MongoDB - ESLE Report 02 - G01

André Augusto 90704, Lucas Vicente 90744, and Manuel Mascarenhas 90751

Instituto Superior Técnico, Edifício Núcleo Central, 100, 2740-122 Porto Salvo, Portugal
{andre.augusto,lucasvicente,manuel.d.mascarenhas}@tecnico.ulisboa.pt

Abstract. *NoSQL* databases have been ever-growing in popularity over the last decades, especially when it comes to very large-scale systems in the realms of *BigData* and the *Web2.0*. In this report, we evaluate and discuss the performance and scalability of one of the industry's standards for non-relational distributed databases, MongoDB. To achieve set goal, we make use of the *YCSB* (Yahoo! Cloud Serving Benchmark), one of the most popular benchmarking tools for *NoSQL* databases. We design a $n^{(k-p)}r$ fractional factorial experiment as a means to measure the latency and throughput of the system in our own defined workload. Moreover, we analyse the obtained results and the scalability properties according to the *Universal Scalability Law* (USL) of the most impactful factor obtained in the benchmark experiment. We also decompose *MongoDB*'s requests into a pipeline of stages and, additionally, identify and propose solutions to diminish the bottleneck.

Keywords: NoSQL · Document-Oriented · Database · Benchmark

1 Introduction

MongoDB [1] is an open-source *NoSQL* document oriented database, with a *JSON*-like format. Its key-value data model consists of a "key-value" relationship, where the value, representing the actual stored data, is indexed by a uniquely identifiable key. This simpler structure allows for schema-less data. *MongoDB* is also able to support multi-document ACID transactions, replication and horizontal scalability through sharding.

Git commit: 7af70a3fe3748be17255be8c24862200b794d096.

1.1 Justification

With the rise of cloud computing, using a relational database to store and query large amounts of dynamic data has proved to be insufficient, especially in large-scale and highly concurrent applications, such as social networks. *NoSQL* databases have gained a lot of traction, as a means to fulfill these demands, however there is a large offer of *NoSQL* databases with different data models while providing different guarantees. With this in mind, determining the right

database for an application's expected workloads is key. We have decided to evaluate and measure *MongoDB*'s scalability and performance, since its features are the richest and most like those present on a relational database, therefore making it interesting to contrast with what we have been working with the most so far.

2 System Description

As described previously in section 1, *MongoDB* [1] is a document oriented distributed database, with *JSON*-like documents as its basic unit of data. Each document consists of a set of key-value pairs that can vary from document to document providing total flexibility regarding the way to store data — dynamic schemas. The main components of the system are: *mongod* which is the core database process; *mongos* which is the controller and query router for sharded clusters and *mongo* the interactive shell.

2.1 MongoDB Characteristics

In order to understand how *MongoDB* works, in this section we focus on describing some core features that compose the system, as a means to identify the service it provides. Having stated that, we focus on the transactions' properties and levels, the data replication process and the various storage engines supported by *MongoDB*.

Transactions [2] To begin, it is crucial to note that *MongoDB* adheres to ACID principles, which is critical for understanding the ideas that follow. Each transaction can adopt different transaction-levels regarding the read preference [3], read concern [4] and write concern [5]. While the first one defines how clients route read operations to the members of a replica set, either to the primary or any secondary, read and write concern allows the client to have strict control of the consistency and isolation required for the application, defining the number of acknowledgments needed for a certain request to return.

Replication [6] Replication in *MongoDB* is accomplished using replica sets. A replica set is a group of *mongod* processes that keep track of the same data set. Replica sets are able to provide redundancy and high availability, on top of a level of fault tolerance against the loss of a single database server. A replica set consists of many data-bearing nodes and, if desired, one arbiter node. One and only one member of the data-bearing nodes is designated as the primary node, while the others are designated as secondary nodes. Each node type has a different role in the replication process as we can see in figures 1 and 2:

– **Primary**: By default, receives all write and read operations. Records all changes to its data set in its operation log (*oplog*).

- **Secondary**: Replicates the primary's *oplog* and applies the operations to its data set, such that it matches with the primary's data set. In the event that the primary is not available, an eligible secondary will hold an election to elect a new primary.
- **Arbiter**: This node does not have a copy of the data set and cannot become a primary. However, it is able to cast a vote. When a cost limitation prevents the installation of another data-bearing node, arbitrator nodes prove useful. To be noted however that, If a secondary is unavailable or behind in a three-member primary-secondary-arbiter (PSA) architecture, requests with write concern "majority" might create performance concerns.

