Tunable Consistency in MongoDB

William Schultz
MongoDB, Inc.
1633 Broadway 38th floor
New York, NY 10019
william.schultz@mongodb.com

Tess Avitabile
MongoDB, Inc.
1633 Broadway 38th floor
New York, NY 10019
tess.avitabile@mongodb.com

Alyson Cabral
MongoDB, Inc.
1633 Broadway 38th floor
New York, NY 10019
alyson.cabral@mongodb.com

ABSTRACT

Distributed databases offer high availability but can impose high costs on client applications in order to maintain strong consistency at all times. MongoDB is a document oriented database whose replication system provides a variety of consistency levels allowing client applications to select the trade-offs they want to make when it comes to consistency and latency, at a per operation level. In this paper we discuss the tunable consistency models in MongoDB replication and their utility for application developers. We discuss how the MongoDB replication system's speculative execution model and data rollback protocol help make this spectrum of consistency levels possible. We also present case studies of how these consistency levels are used in real world applications, along with a characterization of their performance benefits and trade-offs.

PVLDB Reference Format:
William Schultz, Tess Avitabile, Alyson Cabral. Tunable Consistency in MongoDB. *PVLDB*, 12(12): 2071 - 2081, 2019.
DOI: https://doi.org/10.14778/3352063.3352125

1. INTRODUCTION

Distributed databases present a wide variety of implementation and usability challenges that are not present in single node database systems. Weak consistency models, partial failure modes, and network partitions are examples of challenges that must be understood and addressed by both the application developers and the system designers. One of the main goals of the MongoDB replication system is to provide a highly available distributed data store that lets users explicitly decide among the trade-offs available in a replicated database system that are not necessary to consider in single node systems.

The gold standard of consistency for concurrent and distributed systems is *linearizability* [8], which allows clients to treat their system as if it were a single, highly available node. In practice, guaranteeing linearizability in a distributed context can be expensive, so there is a need to offer relaxed consistency models that allow users to trade correctness for performance. In many cases, applications can tolerate short or infrequent periods of inconsistency, so it may not make sense for them to pay the high cost of ensuring strong consistency at all times. These types of trade-offs have been partially codified in the PACELC theorem, an extension to the CAP theorem [1]. For example, in a replicated database, paying the cost of a full quorum write for all operations would be unnecessary if the system never experienced failures. Of course, if a system never experienced failures, there would be less need to deploy a database in a replicated manner. Understanding the frequency of failures, however, and how this interacts with the consistency guarantees that a database offers motivates MongoDB's approach to tunable consistency.

To provide users with a set of tunable consistency options, MongoDB exposes *writeConcern* and *readConcern* levels, which are parameters that can be set on each database operation. *writeConcern* specifies what durability guarantee a write must satisfy before being acknowledged to a client. Higher write concern levels provide a stronger guarantee that a write will be permanently durable, but incur a higher latency cost per operation. Lower write concern levels reduce latency, but increase the possibility that a write may not become permanently durable. Similarly, *readConcern* determines what durability or consistency guarantees data returned to a client must satisfy. Stronger read concern levels provide stronger guarantees on returned data, but may increase the likelihood that returned data is staler than the newest data. Stronger read concerns may also induce higher latency to wait for data to become consistent. Weaker read concerns can provide a better likelihood that returned data is fresh, but at the risk of that data not yet being durable.

Read and write concern levels can be specified on a per-operation basis, and the usage of a stronger consistency guarantee for some operations does not impact the performance of other operations running at lower consistency levels. This allows application developers to decide explicitly on the performance trade-offs they want to make at a fine level of granularity. Applications that can tolerate rare but occasional loss of writes can utilize low *writeConcern* levels and aren't forced to continuously pay a high latency cost for all of their writes. In the absence of failures or network partitions, writes should eventually become durable, so clients can be confident that most of their writes are not lost. When failures do occur, they can employ other mechanisms to detect and reconcile any window of lost writes. If failures are relatively rare, these mechanisms can be a small

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352125

cost to pay in return for greatly improved common case application performance. For highly sensitive applications that cannot tolerate such inconsistencies, they can choose to operate at higher write and read concern levels to be always guaranteed of safety. Furthermore, applications may choose to categorize some operations as critical and some as non-critical, and thus set their write and read concern levels appropriately, per operation.

With the addition of multi-document transactions, MongoDB added a *readConcern* level which provides transactions with snapshot isolation guarantees. This read concern level offers speculative behavior, in that the durability guarantee of any data read or written is deferred until transaction commit time. The transaction commit operation accepts a write concern, which determines the durability guarantees of the transaction and its constituent read and write operations as a whole. The details of transaction consistency guarantees will be discussed in more detail in later sections. Furthermore, MongoDB introduced causal consistency in version 3.6 which provides clients with an additional set of optional consistency guarantees [15]. A full discussion of causal consistency in MongoDB is out of scope of this paper, but when combined with various read and write concern levels, it allows users to tune their consistency guarantees to an even finer degree.

In Section 3, we will discuss the details of write and read concern levels, and how real world deployments utilize these options. In Section 4, we will compare MongoDB's offerings to those of other commercial databases. In Sections 5, 6, and 7, we will discuss the implementation details of MongoDB's replication protocol that allow for these various consistency levels. In Section 8, we will present performance evaluations of different consistency levels and how they perform in the face of failures.

