENCES

[1] M. Ahamad , G. Neiger , J. E. Burns et al: Causal Memory: Definitions, Implementation and Programming. Distributed Computing (1995) 9: 37. DOI: https://doi.org/10.1007/BF01784241.

[2] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguica, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), volume 00, pages 405–414, June 2016.

[3] Amazon. Keeping Time With Amazon Time Sync Service. https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service

[4] Apache, SOLR 7.6.0 http://lucene.apache.org/solr/, 2019.

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, I. Stoica Bolt-on Causal Consistency. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 761-772

[6] P. Bailis and A. Ghodsi. Eventual Consistency today: Limitations, extensions, and beyond. ACM Queue, 11(3), 2013

[7] P. Bailis, K. Kingsbury. The Network is Reliable. ACM queue, 12(7), July 23, 2014.

[8] P. Bailis at al. Highly Available Transactions: Virtues and Limitations, Proceedings of the VLDB Endowment Volume 7 (3): 181-192, November 2013.

[9] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. PVLDB, 5(8):776–787, 2012.

[10] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In SOCC 2012.

[11] Basho, RIAK KV, http://basho.com/products/riak-kv/, 2019.

[12] D. R. Cheriton, D. Skeen. Understanding the limitations of causal and totally ordered multicast. In Proceedings of the 14th Symposium on Operating System Principles (SOSP '93, Asheville, NC, 1993). 44- 57.

[13] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. Proc. VLDB Endow., 1(2):1277–1288, Aug. 2008.

[14] J. C. Corbett, J. Dean, M. Epstein et al. Spanner: Google's Globally-Distributed Database, In Proceedings of OSDI 2012.

[15] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. pp. 205-222.

[16] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel. Causal consistency and latency optimality: Friend or foe?. Technical Report 256091, EPFL, July 2018.

[17] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

[18] C. J. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. Australian Computer Science Communications, 10(1):56-66, February 1988.

[19] S. Gilbert, N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News, 33(2):51–59, 2002.

[20] M. Kleppmann, "A critique of the CAP theorem," 2015, http://arxiv.org/abs/1509.05393.

[21] S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In Principles of Distributed Systems, volume 8878:17–32, 2014.

[22] A. Lakshman P. Malik. Cassandra - A Decentralized Structured Storage System ACM SIGOPS Operating Systems Review, 44(2): 35-40, April 2010.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, July 1978.

[24] S. Liu, P. Viotti, C. Cachin et al. XFT: Practical Fault Tolerance Beyond Crashes. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16), Savannah, GA, November 2016.

[25] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In 23rd ACM Symposium on Operating Systems Principles (SOSP), pages 401–416, October 2011.

[26] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, UT Austin, May 2011.

[27] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! Scalable causal consistency with no slowdown cascades. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17), pages 453–468, Boston, MA, 2017. USENIX.

[28] D. Mills. A brief history of ntp time: Memoirs of an internet timekeeper. ACM SIGCOMM Computer Communication Review, 33(2):9–21, 2003.

[29] MongoDB Atlas, https://www.mongodb.com/cloud/atlas, 2019.

[30] MongoDB Evergreen: https://evergreen.mongodb.com, 2019.

[31] MongoDB Manual 3.6. ReadConcern https://docs.mongodb.com/v3.6/reference/read-concern/.

[32] MongoDB Manual 3.6. WriteConcern. https://docs.mongodb.com/v3.6/reference/write-concern/.

[33] MongoDB Manual 3.6 Change Streams. https://docs.mongodb.com/v3.6/changeStreams/.

[34] MongoDB Manual 3.6 Zone Sharding. https://docs.mongodb.com/v3.6/tutorial/manage-shard-zone/.

[35] MongoDB source code: https://github.com/mongodb/mongo/blob/r3.7.2/src/mongo/db/logical_clock.cpp#L114-L158.

[36] MongoDB source code: https://github.com/mongodb/mongo/blob/r3.7.2/src/mongo/db/logical_clock.cpp#L99-L112.

[37] K. Patella, MongoDB 3.6.4 http://jepsen.io/analyses/mongodb-3-6-4.

[38] TPC-C benchmark, http://www.tpc.org/tpcc/, 2019.