Using MongoDB in Cloud Based Commercial Systems

Ethan M. Uphoff
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
uphof012@morris.umn.edu

ABSTRACT

MongoDB has seen more use in commercial systems in recent years. Commercial systems have many requirements for their users when it comes to accessing their data. Every millisecond counts when it comes to accessing said data as many users will tend to not use a service if it takes too long to access. As the private sector moves to the cloud, these delays are becoming a greater problem, potentially leading to an overall loss of sales. This isn't the only thing that needs to be accounted for, as the data being accessed also needs to be consistent with previous or concurrent operations. This is where causal consistency is required. This paper looks at how MongoDB's most popular drivers affect latency when deployed in the cloud as well as MongoDB's implementation of causal consistency and throughput loss associated with it.

Keywords

MongoDB, Causal Consitency, Latency, Drivers

1. INTRODUCTION

MongoDB is a NoSQL database which was released on February 11th 2009; it is classified as a document store database [9]. Document store databases have numerous advantages over SQL databases, many of which apply to commercial settings due to how the data is stored and maintained. MongoDB specifically lists 3 advantages over more traditional databases: higher productivity due to faster development, less structured data types, and more scalability over older solutions [3]. This makes MongoDB very appealing to companies as it can be used for a variety of projects allowing teams to quickly manage their own database without having to hire employees to maintain SQL tables.

An example of a use case for fast development is proof of concepts. A company wants to limit their investment in developing a proof of concept, as this takes away time from other projects which the company knows will work. MongoDB doesn't have a strict document structure. This means that employees can quickly change data as needed, as is commonly required by proof of concepts. For example, say the employees are working on a page which has information about a product but they don't need the price right away. MongoDB allows them to ignore the price field during development and later on can have it added without any additional hassle.

As companies begin to use MongoDB, some questions start to arise. One of the biggest ones is based on latency. In 2016, Google collected anonymized Google Analytics data and found that 53% of people leave a page if it takes three or more seconds to load [2]. That is 53% of potential customers being lost in the case of a commercial service. While there are multiple factors which need to be accounted for in the case of page load time, one thing that needs to be looked at is database access time. This means that while MongoDB has many features useful for development and use in general, it needs to be fast. One important part in determining this latency is the driver which MongoDB uses to choose where to obtain a users data.

Another interesting thing about MongoDB is its inclusion of a feature called causal consistency. This is incredibly important for systems that require that the most recent write a user makes is readable for said user as soon as possible. A good example of this is ATMS where users may want to deposit and immediately check their balance after. Causal consistency ensures that they can read their data as soon as possible after it is written. While this feature is more common in existing SQL systems, it is less seen in NoSQL databases [3]. This makes MongoDB a rather interesting case as it is extremely different from existing SQL systems but has many of the features to possibly replace existing SQL systems.

In the next section I'll be providing the necessary background material required to understand the studies in Section 3. In Section 3 I will be covering a study conducted by Bogdanov et al. [1] on the impact of various MongoDB drivers on latency and a study conducted by Tyulenev et al. [6] on how causal consistency affects throughput in MongoDB. I will wrap up this paper with a conclusion in Section 4.

2. BACKGROUND

In order to understand the studies in Section 3 exploring MongoDB latency and throughput, a few things need to be covered first about MongoDB and MongoDB concepts.

2.1 MongoDB and Replica Sets

MongoDB is a NoSQL document store database. This means that unlike traditional SQL databases, it stores its information in a JSON like structure rather than tables. This structure means that it lacks the constraints of strict

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/.
UMM CSci Senior Seminar Conference, December 2019 Morris, MN.