Secondary members of a replica set synchronize data from other members to replicate the primary's *oplog*, and therefore keep up-to-date copies of the shared data set. This synchronization method requires secondaries to initially select their synchronization sources, referred to as a replica's sync source, which happens right after their entrance on the replica set. After that, *sync from* sources sends a continuous stream of *oplog* entries from a sync source to their syncing secondaries. If the system parameter chaining is enabled, secondary nodes can function as sync sources for other secondary nodes; if it is disabled, only the primary node can act as a sync source.

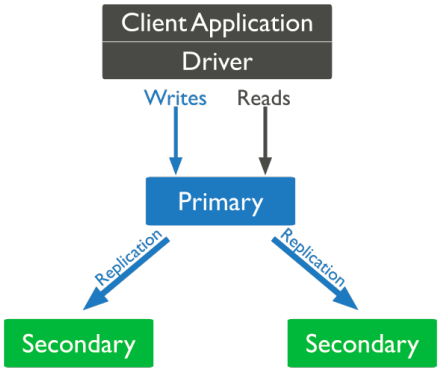

Fig. 1. Primary-Secondary-Secondary (PSS) architecture.

Storage Engine *MongoDB*'s storage engine is the main component in charge of data management, more specifically it controls how data is saved, both in memory and on disk. *MongoDB* offers a range of storage engines to fulfill different workloads.

- **WiredTiger Storage Engine** [7] (Default): Recommended for new deployments, since it is well-suited for most workloads. *WiredTiger*'s main

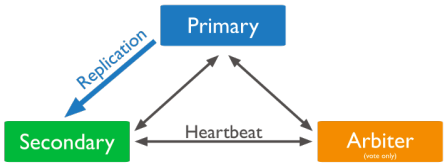

Fig. 2. Primary-Secondary-Arbiter (PSA) architecture.

characteristics include: a document-level concurrency model, checkpointing and compression.
- **In-Memory Storage Engine** [8]: Keeps documents in memory, rather than on disk to ensure more predictable data latencies. To be noted that this particular storage engine does not persist data after process shutdown.

To further increase the durability guarantees, *MongoDB* utilizes *write ahead logging* to on-disk journal files, providing resilience in case of a failure. The *WiredTiger* storage engine already uses checkpoints to maintain a consistent view of data on disk and to allow *MongoDB* to recover from the last checkpoint. However, if *MongoDB* were to crash suddenly in between checkpoints, it would not be able to recover data lost after the last checkpoint, that is why journaling is necessary. There is no separate journal when using the in-memory Storage Engine, since its data is kept in memory.

3 Experimental Design

For the evaluation process of MongoDB, we have decided to select the system parameters which would have the most impact on its performance. The reasons for such decision can be traced back to our stage 01 experimental results, in which we concluded that, for our specific workload, the replication stage of the pipeline was found to be the bottleneck.

3.1 Metrics, Factors and Levels

As for the criteria used to evaluate the performance of the system, we have decided to select the number of executed operations per second (**throughput**) and the response time (**latency**).

The following section will be going over the 7 factors and the 2 levels selected for each one, and will be structured in the following manner:

1. Description of the factor.
2. Justification behind the selection of the factor.
3. Level's description.

Write Concern (A) Write concern describes the level of acknowledgment requested from MongoDB for write operations. As explained in stage 01, this parameter allows us to measure the impact of different consistency levels on the system.

- **Level -1**: Write Concern Majority. This level forces MongoDB to only return to the client after write operations have propagated to the calculated majority of the data-bearing voting members. In our case, using a PSS or PSA architecture, a write must propagate to the primary and one secondary has to acknowledge.
- **Level 1**: Write Concern 1. This level forces MongoDB to return to the client as soon as a write propagates to the primary.

Replica Writer Thread Range (B) Replica writer thread range refers to both the minimum and maximum number of threads used to apply replicated operations in parallel. The default maximum number of threads is set by MongoDB as 16, however since we are now deploying our replica set to the cloud with machines providing higher levels of vCPUs it makes sense for us to provide MongoDB with a higher threshold of threads utilized for applying replication operations.

- **Level -1**: 0 to 16 threads.
- **Level 1**: 0 to 128 threads.

Read Concern (C) Read concern describes the level of acknowledgment requested from MongoDB for data to be returned in read operations. Similarly to what was mentioned regarding the write concern level, this parameter also allows measuring the impact of different consistency requirements on the system.

- **Level -1**: Read Concern Local. This level forces MongoDB to return data from the given query from the instance being requested, with no guarantee that the data has been written to a majority of the replica set members.
- **Level 1**: Read Concern Majority. This level forces MongoDB to return data from the given query that has been acknowledged by a majority of the replica set members.