2. BACKGROUND

MongoDB is a NoSQL, document oriented database that stores data in JSON-like objects. All data in MongoDB is stored in a binary form of JSON called BSON [3]. A MongoDB database consists of a set of collections, where a collection is a set of unique documents. MongoDB utilizes the WiredTiger storage engine, which is a transactional key value data store that manages the interface to a local, durable storage medium. Throughout this paper, we will refer to a transaction at this storage engine layer as a "WiredTiger transaction". To provide high availability, MongoDB provides the ability to run a database as a *replica set*, which is a set of MongoDB nodes that act as a consensus group, where each node maintains a logical copy of the database state. MongoDB replica sets employ a leader based consensus protocol that is similar to the Raft protocol [11]. In a replica set there exists a single primary and a set of secondary nodes. The primary node accepts client writes and inserts them into a replication log known as the *oplog*. The oplog is a logical log where each entry contains information about how to apply a single database operation. Each entry is assigned a timestamp; these timestamps are unique and totally ordered within a node's log. Oplog entries do not contain enough information to undo operations. The oplog behaves in almost all regards as an ordinary collection of documents. Its oldest documents are automatically deleted when they are no longer needed, and

```
{
  // The oplog entry timestamp.
  "ts": Timestamp(1518036537, 2),
  // The term of this entry.
  "t":  NumberLong("1"),
  // The operation type.
  "op": "i",
  // The collection name.
  "ns": "test.collection",
  // A unique collection identifier.
  "ui": UUID("c22f2fe6..."),
  // The document to insert.
  "o":{
      "_id": ObjectId("5a7b6639176928f52231db8d"),
      "x": 1
  }
}
```

Figure 1: Example of key oplog entry fields for an "insert" operation

new documents get appended to the "head" of the log. Secondary nodes replicate the oplog, and then apply the entries by executing the included operation to maintain parity with the primary. In contrast to Raft, replication of log entries in MongoDB is "pull-based", which means that secondaries fetch new entries from any other valid primary or secondary node. Additionally, nodes apply log entries to the database "speculatively", as soon as they receive them, rather than waiting for the entries to become majority committed. This has implications for the truncation of oplog entries, which will be discussed in more detail in Section 6.

Client writes must go to the primary node, while reads can go to either the primary or secondary nodes. Clients interact with a replica set through a *driver*, which is a client library that implements a standard specification for how to properly communicate with and monitor the health of a replica set [10]. Internally, a driver communicates with nodes of a replica set through an RPC like protocol that sends data in BSON format. For horizontal scaling, MongoDB also offers sharding, which allows users to partition their data across multiple replica sets. The details of sharding will not be discussed in this paper.

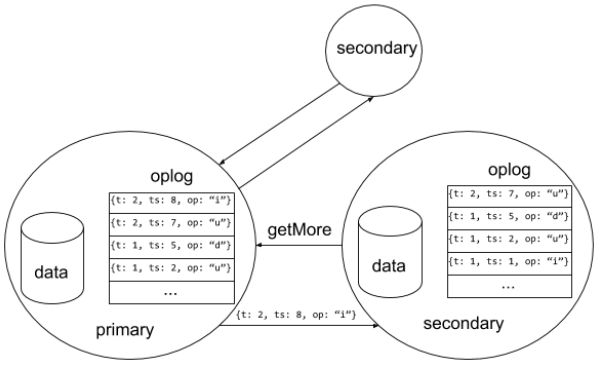

Figure 2: MongoDB Replication Architecture

3. CONSISTENCY LEVELS IN MONGODB

The consistency levels in MongoDB replica sets are exposed to clients via *readConcern* and *writeConcern* levels, which are parameters of any read or write operation, respectively. Understanding the semantics of read and write concern requires some understanding of the lifecycle of operations in MongoDB's replication system. The MongoDB replication system serializes every write that comes into the system into the oplog. When an operation is processed by a replica set primary, the effect of that operation must be written to the database, and the description of that operation must also be written into the oplog. Note that all operations in MongoDB occur inside WiredTiger transactions. When an operation's transaction commits, we call the operation *locally committed*. Once it has been written to the database and the oplog, it can be replicated to secondaries, and once it has propagated to enough nodes that meet the necessary conditions, the operation will become *majority committed* which means it is permanently durable in the replica set.

3.1 Definitions

writeConcern can be specified either as a numeric value or as "majority". A client that executes a write operation at *w:1* will receive acknowledgement as soon as that write is locally committed on the primary that serviced the write. Write operations done at *w:N* will be acknowledged to a client when at least N nodes of the replica set have received and locally committed the write. Clients that issue a *w:"majority"* write will not receive acknowledgement until it is guaranteed that the write operation is majority committed. This means that the write will be resilient to any temporary or permanent failure of any set of nodes in the replica set, assuming there is no data loss at the underlying OS or hardware layers. For a *w:"majority"* write to be acknowledged to a client, it must have been locally committed on at least a majority of nodes in the replica set. Write concern can also accept a boolean *"j"* parameter, which determines whether the data must be journaled on replica set nodes before it is acknowledged to the client. Write concern can also be specified as a "tag set", which requires that a write be replicated to a particular set of nodes, designated by a pre-configured set of "tagged" nodes. The *"j"* and tag set options will not be discussed in detail in this paper.

Client operations that specify a write concern may receive different types of responses from the server. These write concern responses can be classified into two categories: *satisfied* and *unsatisfied*. A write concern that is *satisfied* implies that the necessary (or stronger) conditions must have been met. For example, in the case of *w:2*, the client is guaranteed that the write was applied on at least 2 servers. For a write concern that is *unsatisfied*, this does not necessarily imply that the write failed. The write may have been replicated to fewer servers than needed for the requested write concern, or it may have replicated to the proper number of servers, but the primary was not informed of this fact within a specified operation time limit.

readConcern determines the durability and, in some cases, the consistency of data returned from the server. For a read operation done at *readConcern* level *"local"*, the data returned will reflect the local state of a replica set node at the time the query is executed. There are no guarantees that the data returned is majority committed in the replica set, but it will reflect the newest data known to a particular node, i.e. it reads any locally committed data. Reads with *readConcern* level *"majority"* are guaranteed to only return data that is majority committed. For majority reads, there is no strict guarantee on the recency of the returned data. The data may be staler than the newest majority committed write operation. MongoDB also provides "linearizable" readConcern, which, when combined with *w:"majority"* write operations provides the strongest consistency guarantees. Reads with *readConcern* level *"linearizable"* are guaranteed to return the effect of the most recent majority write that completed before the read operation began. More generally, writes done at "majority" and reads done at "linearizable" will collectively satisfy the linearizability condition, which means the operations should externally appear as if they took place instantaneously at some moment between the invocation of the operation and its response.