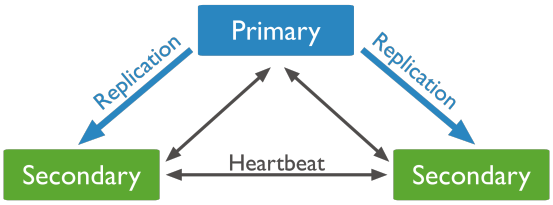

Figure 1: Model of a replica set. The primary node receives all writes and is the source of truth. The secondary nodes are a replica of the primary node. The heartbeat is the communication between nodes and specifies how old the data in a given secondary node is in order to determine if replication should occur [4].

rows and columns in SQL databases. It replaces tables with collections which can be thought of as an array of JSON files, where each set of data is a document within the collection. There are no requirements on what is in a document. This makes MongoDB compelling to work with for developers as it can be easier to work with than traditional SQL databases while keeping much of the more advanced functionality of SQL databases intact.

One important thing about MongoDB in commercial settings is how it is managed in the cloud. MongoDB allows users to create replica sets which are essentially copies of a MongoDB instance. In this setup, there is a primary node, one or more secondary nodes, and an optional arbiter. The primary node is the source of truth for the secondary nodes. All write requests go to the primary node but reads can go to any of the secondary nodes. Secondary nodes are going to hold a replica of the data in the primary node. Arbiters exist to essentially change secondary nodes into primary nodes if needed. A default MongoDB setup will replicate the data from the primary node to the secondary nodes as soon as it possibly can [4]. A simple setup with a primary node and two secondary nodes can be seen in Figure 1.

Using replica sets, MongoDB can decide on which replica to read data from, which has a significant impact on latency. A secondary node doesn't have to be on the same machine as the primary node. This means that a primary node could be based here in Minnesota while a secondary node could be in Hong Kong. This means that Chinese and US customers would see similar times in reading their data from MongoDB. Writes are likely to be slower from China, though, as they need to travel to Minnesota [1]. This means that if MongoDB were to pick a replica which is further away from the user than it should be, it can cause the latency for a given user to increase; thus causing a worse experience for the user.

2.2 Writes

Writes in MongoDB can be broken into two different types, durable and non-durable writes. Durable writing is part of the *write concern* functionality of MongoDB. Write concern allows the user to specify whether their data should be immediately propagated over a number of nodes [5]. A durable write would not be complete until it has spread to every node specified. This means that a durable writes can take quite a long time depending on the number of nodes being written to. Non-durable writes, on the other hand, only write to the primary node; once that write has occurred it is done. Non-durable writes essentially leave data replication up to the replica set algorithm in place and have nothing to do with the data replication process. Overall this leaves it up to the company storing the data to decide if they want the data to be quickly accessible on all replicas or let it eventually reach all replicas over time.

[1]There is something called multi-master to deal with this issue, but it is essentially just a greater scale version of normal replication with multiple primary nodes.

2.3 Consistency

When it comes to consistency, there are multiple models which are used. Two of these models are eventual consistency and causal consistency. Eventual consistency is the default model used by MongoDB, and is the model maintained by the normal replica set algorithm. If a write occurs on a node, it will eventually reach all nodes after some time [8]. Eventual consistency has its issues though. If a user makes a write and immediately follows up with a read, they wont necessarily get the information they just wrote to the database if the replica they're reading from hasn't been updated with the new information. MongoDB does work to try and mitigate some of these issues, but in general eventual consistency has many issues in regards to multi-node systems.

Eventual consistency finds its main use case in systems which customers may not have to interact with. For instance, a database which stores all the products for a company. If a company is releasing a product and it isn't important, they may be able to let eventual consistency propagate the data over all the secondary nodes over time. In cases where a product may be important, they are able to get around the issues of eventual consistency by using write concern to specify that they want it available in a certain number of nodes. Issues with eventual consistency start to appear in multi-node systems when customers begin to get directly involved. In cases where a user may want to read data they just wrote to the database, eventual consistency may not have propagated the data to the node the user is reading from at that time. This is where causal consistency steps in.

MongoDB is able to switch from eventual consistency to causal consistency if the developer decides to. Causal consistency is essentially a series of causal relationships which makes sure users can read their writes as soon as they possibly can from a secondary node [7]. This means that the likelihood of incorrect data being returned, or no data at all, is far less likely. In a multi-node system this means the users will not have to worry about being able to read writes recently made. If a user writes to the primary nodes, then reads from a secondary, the secondary node they are reading from will have the data they just wrote as causal consistency guarantees this.

