A simulation tool for a large-scale NOSQL database

Conference Paper · April 2019

DOI: 10.22360/springsim.2019.hpc.001

4 authors:

Juan ovando Leon
University of Santiago, Chile
6 PUBLICATIONS **17** CITATIONS
SEE PROFILE

Luis Veas-Castillo
University of Santiago, Chile
7 PUBLICATIONS **42** CITATIONS
SEE PROFILE

Mauricio Marín
University of Santiago, Chile
163 PUBLICATIONS **1,208** CITATIONS
SEE PROFILE

Veronica Gil Costa
Universidad Nacional de San Luis
110 PUBLICATIONS **642** CITATIONS
SEE PROFILE

A SIMULATION TOOL FOR A LARGE-SCALE NOSQL DATABASE

Gabriel Ovando-Leon
Luis Veas-Castillo
Mauricio Marin

CeBiB, Universidad de Santiago de Chile
Santiago, Chile
{luis.veasc,juan.ovando,mauricio.marin}@usach.cl

Veronica Gil-Costa

Universidad Nacional de San Luis
San Luis, Argentina
gvcosta@unsl.edu.ar

ABSTRACT

The amount of data on the Web and the number of users accessing those data drastically grows every year. Web-based services have to be able to receive and process those requirements within the shortest possible time. Additionally, they have to manage sudden and unpredictable peaks of user requirements for accessing the data. To tackle these problems, NoSQL databases have been designed to provide scalability and good performance. However, it is important to know in advance the number of resources (i.e. computers, network capacity) to efficiently deploy a database application. In this work, we propose a simulator for a NoSQL database named MongoDB. Our simulator aims to detect saturation levels of the resources to ensure an efficient execution of MongoDB upon different users demands and upon failures of resources.

Keywords: MongoDB, Performance Evaluation, NoSQL Database.

1 INTRODUCTION

During the last decade, Web applications have grown drastically along with the amount of data and users. The way data is treated has changed significantly. More and more data are being collected and more and more users are accessing this data at the same time. This means that scalability and performance have become real challenges for schema-based relational databases. To tackle these problems, companies providing large scale Web-based services like Google, Amazon and Facebook developed new solutions like BigTable (Chang et al. 2008), DinamoDB (DeCandia et al. 2007) and MongoDB (Copeland 2013). These are NoSQL databases that support an unstructured form of storage.

In particular, MongoDB is an open-source multiplatform NoSQL database system. It is a solution designed to improve the scalability of the operations performed on large-scale data. Scalability is achieved by the deployment of different components like a message router, a task and workload manager and a demon that control the access to the database, among others. It uses a JSON (JavaScript Object Notation)/BSON (Binary JavaScript Object Notation) data document model which is intended to be easy to program, easy to handle and offers high performance by internally grouping relevant data. Each record or data set is called document, which can be grouped into collections. Documents can have different structures. Indexes for some attributes of the documents can be created to speed-up the execution of some operations like search or update.

The MongoDB project was started in 2007 by the software company 10gen, which created the product based on the word "humongous". In 2009, it was released, and later 10gen changed the name of its company to

MongoDB, Inc. MongoDB has proven to be scalable and it is frequently used for mobile applications, content management, real-time analysis and applications related to the IoT (Internet of Things) (Wei-ping, Ming-xin, and Huan 2011)(Abramova and Bernardino 2013) (Mailewa Dissanayaka et al. 2017).

In this work, we propose a discrete-event simulator devised to estimate the performance and scalability of MongoDB-based applications. We aim to provide a tool for estimating the resource capacities (i.e. number of computers) and to detect saturation levels under different user demands and upon possible failures without actually installing the MongoDB database.There are some tools like the MongoDB Schema Simulator (https://www.npmjs.com/package/mongodb-schema-simulator) used to generate different workload and monitoring the performance achieved by a real implementation of MongoDB. The MongoDB Schema Simulator is a benchmark tool used to emulate an e-commerce scenario and to generate a real trace of requests, but it requires to actually install and deploy MongoDB. Therefore, to the best of our knowledge, this is the first real simulator devised for the MongoDB database.

Our simulator is developed on top of the LibCppSim library (Marzolla (2004)). It provides a hierarchical design of the components of the MongoDB database, so the number of resources required to deploy the database can be easily scaled-up or down. The simulation cost of each operation (i.e. insert, update, delete, etc.) is adjusted through benchmark programs (http://www.tpc.org/tpce/) (Nambiar et al. 2015). Our experimental results show that our simulator is capable of reproducing the behavior of MongoDB with a small error. Moreover, we show how our simulator can be used as a tool to evaluate the performance and scalability of the database when it is deployed on a cluster of computers.

The remainder of this paper is organized as follows. Section 2 presents the architecture of the MongoDB database. Section 3 details our proposed simulator. Section 4 presents the simulation setup and the results and finally, Section 5 brings the conclusions.