Read Preference (D) Read preference describes how read requests are routed to members of a replica set. By default, read operations are directed to the primary member in a replica set. However, with this parameter we are able to force read operations to be redirected to secondary nodes, allowing us to determine the potential gains of offloading these types of requests.

- **Level -1**: Read Preference Primary Preferred. Operations read from the primary, but if it is unavailable, operations read from secondary nodes.
- **Level 1**: Read Preference Secondary Preferred. Operations read from secondary nodes, but if no secondary nodes are available, operations read from the primary.

Replica Batch Limit (E) Replica batch limit allows us to set the maximum oplog application batch size in bytes. As described in section 2.1, during replication the oplog entries are propagated across data-bearing nodes. By default, MongoDB already utilized the maximum which is 100MB, however with this parameter we are able to determine the negative impact of lowering the batch size.

- **Level -1**: 50MB.
- **Level 1**: 100MB.

Replica Node Configuration (F) As described in section 2.1, there are three types of nodes: Primary, Secondary and Arbiter. In stage 01, we have only set up a three node replica set with a Primary-Secondary-Secondary configuration (PSS), as shown in Figure 1. However, since arbiter nodes do not contain any dataset and simply vote, we have also set up a Primary-Secondary-Arbiter configuration (PSA), as shown in Figure 2.

- **Level -1**: Primary-Secondary-Secondary configuration (PSS).
- **Level 1**: Primary-Secondary-Arbiter configuration (PSA).

Chaining (G) As described in section 2.1, by default secondary nodes set the primary node of their replica set as their sync source, meaning they replicate directly from the primary node. With chaining allowed, secondary nodes may replicate from another secondary member instead of from the primary. This parameter was selected since chained replication can help reduce load on the primary. However, chained replication can also result in increased replication lag, depending on the network topology.

- **Level -1**: Chaining disabled.
- **Level 1**: Chaining allowed.

3.2 Experiment

Ideally, the experimental design would be full factorial, however this would result in a total of $2^7 = 128$ experiments. Each experiment would also require a number of repetitions to improve result accuracy, which would not be feasible to perform in a timely manner. Therefore, we opted to confound 4 factors that, theoretically, would have less meaningful impacts on the performance comparatively to the other 3. Considering the identified factors and their respective levels, we designed a $n^{(k-p)}r$ fractional factorial design where n = 2 levels, k = 7 factors, p = 4 confounded factors and r = 5 repetitions, amounting to a total of 8 experiments, which translate to the same number of system configurations. In the stage 01 experiments, we concluded that the replication stage of the pipeline was found to be the bottleneck. This was highlighted by having the *Write Concern* factor with level *Majority*, as opposed to having it as *1 Acknowledgment*. This lead us to

suspect that *Write Concern* would be the most contributing factor. Additionally, we expected that *Replica Writer Thread Range* would also have a significant impact, since it has a direct effect on the data replication process, which is our bottleneck stage. Lastly, we hypothesized that the *Read Concern* factor could also be contribute significantly, analogous to the *Write Concern* applied to the read flow of the pipeline. Consequently, we concluded that these 3 factors ought not to be confounded with any other. Ideally, insignificant effects would be used for confounding, however with 7 factors and only 3 isolated factors we are left with the 4 possible unique combinations between factors. Each one of these will be used in one of the 4 confounding factors. We determined factors D to G should be confounded as such:

- D confounded with A and B ($A * B$).
- E is confounded with A and C ($A * C$).
- F is confounded with B and C ($B * C$).
- G is confounded with A, B and C ($A * B * C$).

From this fractional factorial design, we created the configurations described in the sign table in Table 2. The evaluation process was conducted utilizing 16 client threads, which is an approximation of the maximum useful size obtained in the experiments conducted previously, in the local environment.

3.3 Workload

Due to time concerns, we have decided to use an already built and proven *NoSQL* database benchmarking tool, *YCSB* (Yahoo! Cloud Serving Benchmark) [9], to run our workload. In order to completely analyze the system, we have selected a workload composed of multiple operations, traversing every component and paths of a hypothetical pipeline of the system. With this in mind, the workload we propose is diverse performing 10,000 target operations over a data set of 1000 records, whilst being composed by 50% single document reads, 20% scans of 100 documents, 15% update and 15% insert requests. The data size used was 1KB records, with 10 fields of 100 bytes each, obtained in the *CoreWorkload* module of YCSB. The requests performed have a record selection distribution following Zipf's law, meaning the frequency of request selection is inversely proportional to their rank (or popularity) in the frequency table.