Additionally, MongoDB provides "available" and "snapshot" read concern levels, and the ability for causally consistent reads. The "snapshot" read concern only applies to multi-document transactions, and guarantees that clients see a consistent snapshot of data i.e. snapshot isolation. The guarantees provided by "available" read concern depend on some sharding specific details, so will not be discussed here. Causal consistency provides the ability for clients to get session guarantees [14], including read-your-writes behavior in a given session.

3.2 A Comparison with ANSI SQL Isolation Levels

In classic, single node database systems, durability of a particular transaction is determined by whether or not a transaction has "committed", which traditionally means the corresponding write has been written to a journal whose data has been flushed to disk. This gives rise to the meaning of the READ COMMITTED and READ UNCOMMITTED SQL isolation levels [2], which specify whether a transaction is allowed to read data from other, concurrent transactions that are not yet committed, i.e. durable. When viewing single document read or write operations in MongoDB as transactions that contain only a single operation, the "local" and "majority" readConcern levels can be seen as analogous to the READ UNCOMMITTED and READ COMMITTED SQL isolation levels, respectively. An operation being "majority committed" in MongoDB replication can be viewed as similar to an operation being "committed" in the standard SQL isolation model. The durability guarantee of the "commit" event, however, is at the replica set level rather than the disk level. Reads at "majority" *readConcern* are only allowed to see majority committed data, and "local" *readConcern* reads are allowed to see data that has not yet majority committed i.e. they can see "uncommitted" data.

3.3 Usage in MongoDB Atlas

To characterize the consistency levels used by MongoDB application developers, we collected operational data from 14,820 instances running 4.0.6 that are managed by MongoDB Atlas, the fully automated cloud service for MongoDB. The data collected includes the values of *readConcern* and *writeConcern* that had been used for all read and write operations since the node had started up.[1] We found that

[1]These counts are fairly low, since all nodes had been recently restarted in order to upgrade them to 4.0.6.

the overwhelming majority of read operations used *readConcern* level *"local"* and the majority of write operations used *writeConcern w:1*. Our results are shown in Table 1 and Table 2. It appears that users generally accept the defaults.

Table 1: Read Concern Usage in MongoDB Atlas

Read Concern	Count	%
available	142	<.01
linearizable	28,082	<.01
local	103,030,820	.27
majority	50,990,496	.13
snapshot	2,029	<.01
none (default local)	38,109,403,854	99.60

Table 2: Write Concern Usage in MongoDB Atlas

Write Concern	Count	%
`{w:1}`	912,856,101	4.06
`{w:"majority"}`	3,565,186,929	15.86
none (default `{w:1}`)	17,793,179,026	79.17
other	203,962,459	.91

3.4 Use Cases

To understand patterns in how users choose consistency levels in production applications, we spoke with MongoDB users, as well as MongoDB consulting engineers and solutions architects, who help customers design solutions using MongoDB. Their impression was that the decision is often made more based on latency than on consistency requirements. They explained that users would prefer, of course, to use *readConcern* level *"majority"* and *writeConcern w:"majority"*, since everyone wants safety. However, when users find stronger consistency levels to be too slow, they will switch to using weaker consistency levels. These decisions are often based on business requirements and SLAs rather than granular developer needs. As we argue throughout this paper, the decision to use weaker consistency levels often works in practice because failovers are infrequent and data loss from failovers is usually small. When reading from and writing to the primary, users usually read their own writes and the system behaves like a single node.

Although consistency levels are often chosen based on latency rather than safety, we did gather some use cases for various consistency levels based on the application's consistency requirements, including cases that require the ability to tune consistency at the operation level. Note that this is a selection of use cases, and it does not cover all consistency offerings in MongoDB (for example, *writeConcern w:2* is also popular with customers).

3.4.1 Majority Reads and Writes

Customers use the combination *readConcern* level *"majority"* and *writeConcern w:"majority"* in applications for which safety is more important than latency. One example is a student loan financing site. This site receives about 2 writes per minute, so a bit of latency does not create a bottleneck. However, if writes are lost, a student may need to restart a long online form from scratch, so durability is essential in this case.

3.4.2 Local Reads and Writes

Customers use *readConcern* level *"local"* and *writeConcern w:1* for data that is not considered the "system of record". This could include ephemeral state, caches, aggregations, and logs. These consistency levels are also used when throughput is much more important than durability, so application infrastructure cannot tolerate high latency database operations. An example of ephemeral state is a game site that matches active players. This site has a high volume of writes, since its popularity means there are many active players looking to begin games. Durability is not important in their use case, since if a write is lost, the player typically retries immediately and is matched into another game.

3.4.3 Multiple Write Concern Values

A long online form, e.g. to create an account or order a pizza, might persist its state at several save points before the user completes it. Such a form is a common use case for multiple *writeConcern* values for the same data. Often a long online form will include multiple save points, where a partial write of the form is sent to the database. The writes at save points are performed using *w:1*, and the final form submission write is performed using *w:"majority"*. The exact timing of the save points is often invisible to end users. These write concern choices protect against write loss for the final form submission, while ensuring low latency for the save points. Here, only the final save is an explicit user action that users expect to be durable. We observed similar choices in the shopping cart model, whereby adding and removing items is done using *writeConcern w:1* but the ultimate order placement is done at *writeConcern w:"majority"*.

3.4.4 Local Reads and Majority Writes

A popular review site uses *readConcern* level *"local"* and *writeConcern w:"majority"* for its reviews. Write loss is painful, since users may spend significant time writing a review, and using *w:"majority"* guards against write loss. Reviews are read with *readConcern* level *"local"*, since users benefit from reading the freshest data, and there is no harm in displaying an unacknowledged write that might be rolled back.

3.4.5 Majority Reads with Causal Consistency and Local Writes

This combination is useful when writes are small, but double writes are painful. Consider a social media site with short posts. Low-latency posts are desirable, and write loss is acceptable, since a user can rewrite their post, so writes use *w:1*. However, double writes are painful, since it is undesirable user behavior to have the same post twice. For this reason, reads use *readConcern* level *"majority"* with causal consistency so that a user can definitively see whether their post was successful.

