MongoDB - ESLE Report 01 - 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 analyse the scalability properties of one of the industry's standards for non-relational distributed databases, *MongoDB*. We make use of the *YCSB* (Yahoo! Cloud Serving Benchmark), one of the most popular benchmarking tools for *NoSQL* databases, with our own defined workload from which, we observe the resulting latency and throughput of the system. Moreover, we analyse the obtained results and analyse the scalability properties according to the *Universal Scalability Law* (USL). 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. Supports multi-document ACID transactions, replication and horizontal scalability through sharding. **Git commit:** c6037ddfbafe6d033b1645163da20ce2128a7157.

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 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

2.1 MongoDB Characteristics

Transactions [2] To begin, it's 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.

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.

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 select their synchronization sources, which happens right after their entrance on the replica set. After that, *sync from* sources sends a continuous stream of *oplog* entries to their syncing secondaries.

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 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.

Journaling [9] To further increase the durability guarantees, *MongoDB* utilizes *write ahead logging* to on-disk journal files, providing resilience in case of a failure.

2.2 Selected Architecture

For our experiment setup, we have decided to follow the production guidelines expressed in *MongoDB*'s manual [10]. For stage 01 of the project, we are using a replication scheme, with a single replica set following a primary-secondary-secondary (PSS) architecture and journaling active, as shown in Figure 1. We are not using any sharding method, however we do intend to conduct further experiments with sharding in stage 02. Each node is using the *WiredTiger* storage engine. Each *MongoDB* database node is a container, with Docker as its container runtime, built using *MongoDB*'s *4.2* version official docker image [11]. Regarding container configuration, each container is setup with a persistent volume for its data set, maximum log level of verbosity (level 5). As for our container orchestration tool, we have decided to use Docker Swarm.

2.3 Selected Workload

Due to time concerns, we have decided to use an already built and proven NoSQL database benchmarking tool, *YCSB* (Yahoo! Cloud Serving Benchmark) [12], to run our workload. In order to completely analyse the system, we have selected a workload composed of multiple operations, traversing every component and paths of an 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 with a write concern $w = majority$, to further analyse the replication component on the critical path. 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. 3.2.

3 Results

3.1 Scalability and Performance

Using *gnuplot*, we have plotted our workload results, alongside the theoretical *Universal Scalability Model*, as shown in Figure 3. The specific *USL* parameters obtained, through our results, were the following:

- λ — Performance Coefficient = 92.4600607815
- σ — Serial Portion = 0.6345138839
- k — Crosstalk Factor = 0.0015122801

Further, function analysis was done using *WolframAlpha* to determine the impact of both *serial* and *crosstalk* components. The serial component of the system, also referred to as contention, ends up limiting asymptotically its speedup. For our workload, the maximum throughput achieved was of approximately 136 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 the $\mathrm{N}_{\max}$ of 15 client threads.

3.2 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 [13] and the manuals of the main components of *MongoDB*, the Query Engine [14] and the Storage Engine (in our case *WiredTiger*) [15]. A visual representation of the pipeline is shown in Figure 2. 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. (Horizontal path)
- ***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 the its sync source log entries and applying the changes to the 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.

3.3 Bottleneck Identification and Mitigation

Given requests with write concern w = *majority*, by analysing the latency plot and the profiler's logs, we noticed high values of latency and a large amount of wait time in between requests due to sync, which would indicate that the replication step could be the bottleneck. So, in order to investigate this hypothesis and verify if the replication factor is indeed the bottleneck, we executed the same workload with a write concern value of 1. Both trials used the same workload described in section 2.3, but with active replication, the benchmark took roughly 1 hour and 20 minutes to execute, compared to 10 minutes without it. The comparison can be made by looking at Figures 3 and 4 regarding throughput and latency, respectively. We can clearly see the increase of throughput and decrease in latency by roughly 10 times when setting write concern to $w = 1$, presenting a new max value of throughput, of approximately 1487 operations/second, using 13 client threads. Table 1 shows an interesting detail about this plot which is that the values of σ and κ are approximately the same, whereas the values of λ have a significant difference, being the reason for the disparity of the values. We can conclude that in either configurations, the scalability of the system is approximately the same, thus the difference is explained based on a worse performance when using w = *majority*. Considering *MongoDB* already applies batching and dallying when fetching *oplog* entries process among secondaries, and since speculation wouldn't have much effect on writes, we propose applying a load shedding policy towards requests with w = *majority*. Further analysis would be needed to to assert the optimal size or percentage of this type of requests that enables an optimal balance between performance and goodput.