3.4 Cloud Deployment Setup

Having analyzed the scalability and performance of *MongoDB* on our local environment, we decided that, in order to properly determine some of the most relevant system performance factors regarding throughput and latency, it would be crucial to evaluate *MongoDB* in a cloud environment. Due to limitations in billing/credit on various cloud providers, the decision was made to deploy the infrastructure on the Google Cloud Platform (GCP) [10], which had fewer constraints given the existing coupons. From the services offered by GCP, the

only one that suited our needs was Google Kubernetes Engine (GKE) [11]. With this in mind, unlike the stage 01, where Docker Swarm was used, we opted for Kubernetes [12] as a container-orchestration tool. To help automate the deployment process, Terraform [13], an infrastructure as code tool, was used to create the necessary components while maintaining configuration consistent. We use GCP to take advantage of a tier that gave us access to several *e2-highmem-8* type machines, where each one offers 8 vCPUs and a 64 GB memory. If for any reason, in the future, we would like to switch cloud providers, Terraform gives us that flexibility since it deploys a Kubernetes cluster in a provider-agnostic manner.

The Kubernetes cluster was set up in the europe-west1 region, with three nodes (each in its own zone within the specified region) hosting the MongoDB pods and service. As in stage 01, we deployed 3 MongoDB replicas, each one in a different pod, containerized using *MongoDB's* version *5.0* official docker image [14]. Each MongoDB node is using the *WiredTiger* storage engine with journaling active and has its own persistent volume providing stable storage of 1 GB.

Regarding the execution of the experimental evaluation, we decided to create an additional pod allowing us to remove any existing bottleneck caused by the tool itself. This pod was built from a docker image [15] created with *YCSB* alongside several scripts used to automate the execution process. This image was pushed to Docker Hub, a container image registry.

4 Results

In this section, we present and analyse both the benchmark, performance and scalability experimental results. Based on this results, we also decompose a request into a pipeline of stages, identify a probable bottleneck stage and, finally, propose a solution to mitigate this problem.

4.1 Benchmark

For readability purposes, we divided the results into 2 tables as follows:

- **Sign Table**: Contains the different experiments' configurations and the impact percentages of each factor for each metric (Table 2).
- **Results Table**: Contains the obtained results of each repetition for each experiment (Table 3) (which was already composed by the average of 5 repetitions) and the derived mean throughput and latency.

We were able to determine a percentage of error for throughput and latency experiments of 0.27% and 0.70%, respectively, therefore we are only able to ascertain with confidence the effect of factors with an impact higher than said error percentage. We suspect the percentage of errors achieved can be attributed to cloud provider instability caused by "noisy neighbours". We infer this conclusion

based on four independent repetitions of the full benchmark experiment, which were conducted on distinct days, and the results across experiments had noteworthy variances. Ideally the percentage of error would be substantially lower than each factor and the effect of their interaction, however we were not able to achieve such objective. The benchmark experimental results we are presenting in this report had the lowest percentage of errors for both metrics.

From our experiments, we were able to determine, with very high confidence, that the main contributing factor is *Write Concern*, as we suspected from the experiments done in our local environment, with an impact of 91.61% and 90.67%, on latency and throughput respectively, accounting for almost the entirety of the effect on both metrics. The confounding of the factor *Read Preference* (D) with the effect of the interaction between *Write Concern* and *Replica Writer Thread Range* (A and B) also had a meaningful impact of 5.07% and 5.21% on latency and throughput, respectively. However, since both the A and B are factors whose effects directly affect the data replication stage, we believe that this impact of roughly 5% to be the resulting effect of the interaction of A and B and, therefore, the impact of the *Read Preference* to not be very significant. The confounding of the factor *Replica Node Configuration* (F) with the effect of the interaction between *Replica Writer Thread Range* and *Read Concern* (B and C) had the third-highest impact of 1.83% and 1.62% on latency and throughput, respectively. Due to the confounding and the fact that both *Replica Node Configurations* have the same majority count, we are under the assumption that both B*C and F contribute in a similar manner for this impact percentage. Lastly, we can only confidently say that the *Replica Writer Thread Range* (B) also has a measurable effect of 1.57% on the throughput of the system, given our error percentage for throughput of 0.27%. All other factors' effect can either be a very small effect on either metric or simply be a consequence of the existing error, thus we can not confidently ascertain any conclusions about them.

4.2 Scalability and Performance

In this section, we showcase and analyse the scalability and performance experimental results. These experiments were done varying the *Write Concern* with both levels *Majority* and *1*, since this factor had the most impact (above 90%) on the performance of the system as per our benchmark results, in section 4.1.