4. TUNABLE CONSISTENCY OFFERINGS IN OTHER COMMERCIAL DATABASES

Many originally single-node databases that have since expanded to support replication do not offer tunable consistency levels across nodes. For example Oracle's add-on replication system, GoldenGate, does not have tunable durability or staleness guarantees. A write will be acknowledged as

successful once it has been committed locally to an Oracle instance, and then will be asynchronously replicated sometime thereafter. Further, GoldenGate allows multiple nodes to accept writes for the same data set [16]. Reads, even if targeting a single node, may see data that will ultimately roll back from a yet-to-be-discovered write conflict. Oracle's Data Guard offers both asynchronous and synchronous one-way replication. Synchronous replication waits for the passive node to commit the write before returning success to the client. Synchronous replication in Data Guard cannot be specified at the operation level. Additionally, there is no way to specify the durability requirements of data read in a query. A query may see data yet to be committed to the passive node, data that may ultimately be lost.

MySQL's group replication supports tunable consistency across nodes at session granularity. It can be configured for single primary or multi-primary replication. By default, every transaction first determines ordering of the transaction across a majority of nodes then commits the transaction to a majority before acknowledging success [6]. MySQL group replication allows for only one notion of durability equivalent to *majority committed*. Every read only sees the equivalent of *majority committed* data, no in-progress or *locally committed* transactions are visible to outside operations. In systems of high write volume readers will perpetually see out-of-date data, since the *majority committed* data will be stale as recent writes will not yet be reflected. Multiple writers attempting to concurrently write to the same row may continue to see write conflicts. MySQL has recently introduced tunable consistency levels that mitigate the effects of seeing stale data [7]. The consistency levels offered are *EVENTUAL*, *BEFORE*, *AFTER*, and *BEFORE AND AFTER*. The *EVENTUAL* level is the default behavior described above where readers only read the equivalent of *majority committed* data. *BEFORE* specifies that the transaction will wait until all preceding transactions are complete before starting its execution. *AFTER* specifies that the system will wait for the transaction to be committed and visible on every node before being acknowledged, so that any reader will see the write regardless of which node it reads from. *BEFORE AND AFTER* combines these guarantees. While these knobs are useful to minimize the staleness of readers, each consistency level requires coordination across at least a majority of nodes, introducing a latency cost.

Postgres has both asynchronous (the default) and synchronous replication options, neither of which offers automatic failure detection and failover [12]. The synchronous replication only waits for durability on one additional node, regardless of how many nodes exist [13]. Additionally, Postgres allows one to tune these durability behaviors at the user level. When reading from a node, there is no way to specify the durability or recency of the data read. A query may return data that is subsequently lost. Additionally, Postgres does not guarantee clients can read their own writes across nodes.

Cassandra's multi-primary replication system offers tunable consistency at an operation level. Similar to MongoDB, Cassandra has two notions of durability equivalent to *locally committed* and *majority committed*. Cassandra has *QUORUM* writes that ensure the data is committed to a majority of nodes with the row before acknowledging success, giving the behavior of MongoDB's *writeConcern* level *majority* [9]. However, Cassandra's *QUORUM* reads do not guarantee that clients only see *majority committed* data, differing from MongoDB's *readConcern* level *"majority"*. Instead Cassandra's *QUORUM* reads reach out to a majority of nodes with the row and return the most recent update [4], regardless of whether that write is durable to the set. Because of MongoDB's deterministic write ordering, reads at *readConcern* level *"majority"* can be satisfied without cross-node coordination. In order to get causal consistency (or "read your writes") guarantees in Cassandra, each read and write operation must pay the latency cost of coordinating a majority of nodes.

MongoDB's consistency model was designed to offer users the power of granularly tuning consistency, staleness, and latency tradeoffs. Additionally, MongoDB has a reliable ordering of operations, enforced on the primaries of replica sets and maintained by the *data rollback* procedure discussed in later sections. This allows for *majority committed* reads and causally consistent reads to be satisfied without requiring coordination across multiple nodes.

5. SPECULATIVE MAJORITY AND SNAPSHOT ISOLATION FOR MULTI STATEMENT TRANSACTIONS

In this section, we discuss the behavior and implementation of consistency levels within multi-statement transactions. We chose an innovative strategy for implementing *readConcern* within transactions that greatly reduced aborts due to write conflicts in back-to-back transactions. When a user specifies *readConcern* level *"majority"* or *"snapshot"*, we guarantee that the returned data was committed to a majority of replica set members. Outside of transactions, this is accomplished by reading at a timestamp at or earlier than the majority commit point in WiredTiger. However, this is problematic for operations that can write, including transactions. It is useful for write operations to read the freshest version of a document, since the write will abort if there is a newer version of the document than the one it read. This motivated us to implement "speculative" majority and snapshot isolation for transactions: transactions read the latest data for both read and write operations, and at commit time, if the *writeConcern* is *w:"majority"*, they wait for all the data they read to become majority committed. This means that a transaction only satisfies its *readConcern* guarantees if the transaction commits with *writeConcern w:"majority"*. This is acceptable as long as the client application avoids creating side effects outside the database if transaction commit fails.

Waiting for the data read to become majority committed at commit time rarely adds latency to the transaction, since if the transaction did any writes, then to satisfy the *writeConcern* guarantees, we must wait for those writes to be majority committed, which will imply that the data read was also majority committed. Only read-only transactions require an explicit wait at commit time for the data read to become majority committed. Even for read-only transactions, this wait often completes immediately because by the time the transaction commits, the timestamp at which the transaction read is often already majority committed.

Our decision to implement speculative majority and snapshot isolation was motivated by the high incidence of transaction aborts due to write conflict in user beta-testing. Making this change drastically improved user experience. Fig-

ure 3 illustrates this point. In the diagram, time proceeds from top to bottom. In both examples shown, C1 and C2 both perform a successful update to the same document with *writeConcern w:"majority"*. On the left, C2 reads from the locally committed view of the data on the primary, P, and makes its update relative to that view. On the right, C2 reads from the majority committed view of the data, and makes its update relative to that view. For C2's write to be successful in the diagram on the right, it must not begin the read phase of its update until C1's write is majority committed, leading to increased delay between successful back-to-back writes to the same document. The red oval indicates the extra time required between successful back-to-back writes when reading from the majority committed view.