For example, if someone is using an ATM which is based on MongoDB and wanted to make a deposit followed by a check balance request, causal consistency would step in. It would first deposit the money, effectively writing that deposit to the primary node. Once it is done with the write, it needs to make the information accessable to the user for reading as soon as possible. Causal consistency replicates the data to the node the user is reading their balance from, thus resulting in the correct balance. While in an eventually

consistent system with the same request, the user may not immediately receive their proper balance as the data needs to propagate eventually to the node they are reading from.

3. MONGODB PERFORMANCE STUDIES

There are a multitude of pieces in MongoDB which can affect overall latency and throughput. Two of these things are the drivers being used and the consistency model. Bogdanov et al. [1] chose to look at the two most popular MongoDB drivers and the overall effects drivers had on latency. Tyulenev et al. [6] decided to look into the effect causal consistency has on throughput compared to eventual consistency with MongoDB. Both latency and throughput are important for keeping users, as a system with high latency will bottleneck the overall throughput of the system causing nodes to be utilized poorly and low throughput will increase latency as users cannot get their data quickly.

3.1 MongoDB Replica Selection

When using MongoDB in a commercial environment, companies may want to sell their products around the world. An issue arises from this: optimizing the latency for users when accessing their websites. Bogdanov et al. [1] set out to find the optimal algorithm for selecting replicas, as well as determining the optimal distance users can be from replicas as to reduce latency [1].

A good example of this is a company such as Amazon who may want to sell products in a country such as Germany. While Amazon is based in the United States, they want to make sure their users in Germany have similar latencies when connecting to their website as users in the United States. Companies such as Amazon have noticed loss of revenue due to latency, as mentioned by Bogdanov et al.

> For example, Amazon reports that it loses 1% of sales if response latency increases by 100 ms [1].

Bogdanov et al. primarily used Amazon EC2, a cloud deployment service, for their research. The main focus of their research was to find bugs in commonly used replica selection algorithms [1]. In the process of doing so, they also ended up gathering data on optimal replica selection algorithms as well as overall latencies.

When analysing replica selection algorithms in MongoDB, Bogdanov et al. had to look at MongoDB's drivers [1]. These drivers manage which node is selected when a user is trying to read data. It will attempt to choose the most optimal node for the user. This makes drivers an important part of MongoDB when evaluating latencies.

3.1.1 Tools

To evaluate the algorithms being used, Bogdanov et al. [1] built a tool which they call GeoPerf. GeoPerf tests how the various replica selection algorithms respond to changes in latency on various nodes. This helped them to find issues in the algorithms as well as determine their effect on latency [1].

As for the latency information GeoPerf used, Bogdanov et al. collected data on the latency of all Amazon EC2 datacenters at the time. They had not found any previous latency information for this service so they had to collect it all themselves. This is important information as they found that data centers are not stable in terms of latency [1]. A request to a datacenter may have a latency of five milliseconds at some point and twenty milliseconds at another point. Bogdanov et al. used this important information in using GeoPerf to test the algorithms, as the algorithms would need to cope with latency changes.

Bogdanov et al. looked at two drivers in particular for MongoDB, C++ and Java, as they were the most widely used at the time of the study. There are other drivers which MongoDB can make use of, but they decided to only look at these since they were the most popular at the time [1].

3.1.2 Replica Selection

According to Bogdanov et al., replica selection has two important pieces to keep track of: latency smoothing and the replica selection algorithm. MongoDB manages latency smoothing based on the location of the client, which they call latency estimation. This latency estimation is calculated by the driver said MongoDB instance is using. The drivers goal is to look over available MongoDB nodes and determine which node is the fastest to access, or has the lowest latency. Each driver has different calculations they run in order to find the ideal node which causes variability when it comes to latency evaluation. This means while the C++ driver may find one node to be the best based on it's estimation, the Java driver may find a different node as each calculate latency evaluation differently [1].