Using *gnuplot*, we have plotted the scalability experimental results of our workload for both Write Concern levels, alongside the theoretical *Universal Scalability Model*, as shown in Figure 4. The specific *USL* parameters obtained through our results can be seen in Table 1. Additionally, we also plotted the latency scalability for read (Figure.5) and write (Figure.6) operations on both *writeConcern* levels.

Further, function analysis was done using *Wolfram Alpha* to determine the impact of both *serial* and *crosstalk* components with the different levels of *Write Concern*. The serial component of the system, also referred to as contention, ends up limiting asymptotically its speedup. For our workload, with $w = majority$ the maximum throughput achieved was of approximately 3365 ops/sec, while with w

Table 1. USL values according to write concern value, either $w = 1$ or $w = majority$.

Parameter	w = 1	w = majority
λ	1063.4414341392	330.0045683709
σ	0.1488820052	0.0356544418
κ	0.0031035573	0.0010436999

$= 1$ the maximum throughput achieved was of approximately 4278 ops/sec. The *crosstalk* portion limits the maximum system achievable size. Using this formula $N_{\max} = \lfloor \frac{\sqrt{1-\sigma}}{k} \rfloor$, we are able to calculate $\mathrm{N_{max}}$ of 30 client threads with $w = majority$ and $\mathrm{N_{max}}$ of 17 client threads with $w = 1$.

Intuitively, we would expect to observe higher values on both *serial* and *crosstalk* portions with *Write Concern Majority*. However, this is not the case, only the performance portion of the *USL* is lower when *Write Concern* is set to *Majority* relative to when it is set to *1*. With this in mind, our discussion is based on the combining effects of each portion. As we have said previously, the serial component of the system, ends up limiting asymptotically its speedup. This, combined with the greater performance portion of $w = 1$, results in a speedup of the observed higher maximum throughput of approximately 1.27x. We expected this sort of improvement with $w = 1$, since the rate of which clients are able to perform writes is much greater, and the rate of read operations is not affected by *Write Concern*. We also know that the *crosstalk* portion limits the maximum system achievable size. This, combined with the greater performance portion of $w = 1$, results in $w = 1$ reaching its maximum throughput earlier than $w = majority$. This means that more client threads are required to reach the maximum throughput with $w = majority$, approximately the double. We observe this, since a single write operation using $w = majority$ in our replica node configuration involves the acknowledgment of 2 nodes, which is a majority, whereas using $w = 1$ it is only required the acknowledgment of 1 node.

4.3 Stage Pipeline

In order to help us identify a possible bottleneck and further expand our knowledge about the system, we decomposed our workload's diversified requests into stages. This allowed us to create a pipeline to demonstrate the multiple components utilized in *MongoDB*'s system. The pipeline created was based on our interpretation of the logs, with verbosity level 5, obtained through our workload's execution, and further supplemented with research done on both *MongoDB*'s source code [16] and the manuals of the main components of *MongoDB*, the Query Engine [17] and the Storage Engine (in our case *WiredTiger*) [18]. A visual representation of the pipeline is shown in Figure 3. Firstly, *MongoDB*'s driver issues a request which gets handled by different components, based on its type.

The primary node receives a write request and immediately converts it into a transaction. Within the scope of this transaction, the primary node applies local changes by writing to the document's corresponding collection while updating any necessary information regarding the indexes. From this moment, the primary inserts the entries in its own *oplog* and commits the transaction. After this, it waits for the write concern to complete, according to the provided value (w) in the request. As specified in section 2.1, this value allows the client issuing the request to define different consistency levels summarized as:

- ***w = 1***: It only takes one replica to commit the transaction before returning to the client, thus it is sufficient if only the primary returns.
- ***w = majority***: Before returning to the client, the transaction must be committed by a majority of the system's nodes. Every secondary node accomplishes this by retrieving its sync source log entries and applying the changes to their corresponding local collections and *oplog*. A notification is then issued upstream, until it reaches the primary. In the primary, as soon as the write concern is met, in this case a majority of nodes, the operation is successful, returning to the client.

In the other hand, if there is a read request, it can be forwarded to either a primary or a secondary based on the read preference specified. Basically, the query planner checks in cache for a plan that is comparable to the one needed to respond to that request, and if it succeeds a performance evaluation, the transaction is issued. If this is not the case, many candidate plans are generated and evaluated, until one is picked as the one to execute, at which point a cache entry for this plan is created. The transaction is also issued, at this stage. The value will be read from the relevant collection as soon as the transaction starts. Additionally, the transaction will commit, and the value will be returned to the client.