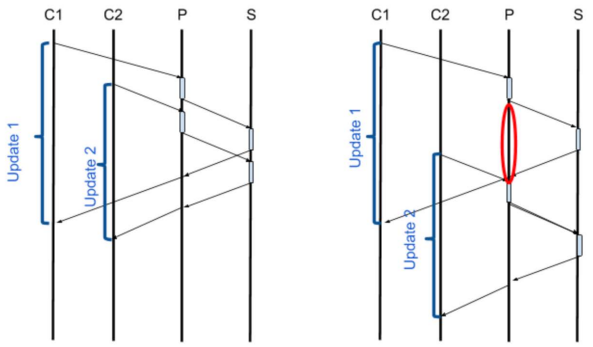

Figure 3: Back-to-Back Transactions with and without Speculative Snapshot Isolation

6. SUPPORTING A SPECTRUM OF CONSISTENCY LEVELS IN A SINGLE DEPLOYMENT

MongoDB's replication system architecture is specifically designed to support the spectrum of consistency levels mentioned above. The two novel aspects of the system that enable this are its *speculative execution* model and its *data rollback* algorithms. The former necessitates the latter, so we will first discuss speculative execution in MongoDB replication and subsequently discuss how it motivates data rollback.

6.1 Speculative Execution Model

The MongoDB replication protocol is similar to the Raft consensus algorithm, which is based on a replicated state machine model. A leader node accepts client requests, serializes them into an operation log, and replicates these log entries to follower nodes. All servers apply these log entries in a consistent order, producing the same final state. In the case of MongoDB, the database itself is the state machine, and entries in the oplog correspond to operations on this state machine. In Raft, log entries are not applied to the state machine until they are known to be *committed*, which means that they will never be erased from the log. In contrast, MongoDB replicas apply log entries to the database as soon as they are received. This means that a server may apply an operation in its log even if the operation is uncommitted. This allows MongoDB to provide the *"local"* read concern level. As soon as a write operation is applied on some server, a *"local"* read is able to see the effects of that write on that server, even before the write is majority committed in the replica set.

6.2 Data Rollback

MongoDB's speculative execution model makes it necessary for the replication system to have a procedure for *data rollback*. As discussed above, servers may apply log entries before they are majority committed. This implies that these log entries may need to be erased from the log at some later point. In a protocol like Raft, this rollback procedure consists of truncating the appropriate entries from a log. In MongoDB, in addition to log truncation, it must undo the *effects* of the operations it deletes from a log. This requires modifying the state of the database itself, and presents several engineering challenges.

MongoDB's rollback protocol engages two separate replica set nodes, a *rollback node* and a *sync source node*. The process is initiated by the rollback node when it detects that its log has diverged from the log of the sync source node, i.e. its log is no longer a prefix of that node's log. The precise conditions for rollback can be examined in the Raft paper and the MongoDB source code, so the full details will not be described here. Once a node determines that it needs to begin the rollback procedure, it will determine the newest log entry that it has in common with the sync source. The timestamp of this log entry is referred to as t_{common}. The node then needs to truncate all oplog entries with a timestamp after t_{common}, and modify its database state in such a way that it can become *consistent* again. A database state is *consistent* if it corresponds to a state that could have been produced by applying log entries up to some particular point. There are two different algorithms that a node can use in order to complete the rollback process.

6.2.1 Recover to Timestamp Algorithm

The first algorithm is known as "recover to timestamp" (RTT). Since MongoDB version 4.0, the WiredTiger storage engine has provided the ability to revert all replicated database data to some previous point in time. The MongoDB replication system periodically informs the storage engine of a *stable timestamp* (t_{stable}), which is the latest timestamp in the MongoDB oplog that is known to be majority committed and also represents a consistent database state. The algorithm works as follows. First, the rollback node asks the storage engine to revert the database state to the newest stable timestamp. After this procedure is complete the database state is guaranteed to reflect the timestamp, t_{stable}, that storage reverted the data to. Note that t_{stable} may be a timestamp earlier than the rollback common point. Then, the node applies oplog entries forward from t_{stable} up to and including the common point. After this procedure is finished, the database state is consistent and reflects the common point timestamp t_{common}.

6.2.2 Refetch Based Rollback

The second rollback algorithm, known as "refetch based rollback", was the MongoDB replication system's original rollback algorithm. It was replaced as the default by RTT in version 4.0. It still exists in the current system to support certain scenarios where RTT is not possible. The refetch based rollback algorithm can be viewed as a type of data

re-synchronization procedure. While the goal of RTT is to bring the database back to a consistent state at t_{common}, refetch based rollback will instead bring the database back to a consistent state at *some* timestamp newer than t_{common}. This algorithm was designed before the storage engine exposed the ability to revert data to a historical timestamp. The refetch rollback algorithm steps through each operation that needs to be rolled back and records information about how to undo each operation. The particular semantics of "undo" depend on the operation type, and undoing an operation may be non-trivial since the oplog is not an undo log. For example, an *insert* operation can be reverted locally by simply deleting the associated document. For other operations, like document deletions, the node does not have all the data necessary in order to revert the operation. In this case, it will refetch the necessary data from the sync source node, and update its local database with the data it fetched. After refetching all necessary data and undoing any possible operations, the rollback node transitions into a "recovering" state, where it will start syncing oplog entries again from its sync source. Client reads are prohibited during this state, since the data may not be consistent yet. During this phase the rollback node will apply oplog entries as if it were a secondary, until it is guaranteed that its database state is consistent again at a timestamp known as *minValid*. *minValid* is the timestamp of the newest oplog entry on the sync source node at a time no earlier than when the rollback node last fetched new data from it. After applying log entries up to *minValid*, the node will be consistent and can become a normal secondary again.

The refetch based rollback algorithm is complex and challenging to implement correctly. It involves network communication between two nodes, and it depends on knowledge about the semantics of all possible operation types that may be put into the oplog. The RTT algorithm eliminates much of this complexity by moving the rollback logic to a lower layer of the system. We feel that MongoDB's design and implementation of rollback in a consensus system like this is novel, and it enables the wide spectrum of consistency levels.