The second factor in replica selection is the replica algorithm itself. MongoDB manages this mostly through the latency estimation process but also has an additional three steps [1].

1. Collects latency samples to find the closest replicas. These samples are the latency of calling MongoDB replicas, which are used to find the ideal replica for a user to read from.

2. Uses latency estimation to find relevant replicas with less than 15ms of latency, from the latency samples, if possible.

3. Selects a random replica from the remaining replicas.

These three steps are common across all the drivers, as most of the variance in replica selection is based in the latency estimation piece of MongoDB. The only part of this process which does not happen during latency evaluation is the final selection of a random node [1].

3.1.3 Evaluation of Algorithms and Drivers

For evaluating the algorithms being used, Bogdanov et al. used the GeoPerf tool they created, as well as a series of other tools to create tests for the algorithms. In order to test their the drivers, they used GeoPerf to emulate the latencies of the various AWS EC2 instances around the world. By doing this they could attach these fake latencies to nodes in their system, from there they were able to see which node the drivers would pick and analyse the results from this. It is important to note they were emulating datacenters as well so latencies were not always stable, the ideal datacenter may change randomly. An example of this process can be seen in Figure 2 [1]. They compared the results returned from the tests run by GeoPerf to determine which algorithm was the most optimal. To do this they calculated the total time each request would take based on the following equation:

$$T_{\text{total}} = \frac{RTT_{\text{request}}}{2} + T_{\text{processing}} + \frac{RTT_{\text{reply}}}{2}$$

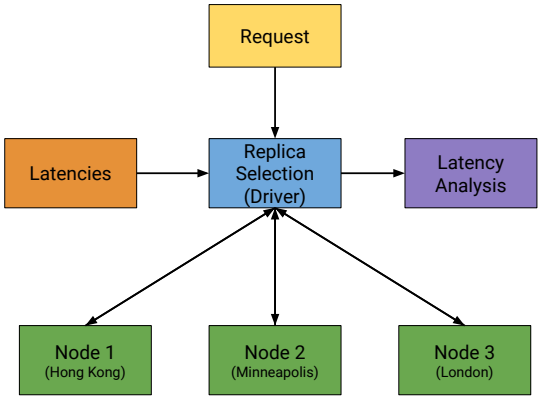

Figure 2: This is an abstracted model in which Bogdanov et al. used to test replica selection algotihms. Bogdanov et al. sent in requests and attached a fake latency to each node. They would then analyse which node was accessed and use the latency attached to said node to check the effect each driver had on latency. In this example, each node is being treated as a different city with it's own latency. The latency can then be analysed based on which node is selected. Based on [1].

In this equation, T is the total time spent doing a task. RTT is the simulated time gathered from looking into the latency of each AWS EC2 instance, ie, RTT is the simulated time from when info is sent to a replica and time to return. By using this equation, Bogdanov et al. could properly estimate the latency of each algorithm.

After running 230 iterations of GeoPerf over a MongoDB instance using Java drivers and one using C++ drivers, Bogdanov et al. found that the C++ drivers were significantly better than the Java drivers [1]:

> when Java and C++ drivers are compared under identical conditions, the Java driver demonstrates inferior performance in 80% of the cases.

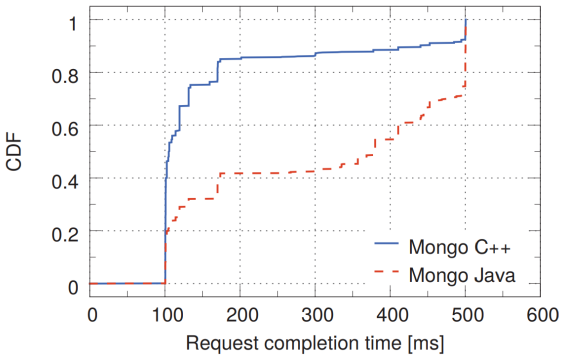

Figure 3: This shows the difference in latencies between MongoDB drivers. It is showing the percent of requests which completed in a given time. CDF is the percent of completed requests and Request Completion Time is the time in milliseconds it took for that percent of requests to complete [1].