4.4 Bottleneck Identification and Mitigation

Based on the experiments performed in section 4.1, for the given workload, it is possible to state with confidence that the factor Write Concern, is the most impacting on both the throughput and latency. By further analyzing the scalability detailed in section 4.2 and the profiler's logs, it is possible to extrapolate that the replication stage of the pipeline produces an enormous bottleneck to the system, as suspected in stage 01. More specifically, using *Write Concern* in a level superior to 1 requires the primary to "*wait*" for a majority of acknowledgments per write request. This implies moving the replication process onto the critical path, generating the observed bottleneck.

Considering *MongoDB* already applies batching and dallying when fetching *oplog* entries process among secondaries, and since speculation would not have much of an effect on writes, we propose applying a load shedding policy towards requests with $w = majority$. Further analysis would be needed to assert the optimal size or percentage of this type of requests that enables an optimal balance between performance and goodput.

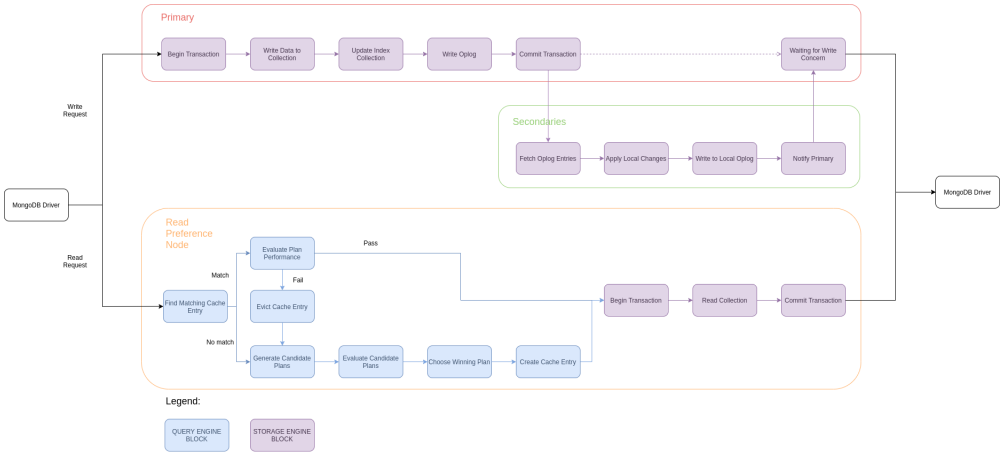

Fig. 3. Request Stage Pipeline.

5 Conclusion

Through a detailed analysis of *MongoDB's* characteristics and properties, we were able to identify the service provided by MongoDB and the several parameters which could alter its performance while providing such service. To further asses the performance of MongoDB based on its response time per request and number of executed operations per second, we designed a $2^{(7-4)}5$ fractional factorial experiment, with 7 factors being: **write concern**, **replica writer thread range**, **read concern**, **read preference**, **replica batch limit**, **replica node configuration** and **chaining**. The results of these experiments included a mean error percentage of **0.27%** and **0.7%** for throughput and latency, respectively. The evaluation conducted enabled us to conclude that the most impacting factor for the given workload is ***Write Concern***.

In addition to our benchmark experiments, we analyzed MongoDB's scalability properties with the most contributing factor *Write Concern*, according to the Universal Scalability Law. According to this evaluation the *performance coefficient* is **330.0045683709**, the *serial portion* is **0.0356544418** and the *crosstalk factor* is **0.0010436999**. We also determined that the system has a maximum throughput of **3365 ops/sec**, which corresponds to N_{max} of **30 client threads**. Moreover, we decompose a request into a pipeline of stages allowing us to clearly identify the bottleneck, which for our workload was the data replication stage. Consequently, we propose that a load shedding policy to be applied towards requests with $\boldsymbol{w = majority}$, in order to mitigate the effects of this stage on the pipeline.

References

1. MongoDB, https://www.mongodb.com/

2. MongoDB Manual - Transactions, https://docs.mongodb.com/manual/core/transactions/.
3. MongoDB Manual - Read Preference, https://docs.mongodb.com/manual/core/read-preference/.
4. MongoDB Manual - Read Concern, https://docs.mongodb.com/manual/reference/read-concern/.
5. MongoDB Manual - Write Concern, https://docs.mongodb.com/manual/core/write-concern/.
6. MongoDB Manual - Replication, https://docs.mongodb.com/manual/replication/.
7. MongoDB Manual - WiredTiger Storage Engine, https://docs.mongodb.com/manual/core/wiredtiger/.
8. MongoDB Manual - In-Memory Storage Engine, https://docs.mongodb.com/manual/core/inmemory/.
9. YCSB (Yahoo! Cloud Serving Benchmark), https://github.com/brianfrankcooper/YCSB.
10. Google Cloud Platform (GCP), https://cloud.google.com/.
11. Google Kubernetes Engine (GKE), https://cloud.google.com/kubernetes-engine.
12. Kubernetes, https://kubernetes.io/.
13. Terraform, https://www.terraform.io/.
14. Dockerhub - MongoDB Image, https://hub.docker.com/_/mongo.
15. Dockerhub - aaugusto11/ycsb Image, https://hub.docker.com/repository/docker/aaugusto11/ycsb.
16. MongoDB - Github Repository, https://github.com/mongodb/mongo.
17. MongoDB Manual - Query Planner, https://docs.mongodb.com/manual/core/query-plans/.
18. WiredTiger, https://source.wiredtiger.com/3.2.1/index.html.