7. OPERATIONS MINIMIZE THE IMPACT OF THEIR CONSISTENCY LEVEL ON THE SYSTEM

Since multiple consistency levels are supported in the same deployment, it is important that choosing a stronger consistency level has little impact on the performance of the rest of the system. In this section, we describe how this is achieved in MongoDB for read and write operations.

7.1 Write Operations

The cost of consistency for a write operation is completely paid by the latency of that particular write operation. A write operation with *writeConcern w:1* can return as soon as the operation is durable on the primary. A write operation with *writeConcern w:"majority"* can return when the operation is durable on a majority of voting nodes in the replica set. The server performs the same work in both cases, and the only difference is how soon the operation returns success.

7.2 Read Operations

```
batch := ∅;
while !batch.full() & wiredTigerCursor.more() do
    if timeToYield() & getReadConcern()
     ≠ "snapshot" then
        releaseLocks();
        abortWiredTigerTransaction();
        beginWiredTigerTransaction();
        reacquireLocks();
    end
    batch.add(wiredTigerCursor.next());
end
return batch;
```

Algorithm 1: Query Execution in MongoDB

To understand the impact of *readConcern* on the rest of the system, it is necessary to discuss some details of reads in the WiredTiger storage engine. All reads in WiredTiger are done as transactions with snapshot isolation. While a transaction is open, all later updates must be kept in memory [2]. Once there are no active readers earlier than a point in time t, the state of the data files at time t can be persisted to disk, and individual updates earlier than t can be forgotten. Thus a long-running WiredTiger transaction will cause memory pressure, so MongoDB reads must avoid performing long-running WiredTiger transactions in order to limit their impact on the performance of the system.

7.2.1 Local Reads

Reads with *readConcern* level *"local"* read the latest data in WiredTiger. However, local reads in MongoDB can be arbitrarily long-running. In order to avoid keeping a single WiredTiger transaction open for too long, they perform "query yielding" (Algorithm 1): While a query is running, it will read in a WiredTiger transaction with snapshot isolation and hold database and collection locks, but at regular intervals, the read will "yield", meaning it aborts its WiredTiger transaction and releases its locks. After yielding, it opens a new WiredTiger transaction from a later point in time and reacquires locks (the read will fail if the collection or index it was reading from was dropped). This process ensures that local reads do not perform long-running WiredTiger transactions, which avoids memory pressure. The consequence is that local reads do not see a consistent cut of data, but this is acceptable for this isolation level.

7.2.2 Majority Reads

Reads with *readConcern* level *"majority"* also perform query yielding, but they read from the majority commit point of the replica set. Each time a majority read yields, if the majority commit point has advanced, then the read will be able to resume from a later point in time. Again, majority reads may not read a consistent cut of data. A majority read could return 5 documents, yield and open a WiredTiger transaction at a later point in time, then return 5 more documents. It is possible that a MongoDB transaction that touched all 10 documents would only be reflected in the last 5 documents returned, if it committed while the

[2]This is a simplification. WiredTiger, in fact, has a mechanism to overflow these updates to disk in a "lookaside buffer", so WiredTiger is not limited by the amount of RAM on the system. However, there is a performance impact of utilizing the lookaside buffer.

read was running. This inconsistent cut is acceptable for this isolation level. Since the read is performed at the majority commit point, we guarantee that all of the data returned is majority committed.

It is worth noting that supporting majority reads impacts the performance of the system, since all history since the majority commit point must be kept in memory. When the primary continues to accept writes but is unable to advance the majority commit point, this creates memory pressure. Considering this, one might conclude that it would be desirable to disable majority reads in order to improve performance. However, keeping history back to the majority commit point in memory is necessary for other parts of the system, such as the "recover-to-timestamp" algorithm (Section 6.2.1), so we have chosen to architect the system to ensure that the majority commit point can always advance, which allows us to support majority reads with no additional cost.

7.2.3 Snapshot Reads

Reads with *readConcern* level *"snapshot"* must read a consistent cut of data. This is achieved by performing the read in a single WiredTiger transaction, instead of doing query yielding. In order to avoid long-running WiredTiger transactions, MongoDB kills snapshot reads that have been running longer than 1 minute.

8. EXPERIMENTS

8.1 Latency Comparison of Different Write Concern Levels

In this section, we compare the write latency for different values of *writeConcern*. Most application developers would prefer to use stronger durability guarantees, but choose weaker guarantees when they are unable to pay the latency cost. This generally works in practice because failures are infrequent and write loss is small, so the developer tolerates small write loss in exchange for lower latency on all operations. These experiments demonstrate the higher latency of stronger durability guarantees, motivating why developers choose weaker guarantees. The experiments in this section were performed by Henrik Ingo, of the server performance team at MongoDB.

We performed three experiments on 3-node replica sets using different geographical distributions of replica set members. In each experiment, we tested different *writeConcern* values and different client locations, where applicable. Each experiment performed 100 single-document updates. All operations specified that journaling was required in order to satisfy the given *writeConcern*.

8.1.1 Local Latency Comparison

In this experiment, all replica set members and the client were in the same AWS Availability Zone (roughly the same datacenter) and Placement Group (roughly the same rack). All replica set members were running MongoDB 4.0.2 with SSL disabled. The cluster was deployed using sys-perf, the internal MongoDB performance testing framework. Results are shown in Table 3 and Figure 4.

8.1.2 Cross-AZ Latency Comparison

In this experiment, all replica set members were in the same AWS Region (the same geographic area), but they were in different Availability Zones. Client 1 was in the same Availability Zone as the primary, and Client 2 was in the same Availability Zone as a secondary. All replica set members were running MongoDB 4.0.3 with SSL enabled. The cluster was deployed using MongoDB Atlas on M60 instances with 5280 PIOPS.[3] Results are shown in Table 4 and Figure 5. We recognize that it is surprising that Client 2 had lower latency for *writeConcern w:"majority"*, and we believe this is just noise.