They found that Java was not accounting for changes in the simulated latencies gathered from AWS EC2 many of the times. It was essentially taking a snapshot of latencies when the driver began running and failed to update them as it ran. C++ on the other hand, actively updated latencies resulting in better latencies overall [1]. This can be seen in Figure 3 as the C++ driver has a lower overall completion time than the Java driver.

This means overall, if a user were choosing between the two most popular MongoDB drivers to use in the cloud, they should use C++ whenever possible. C++ will update for changes in the cloud environment much more often than Java will. This results in lower overall latency for users.

It is important to note that the research done by Bogdanov et al. is now relatively outdated. Since the paper's writing in 2015, MongoDB has expanded it's drivers substantially and there is now a wider variety of driver choices, as well as drivers for specific purposes. This study would need to be run on the variations of the new C++ and Java drivers in order to be accurate again.

3.2 Causal Consistency

When developing cloud systems which make use of databases, an important requirement is that users be able to read their data after writing it. MongoDB has causal consistency built into it in order to circumvent these issues.

The MongoDB team did not want to eliminate eventual consistency when adding causal consistency to the database, and made a point to make both compatible in the same system [6]. They implemented this so only relevant data implements causal consistency, while less important data can propagate to nodes over time with eventual consistency. This means there does not need to be unnecessary strain on the database as it will not have to apply causal consistency to every request.

When comparing consistency models, throughput can be analysed in place of latency. Throughput is the number of operations a computer can process in a given amount of time. Throughput plays a role in the overall latency of a request as a device with lower throughput may process requests slower if multiple users are accessing it, thus causing increased latency. This means a system with greater throughput can process a greater number of users requests in a given amount of time.

This is particularly important in multi-node systems where a user's read requests may be picked up by different nodes. Causally consistent systems will make sure all the data the user is reading is up to date. An example of this, as mentioned in 2.3, is an ATM. Users need to be able to read their balance after depositing or withdrawing.

3.2.1 Logical Clocks

Tyulenev et al. [6] wanted a way to test the impacts of causal consistency in MongoDB's throughput to see the effects the user would see. They decided on testing this through a logical clock based on a paper written by Leslie Lamport in 1978 [6].

Tyulenev et al. decided to use a modified version of Leslie Lamport's scalar logical clock to track causal consistency. The logical clock they decided to use is called a hybrid logical clock. This clock takes the basic idea of a scalar logical clock and adds a real time value as well. The scalar portion is a

very simple incrementing clock. This clock increments when specific operations occur, attaching a logical time to specific requests. The part that makes this a hybrid clock is the real time aspect.

The incrementing, scalar, part of the clock and real time part are two separate entities in this system. The scalar clock exists to ensure that operations are occurring in a causally consistent manner. The real time is a separate value which is used to keep many of MongoDBs features intact. They wanted to make sure the rollback functionality of MongoDB could be used with this in case there were any issues with the data they needed to deal with. By keeping real time available, they could roll back to a specified time. If they did not care about additional MongoDB features, they could have gone with a normal scalar logical clock.

For a user using an ATM, this would be the system keeping track of their transactions. For instance, they make a deposit earlier in the week and said deposit is given a logical time of 31 as the deposit before it was given a logical time of 30 . If the user makes another deposit, that time will then be updated to 32 and they should expect that the data they then read has a logical time of 32 or 33 depending on how the time is updated. Each of these requests would also be given a real timestamp as well as it is a hybrid logical clock.

3.2.2 Dependencies

Tyulenev et al. decided to use explicit dependency tracking to ensure causal consistency was in effect. This dependency tracking acts as a layer which updates the scalar clock every time a specific type of request occurs. They decided to only update this time when a write occurs, this is due to the fact that a read will not change the data, thus not causing causal consistency to take effect. By doing this they were able to attach an additional write to the write the user sent, effectively adding a logical time and a real time to their write. They do note that while this does have very little overhead, it does have some and was the best option they had available. This method ends up putting more strain on the user rather than the server processing requests though, making it's overall effect on throughput much lower than a service running directly within the MongoDB instance [6].