6 Annexes

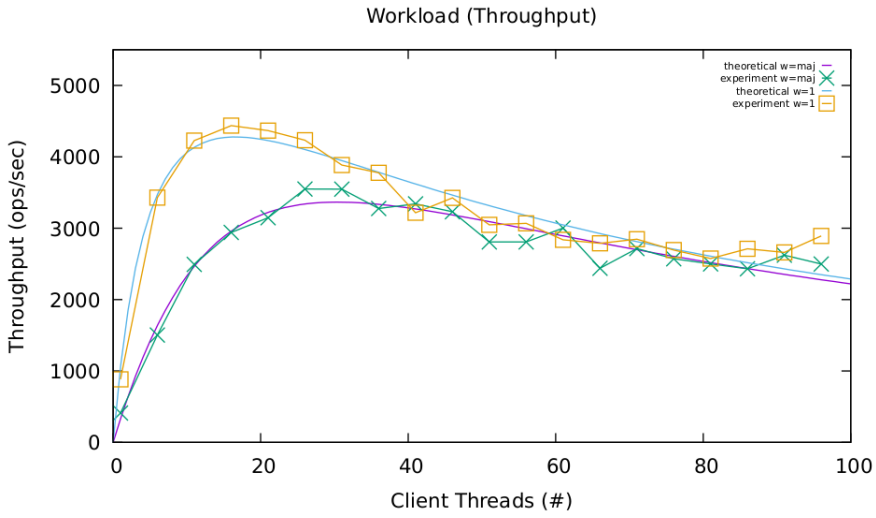

Fig. 4. Experimental and theoretical throughput for both write concerns of 1 and majority with the workload described in section 3.3.

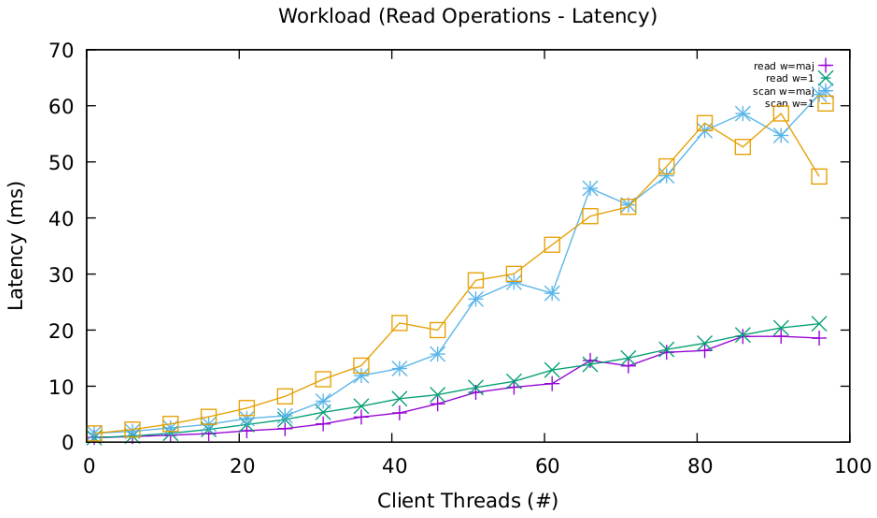

Fig. 5. Experimental and theoretical read operations latency for both write concerns of 1 and majority with the workload described in section 3.3.