Table 3: Local Latency Comparison of Write Concern Values in Milliseconds

Write Concern	Avg	99%
{w:1}	1.25	2.05
{w:"majority"}	3.18	4.12

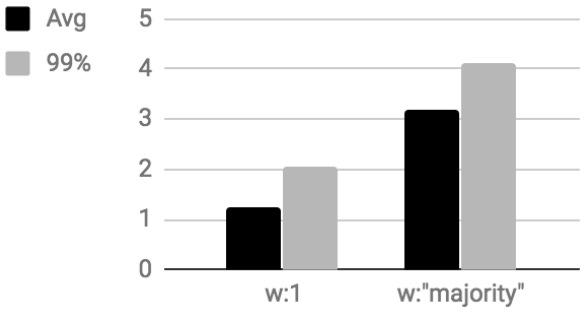

Figure 4: Local Latency Comparison of Write Concern Values in Milliseconds

8.1.3 Cross-Region Latency Comparison

In this experiment, all replica set members were in different AWS Regions. The primary was in US-EAST1, one secondary was in EU-WEST-1, and the other secondary was in US-WEST-2. Client 1 was in US-EAST1, and Client 2 was in EU-WEST-1. All replica set members were running MongoDB 4.0.3 with SSL enabled. The cluster was deployed using MongoDB Atlas on M60 instances with 5280 PIOPS. Results are shown in Table 5 and Figure 6.

8.2 Quantifying w:1 Write Loss In a Fail-Stop Model

As discussed previously, users may choose to trade off latency for consistency depending on their application requirements. We designed a simple workload simulation to estimate the degree of *w:1* write loss in a replica set under standard failure modes. We developed an insert-only workload that runs against a 3 node replica set deployed on AWS in the US-EAST-1 Availability Zone. The replica set hosts were c3.8xlarge instances with 320 GB EBS volumes, with 5500 IOPS, running Amazon Linux 2. All replica set nodes were running MongoDB v4.0.2-rc0. There is a dedicated workload client host that runs on the same instance type.

The workload client is written in Python and uses the PyMongo driver to communicate with the replica set. It has 30

[3]The differences among these test setups (e.g. MongoDB version, SSL, instance size) are acceptable because our focus is comparing performance of different write concern levels within one experiment, rather than performance comparisons across experiments.

Table 4: Cross-AZ Latency Comparison of Write Concern Values in Milliseconds

	Client 1		Client 2	
Write Concern	Avg	99%	Avg	99%
{w:1}	1.83	2.92	2.01	3.02
{w:"majority"}	4.85	5.95	4.32	5.25

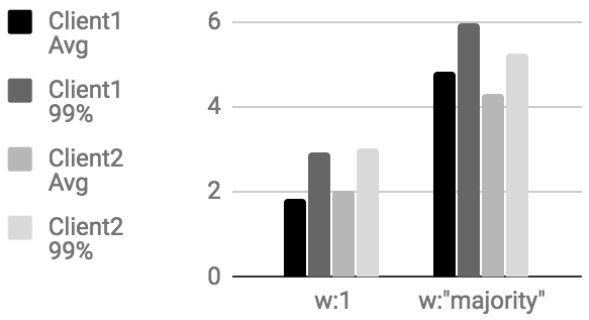

Figure 5: Cross-AZ Latency Comparison of Write Concern Values in Milliseconds

Table 5: Cross-Region Latency Comparison of Write Concern Values in Milliseconds

	Client 1		Client 2	
Write Concern	Avg	99%	Avg	99%
{w:1}	2.16	14.9	72	73
{w:"majority"}	185	192	217	480

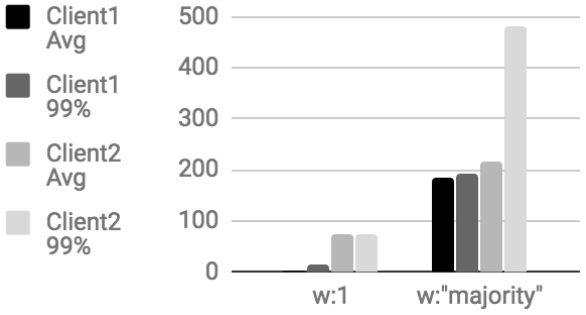

Figure 6: Cross-Region Latency Comparison of Write Concern Values in Milliseconds

concurrent writer threads that each insert documents into a test collection, which is initially empty. All writes are done at *w:1* write concern. The workload client inserts as many documents as possible until exceeding a specified workload time limit, which was set at 1 hour for our experiments. To simulate failures, the MongoDB process for each replica set node is forcefully killed and restarted at periodic intervals. The processes are shut down with a SIGKILL signal. There are two time intervals of importance to the failure simulation: (1) the time between a restart and subsequent shut down and (2) the time between shut down and restart, which can be viewed as the "time to repair". The results from [5] provide an empirical analysis of failure and repair behavior in cloud environments, specifically in Google Cloud. They find that a Weibull distribution is a good model for time between server failures, and we used this model for our workload simulation. We scale down the time intervals so that we can run a single workload in a reasonable amount of time, but we model time between failure using this distribution. Thus, we use a Weibull distribution with a shape parameter $k = 1.5$ and a scale parameter $\lambda = 60$. The shape parameter was based on Table 2, Server Architecture 7 from [5]. The scale parameter has units of seconds for our experiment. This gives a mean time between shutdowns of 54.16 seconds. The referenced paper also examines the "time to repair" distribution but for simplicity we do not include those details in our test. We restart nodes 10 seconds after killing them. We also kill any nodes in the replica set with equal likelihood. Killing only the primary node would potentially lead to more lost writes, but uniformly killing nodes models real world failure more accurately, since failures should be just as likely to occur on a primary or secondary. The results of our experiments are shown in Table 6. We carried them out at several different network latency levels. We induced artificial network latency by using the Linux traffic control ("tc") tool to delay IP traffic between the replica set hosts. No latency was added between the workload client and the replica set hosts. At the end of the workload, we compare the set of inserts that were acknowledged to the client versus the set of inserts that actually appear in the database i.e. the durable operations. We look at the proportion of acknowledged to durable to quantify how many writes were lost. This quantity *durable/acknowledged* is shown as "Durable %" in Table 6.

At higher latency levels we see higher write loss levels. This is likely due to the fact that, with higher latency, it is more probable that many writes have been written on the primary but not yet replicated to secondaries at the time a crash occurs.