2 MONGODB: A NOSQL DATABASE

MongoDB is a noSQL database written in C++, where the data is not stored in tables, but rather flat files are used in JSON (JavaScript Object Notation) format. Additionally, it supports a dynamic scheme named BSON (Binary JavaScript Object Notation). It is a binary representation of data structures and maps, designed to be lighter and more efficient than JSON. MongoDB has a decentralized structure which allows to use distributed schemes. This feature provides horizontal scalability, meaning that we can use more computers with less computing capacity, instead of having to resort to a single, more powerful machine.

MongoDB is a document-based database where the data is organized in databases and each one holds collection of documents. Documents are composed by different fields.

In Figure 1 we show the architecture of MongoDB (to be deployed in a cluster of processors) which contains the following components:

- Mongod: Is the main process (daemon) to manage the access to the data. It is the database server.
- Mongos: Is the routing process between a user application and the MongoDB database.
- config.server: Stores the metadata to locate the data of the operations required by the client.
- Replica set: group of mongod processes that store the same copies of the data. A *Primary* replica set stores the main copies. A *Secondary* replica set makes copies of the primary replica set. In case the primary replica set fails, an *Arbitrer* service is used to select the secondary replica set.
- Shard: It is a replica set that keeps a portion of the database. It is used to distribute the database among different processors of the cluster.

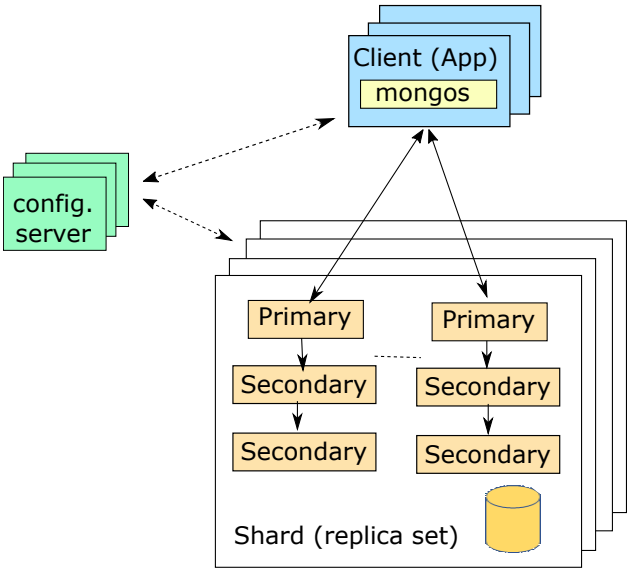


Figure 1: Components of the architecture used by MongoDB to be deployed on a cluster of processors.

A client (App server) connects to a database router (mongos). Then, the mongos connects to configuration servers (config.servers) to determine where it is located the requested data or where to write the new data. Mongos connects to the Shard (set of replicas) responsible for a range of values on some indexed values in the database, to fetch the requested data and then return it to the user application. It requires to set up an index on a unique value on the collection used to balance the load of data among multiple replicas sets. Shards are used to scale data among more than one server with MongoDB and each one has a primary mongod (MongoDB server) and zero or several secondary mongod. MongoDB also runs a balancer in background to rebalance the values and data across the replica sets in real time.

MongoDB supports different configurations. The simplest configuration is “standalone” (a single database source). The “replica” configuration uses a single database with replicas. The “shard” configuration is used to deploy a distributed database and the ”shard-replica” configuration allows to deploy a distributed database with replicas.

There are two types of replication schemes in MongoDB: Master-Slave Replication and Replica-Set Replication. Replica-Set provides better automation and handling for failures. Regardless of the replication scheme, at any shard, only one replica acts as a primary. All the writing and reading operations are sent to the primary replica and then they are distributed uniformly, (if necessary), to the other secondary replicas of the set.

3 SIMULATING MONGODB

3.1 Simulation Model

The simulation model uses a processes and resources approach. Processes represent threads in charge of processing high cost operations executed in the Web search engine. Resources are shared artifacts such as posting lists, data structures for partial results, global variables, RAM memory, cores and processor caches, and communication interfaces and switches for the Fat-Tree network. Our simulator programs are implemented on top of the LibCppSim library (Marzolla (2004)). This library manages the creation/removal of co-routines as well as the future event list. The library ensures that the simulation kernel grants execu-

tion control to co-routines in a mode of one co-routine at a time. Co-routines are activated following the sequential occurrence of events in chronological order.

Co-routines represent process that can be blocked and unblocked at will during simulation by using the operations **passivate()**, **hold()** and **activate()**. When a hold(Δ_t) operation is executed, the co-routine is paused for a given amount of Δ_t units of simulation time representing the dominant cost of a task. Once the simulation time Δ_t has expired, the co-routine is activated by the simulation kernel. The dominant costs come from tasks related to ranking of documents, intersection of posting lists and merge of partial results. These costs are determined by benchmark programs implementing the same operations executed on single processors. The benchmarks for every CRUD operation (and for every request size) can take hours to complete, while a simulation of 48 hours takes 30 minutes to complete. Additionally, a co-routine executes a passivate() operation to stop itself, indicating it has paused its work. Finally, a co-routine in passivate state can be activated by another co-routine using the activate() operation.