Explicit dependency tracking would attach an `operationTime`, also referred to as a logical time, to requests as they came in. This method, while reducing throughput due to adding an additional write to each write request, would make sure every request happens in the correct order and maintains a correct logical time. `operationTime` is a variable stored in each item which records the logical ordering in which said action occurred. By comparing the `operationTime` in various requests, they could validate that causal consistency was in effect [6].

In terms of a user using an ATM, this is how the logical time is going to be written in the database. So in the case of the explicit dependency tracking Tyulenev et al. use, the logical time is stored as `operationTime`, and is updated only on writes. This means if a user deposits and the `operationTime` is 31 after the deposit, when they read their balance they will see an `operationTime` of 31 if it is shown.

3.2.3 Synchronizing Clocks

Once a system to track dependencies and a clock were decided on, there needed to be a way to keep the clock consistent with incoming writes. Tyulenev et al. decided to use SCT, Stable Cluster Time, to manage synchronization, since SCT is optimal for strictly increasing time [6].

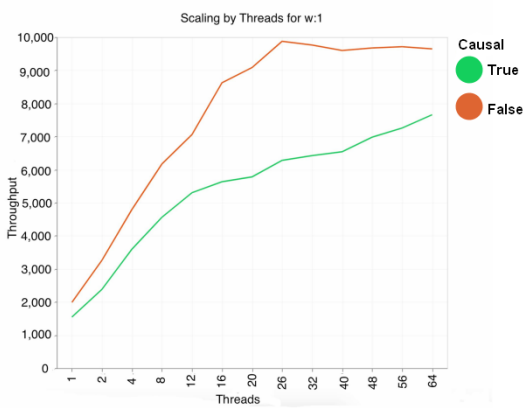

Figure 4: Non-Durable writes with and without causal consistency. w:1 is a level of write concern which can roll back writes if there is a failover. Throughput is measured using TCP-C which has clients sending random read and write operations which then measures the operations per second. [6]

SCT uses the MongoDB oplog, operation log, to track whenever updates happen. The oplog is a log in the primary node which tracks all writes which have occurred. SCT can make use of this by reading the oplog, seeing whenever it updates. Whenever the oplog updates, it indicates that a write has occurred which SCT uses to increment the `operationTime` value, thus making sure the `operationTime` is the correct value. When SCT updates `operationTime`, it uses a noop write. This write makes sure that the write which updates the `operationTime` value does not update the oplog. If the updating of the `operationTime` value was a normal write, it would result in an endless loop as SCT would see updates in oplog and continuously update `operationTime` when it should not be [6].

For the ATM example, this is how the `operationTime` updates. When the user makes their deposit on the ATM, it sends a write request to the database. This write is then recorded in the oplog which SCT then uses to increment the `operationTime` value. This ensures that the logical time is always valid when incrementing for writes.

3.2.4 Results

To calculate the effects causal consistency had on throughput there were a few requirements for the system Tyulenev et al. decided to use. They used a system with a multitude of threads in order to see how the number of threads affected throughput [6]. Threads in this case are logical processors. This means there can be more concurrent requests with more threads, thus increasing the overall throughput of a server. More threads also means more availability for the server to move data between nodes which is important for causal consistency.

Their testing was done on a single machine running MongoDB with multiple nodes. They compared said system running a causally consistent model as well as one running an eventually consistent model over a number of threads. They broke their testing into three distinct parts: non-durable

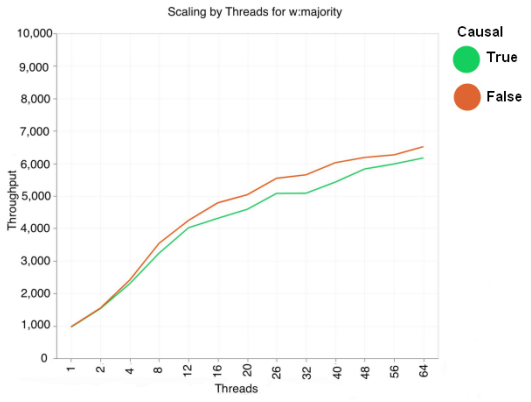

Figure 5: Durable writes with and without causal consistency. w:majority is a write concern which states that a majority of the data will be durable. Throughput is managed in the same way as in Figure 4.