Table 6: w:1 Write Loss Under Failure, 1 hour workload duration

Latency(ms)	Ack'd	Lost	Durable %
0	4,039,799	1,297	99.968
100	3,770,680	4,395	99.883
200	3,687,984	22,174	99.399

8.3 Failover Data from MongoDB Managed Replica Set Deployments

To further quantify the frequency of replica set failure in the real world, we collected operational data from 17,720 3-node replica set deployments managed by MongoDB over the course of 1 week. We were not able to directly measure the number of node crashes or restarts with the data collected, but we analyzed the number of unique *terms* seen by each replica set over this time period. Recall that terms are monotonically increasing integral values maintained on each replica set node, and are used as a way to totally order elected replica set leaders. There can be at most one leader for a given term. Thus, by examining the set of all unique terms over some period of time, it is possible to get a sense of how many elections that replica set experienced. This additionally acts as a rough proxy for measuring failure frequency, since the failure of nodes may cause an election

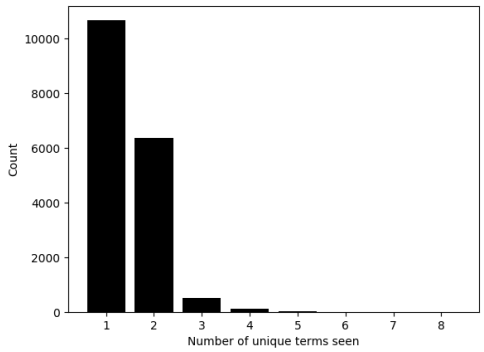

Figure 7: Number of terms seen by 3 node replica sets over 1 week in MongoDB Managed Deployments

to occur. It is possible for a term increase to occur across a replica set without the successful election of a new primary, but that is expected to be relatively rare. The results are depicted in Figure 7. 10,668 replica sets experienced 1 term (60.2 % of total), 6376 experienced 2 terms (35.98 % of total), and 515 experienced 3 terms (2.9 % of total). Less than 1 percent experienced 4 terms or more. In many of these managed deployments, it is likely that one of the most common source of term changes is an election due to a planned maintenance event, e.g. upgrading a server's binary version. In these cases, an election is necessary since the node undergoing maintenance (e.g. the primary) must eventually be shut down, but a planned stepdown will wait for a majority of replica set nodes to be caught up to the primary before the stepdown succeeds. Thus, all writes that were present on the primary at the start of the stepdown should become majority committed, reducing the loss of any *w:1* writes.

9. CONCLUSIONS AND FUTURE WORK

MongoDB allows users to configure each operation's consistency level, allowing them to choose the safety and performance requirements of each operation. As a highly available replicated database, it is important to give operators the choice of whether to view the system as a single-node system-of-record or a low-latency distributed system that permits short periods of inconsistency, and MongoDB allows this flexibility. In MongoDB 3.6 and 4.0, we extended our consistency offerings with causal consistency and transactions with snapshot isolation, and we intend to continue improving our consistency features. In upcoming releases, we plan to reduce the latency of majority writes, which will allow developers to more readily choose this durability guarantee. We also plan to build support for long-running reads at snapshot isolation that have no impact on the rest of the system, where any performance cost is entirely paid for by the operation. As future work, we would like to translate our matrix of *readConcern* and *writeConcern* offerings into coherent consistency settings, in order to improve the comprehensibility of these features.

10. ACKNOWLEDGMENTS

We would like to thank Henrik Ingo of the server performance team at MongoDB for allowing us to use his results on comparing latencies of different *writeConcern* values. We would like to thank the MongoDB Consulting Engineers and Solutions Architects who described real-world use cases and customer attitudes on consistency: André Spiegel, Jay Runkel, John Page, Matt Kalan, and Paul Done. We would like to thank Esha Maharishi for her example illustrating the performance improvement of speculative majority and snapshot isolation. We would like to thank our internal reviewers: A. Jesse Jiryu Davis, Henrik Ingo, Judah Schvimer, Andy Schwerin, and Siyuan Zhou. Finally, we would like to thank the entire MongoDB Replication team and its alumni, all of whom contributed to the design and implementation of features described in this paper.

11. REFERENCES

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, Feb 2012.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[3] BSON Specification. `http://bsonspec.org/`. Accessed: 2019-02-04.

[4] Examples of read consistency levels. `https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlClientRequestsReadExp.html`. Accessed: 2019-02-12.

[5] P. Garraghan, P. Townend, and J. Xu. An Empirical Failure-Analysis of a Large-Scale Cloud Computing Environment. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 113–120, Jan 2014.

[6] Group Replication Background. `https://dev.mysql.com/doc/refman/8.0/en/group-replication-background.html`. Accessed: 2019-02-12.

[7] Group Replication - Consistent Reads. `https://mysqlhighavailability.com/group-replication-consistent-reads/`. Accessed: 2019-02-12.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[9] How is the consistency level configured? `https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlConfigConsistency.html`. Accessed: 2019-02-12.

[10] D. Jordan. Sdam monitoring specification. `https://github.com/mongodb/specifications/blob/master/source/server-discovery-and-monitoring/server-discovery-and-monitoring-monitoring.rst`, 2016–2018.

[11] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages

305–320, Berkeley, CA, USA, 2014. USENIX Association.

[12] Postgres Failover. https://www.postgresql.org/docs/9.1/warm-standby-failover.html. Accessed: 2019-02-12.

[13] Postgres Synchronous Replication. https://www.postgresql.org/docs/9.1/warm-standby.html#SYNCHRONOUS-REPLICATION. Accessed: 2019-02-12.

[14] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[15] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, and J. Mulrow. Implementation of cluster-wide logical clock and causal consistency in mongodb. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, 2019.

[16] Using Oracle GoldenGate for Oracle Database: Automatic Conflict Detection and Resolution. https://docs.oracle.com/en/middleware/goldengate/core/18.1/oracle-db/automatic-conflict-detection-and-resolution2.html#GUID-EB3D5499-7F28-45B6-A64E-53BF786E32A5. Accessed: 2019-02-12.