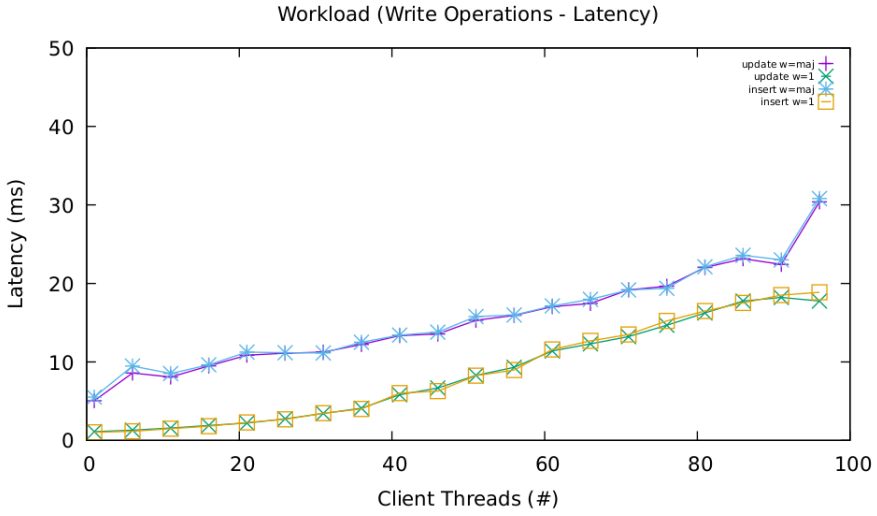

Fig. 6. Experimental and theoretical write operations latency for both write concerns of 1 and majority with the workload described in section 3.3.

Table 2. Cloud benchmarking experiment sign table for all 7 factors, 2 levels each, with 4 confounded factors, repeated each 5 times. Errors for Latency and Throughput are 0.70% and 0.27%, respectively

	Experiment	WriteConcern	ThreadRange	ReadConcern	ReadPreference	BatchLimit	NodeConfiguration	Chaining
		A	B	C	D (A*B)	E (A*C)	F (B*C)	G (A*B*C)
	1	-1	-1	-1	1	1	1	-1
	2	1	-1	-1	-1	-1	1	-1
	3	-1	1	-1	-1	1	-1	-1
	4	1	1	-1	1	-1	-1	-1
	5	-1	-1	1	1	-1	-1	-1
	6	1	-1	1	-1	1	-1	-1
	7	-1	1	1	-1	-1	1	-1
	8	1	1	1	1	1	1	1
Latency (μs)	22623.7970	-4540.923	-261.0074	-134.741	-1068.0178	281.1078	-642.0566	117.857
Throughput (ops/sec)	35252.212	5789.148	762.412	116.588	1388.036	-238.492	773.604	103.308
Latency/8 (μs)	2827.974625	-567.615375	-32.625925	-16.842625	-133.502225	35.138475	-80.257075	14.732125
Throughput/8 (ops/sec)	4406.5265	723.6435	95.3015	14.5735	173.5045	-29.8115	96.7005	12.9135
SS Latency	14068503.58	12887488.56	42578.03928	11346.96068	712913.7632	49388.49701	257647.9235	8681.420281
SS Throughput	23101837.49	20946396.6	363295.0361	8495.47609	1204152.461	35549.02129	374039.468	106725.4286
% Latency	100.00%	91.61%	0.30%	0.08%	5.07%	0.35%	1.83%	0.06%
% Throughput	100.00%	90.67%	1.57%	0.04%	5.21%	0.15%	1.62%	0.46%

Table 3. Cloud benchmarking experiment's results of Latency (μs) and Throughput (ops/sec).

Exp	Latency	Throughput	Mean Latency (μs)	Mean Throughput (ops/sec)
1	3276.390, 3274.459, 3192.172, 3263.193, 3252.314	3789.72, 3747.1, 3837.32, 3829.44, 3798.86	3251.7056	3800.488
2	2368.101, 2312.759, 2378.268, 2345.562, 2308.643	4957.94, 5022.56, 4919.56, 4992.58, 5038.44	2342.6666	4986.216
3	3615.464, 3576.895, 3821.308, 3618.074, 3585.442	3494.56, 3511.92, 3363.8, 3504.06, 3508.2	3643.4366	3476.508
4	2141.965, 2146.032, 2162.843, 2132.020, 2124.441	5325.22, 5297.78, 5270.62, 5321.42, 5307.96	2141.4602	5304.600
5	3313.612, 3293.579, 3398.101, 3361.189, 3322.128	3763, 3751.34, 3657.52, 3726.28, 3710.28	3337.7218	3721.684
6	2532.004, 2501.415, 2478.436, 2511.347, 2528.339	4708.88, 4742.9, 4771.72, 4749.66, 4709.4	2510.3082	4736.512
7	3322.205, 3265.015, 3298.097, 3348.317, 3513.846	3749.2, 3780.18, 3750.78, 3770.68, 3613.42	3349.4960	3732.852
8	2074.475, 2018.602, 2049.619, 2046.823, 2045.491	5464.52, 5518.98, 5469.92, 5506.64, 5506.7	2047.0020	5493.352