writes, durable writes, and a read only workload. The non-durable writes and durable writes were aimed at how causal consistent systems affect overall throughput. The reads only workload was run to see how the number of nodes affects reads per second over a number of threads [6].

Non-Durable Writes

When using non-durable writes, Tyulenev et al. found that there was a significant loss of throughput with causal consistency enabled for numbers of threads between 12 and 40 but the gap grew smaller with larger numbers of threads. This can be seen in Figure 4. This means that if the user can afford a large number of threads, eventually the cost of causal consistency on non-durable writes will be almost the same as systems using eventual consistency [6].

Durable Writes

Using durable writes led to a very different result from non-durable writes. Due to the upfront cost of durable writes, the throughput was similar between systems using causal consistency and systems without it. This can be seen in Figure 5. In these cases, causal consistency would be useful in most systems as there isn't much of a penalty to using it [6].

Reading over Multiple Nodes

Finally, the tests on reads across multiple nodes show that with more nodes and threads, the number of reads per second increases substantially if a system has 16 or more threads. This means using a multi node system with durable writes would result in a very similar system to one without causal consistency maintaining durable writes [6]. This can be seen in Figure 6.

4. CONCLUSIONS

When developing a system in a commercial environment using MongoDB, there are a variety of factors to keep track of. Two of these factors are the overall latency and ordering of requests. Latency is made up of multiple pieces, some of which are easier to control than others. One which can

Figure 6: This shows the possible number of reads per second over a number of threads. The more nodes being used, the greater the number of reads per second which can occur.

be controlled is the driver which MongoDB uses. The most optimal driver between the two most used drivers in terms of latency is the C++ driver as the Java driver tends not to manage well under change according to tests run using GeoPerf [1]. Though it is very likely this research is now outdated as MongoDB's drivers have changed substantially since the publishing of the research done by Bogdanov et al. in 2015.

Another important feature is causal consistency. MongoDB has an implementation of causal consistency built into it which makes sure that users can immediately read what they write. Tyulenev et al. set out to see the effects causal consistency had on overall throughput. The results showed that causal consistency had very little effect on throughput for durable writes while having a larger effect on non-durable writes [6].

Acknowledgments

Thank you to Nic McPhee, KK Lamberty, and Kevin Arhelger for giving suggestions on how to improve this paper.

5. REFERENCES

[1] K. Bogdanov, M. Peón-Quirós, G. Q. Maguire, Jr., and D. Kostić. The nearest replica can be farther than you think. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 16–29, New York, NY, USA, 2015. ACM.

[2] GoogleData. Mobile site load time statistics. [Online; accessed 03-November-2019; https://www.thinkwithgoogle.com/data/mobile-site-load-time-statistics/].

[3] MongoDB. Mongodb and mysql compared. [Online; accessed 03-November-2019; https://www.mongodb.com/compare/mongodb-mysql].

[4] MongoDB. Replication. [Online; accessed 03-November-2019; https://docs.mongodb.com/manual/replication/].

[5] MongoDB. Write concern. [Online; accessed 03-November-2019; https://docs.mongodb.com/manual/reference/write-concern/].

[6] 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, pages 636–650, New York, NY, USA, 2019. ACM.

[7] Wikipedia. Causal consistency — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 03-November-2019; https://en.wikipedia.org/wiki/Causal_consistency].

[8] Wikipedia. Eventual consistency — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 03-November-2019; https://en.wikipedia.org/wiki/Eventual_consistency].

[9] Wikipedia. Mongodb — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 03-November-2019; https://en.wikipedia.org/wiki/MongoDB].