4 Conclusion

In this paper, we have analysed the scalability and performance properties of *MongoDB*, according to the Universal Scalability Law, where the *performance coefficient* is 92.4600607815, the *serial portion* is 0.6345138839 and a *crosstalk factor* is 0.0015122801. We also determined that the system has a maximum throughput of 136 operationssecond, which corresponds to $\mathrm{N}_{\mathrm{max}}$ of 15 client threads. Moreover, we decompose the request handling into a pipeline of stages, analyse it and conclude that for our workload, the data replication process is the bottleneck stage, as it severely increases the *performance coefficient* of the system. Consequently, we propose that a load shedding policy, towards requests with w = *majority*, is applied in order to mitigate the effects of the the data replication 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. MongoDB Manual - Journaling, https://docs.mongodb.com/manual/core/journaling/.
10. MongoDB Manual - Production Notes, https://docs.mongodb.com/manual/administration/production-notes/.
11. Dockerhub - MongoDB Image, https://hub.docker.com/_/mongo.
12. YCSB (Yahoo! Cloud Serving Benchmark), https://github.com/brianfrankcooper/YCSB.
13. MongoDB - Github Repository, https://github.com/mongodb/mongo.
14. MongoDB Manual - Query Planner, https://docs.mongodb.com/manual/core/query-plans/.
15. WiredTiger, https://source.wiredtiger.com/3.2.1/index.html.

5 Annexes

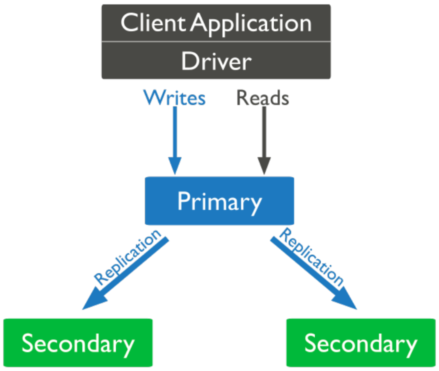

Fig. 1. Primary-Secondary-Secondary (PSS) architecture.

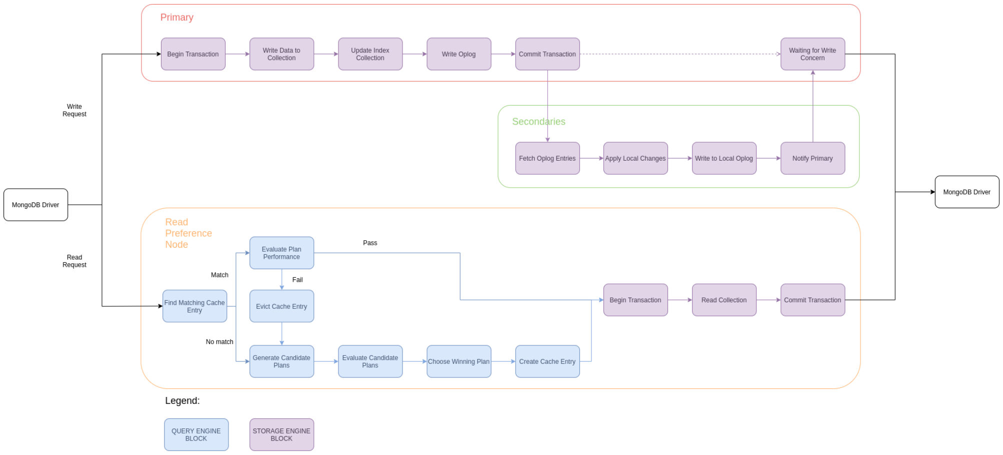

Fig. 2. Request Stage Pipeline.

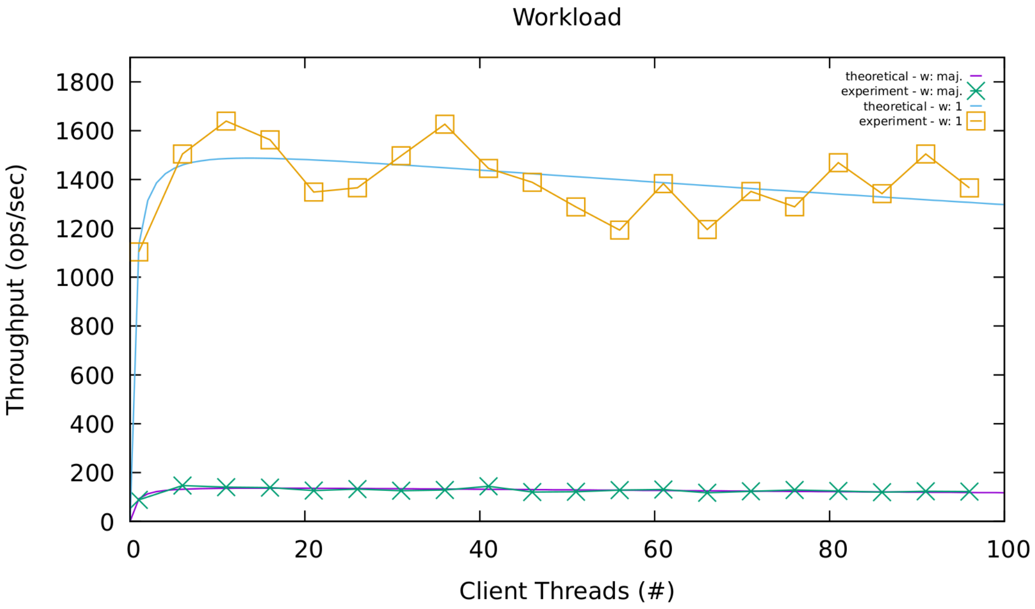

Fig. 3. Experimental and theoretical throughput for both write concerns of 1 and majority with the workload described in sub section 2.3.

Table 1. USL values according to write concern value, either *w = 1* (Primary) or *w = 1* (Majority).

Parameter	Primary	Majority
λ	1131.1959551617	92.4600607815
σ	0.7211525602	0.6345138839
κ	0.0014960699	0.0015122801

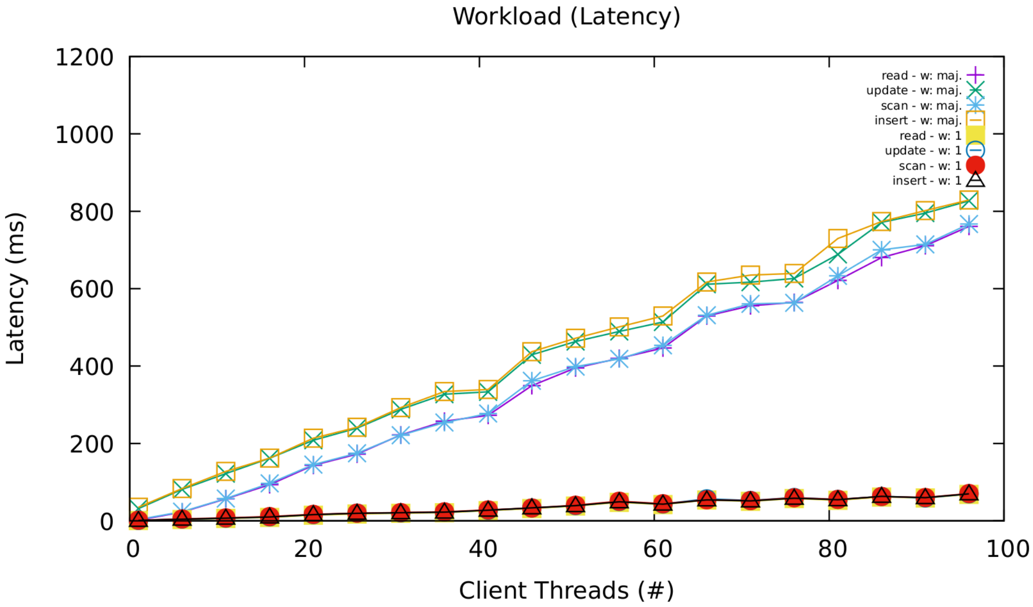

Fig. 4. Experimental and theoretical latency for both write concerns of 1 and majority with the workload described in sub section 2.3.