Typically, the implementation of large-scale databases enforces the scheduling policy of one single thread per core to prevent from saturation at processor level. Thus, the incoming requests (operations like search, update, insert) are queued up at the assigned thread to receive service which is also reflected in the corresponding simulation. The requests compete for accessing the threads and through them they get access to processor resources.

To simulate the network, the messages are divided into packages of fixed size. Each package includes the data, a header with sender and receiver identifiers, and the number of packages forming the message. All input packages go to the same input queue. Benchmark programs are devised to evaluate the cost of different communication patterns such as multicast, broadcast, and point-to-point messages.

3.2 MongoDB Simulation Tool

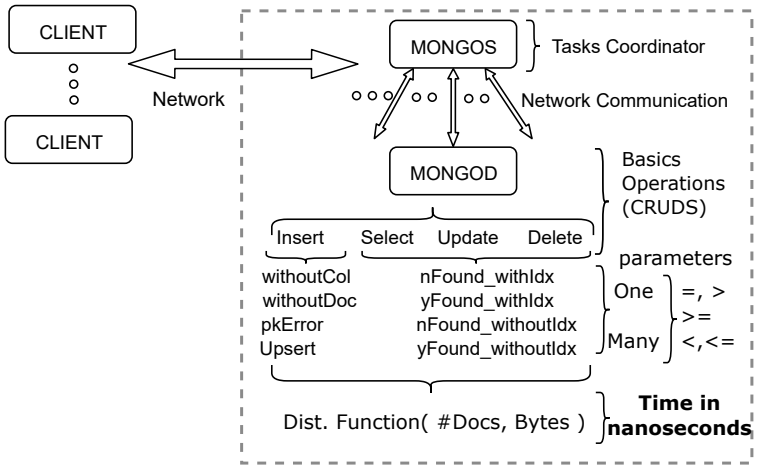


Figure 2: Database operations and parameter configurations.

Our proposed simulator includes the main components of MongoDB (such as the Mongos and Mongod), the operations performed on the database and the infrastructure and architecture model. Figure 2 shows the simulated components of MongoDB. The MongoDB components and the client applications can run on the same computer or in different computers. In this last case, the client application connects through the network. The infrastructure -as network, number of servers and their characteristics-, architecture -as databases models and collections models (primary key, partitions, replicas) and the relations between them (database instance belonging to a server) are specified in a JSON file, which is easy to modify for deployment testings

purposes. To simplify the simulation model we do not simulate the config.server but we include the costs associated to the scheduling of tasks in the simulated mongos component. Each basic operation (Create, Read, Update, Delete, Search - CRUDs) can be executed on the database with different parameters. The insert operation can be executed without a collection, without a document or with a primary key error (pkError). A special case of the update operation (Upsert) is when the client search for a data but it is not found in the database. Then, the data is inserted. The search, update and delete operations can be executed with or without an index. The parameters "One/Many" are used to determine the amount of data to be retrieved or processed. E.g., when a select operation (implemented with a find instruction in MongoDB) is set with the parameter "One", it retrieves the first data that satisfies a given condition like ≥ 4 ("db.collection.findOne({ "data": { "$gte" : 4 } })"). Notice that "$gte" is a MongoDB instruction. A select operation set with the parameter "Many" retrieves all data that satisfies a given condition. Each operation involves processing a given number of documents (#Docs) and for each document we use a uniform distribution between 100 bytes and 16MB, to set the number of bytes. Finally, a normal distribution function is used to estimate the execution time of each operation in nanoseconds.

In Figure 3 we show the general scheme. At right we show the main classes of our implemented code and at left we show how instances of these classes interact to each other during the simulation time. The classes represented with a darker border execute co-routines of the libcppsim library. Thus, these classes advance the simulation time by executing operations like hold() and passivate().

The "Simulator" class creates the servers (computers), the communication network, activate the main process of the simulation and starts the simulation. When the simulation is finished, it collects and prints the statistics (for example, number of CRUD operations and times for each one in nanoseconds). The "Server" class represents a computer with a limited number of resources like a RAM memory and a secondary memory (HHD). Each server is composed by "Processors" with "Cores". The processor class en-queue incoming tasks and schedules those tasks to the cores when they are available. The cores simulate the database access costs and the processing costs by executing a hold() operation.

The "client", "mongos" and "mongod" classes inherit from a "Program" class. Every Δ_t unit of time an instance of the client class sends requirements to the simulated MongoDB database (an instance of the mongod class) through the mongos. Namely, the client class executes a hold(Δ_t) operation belonging to the libpccsim library. The requirements include operations like insert, update, select or delete. The mongos class coordinates the execution of the operations requested by the clients and distributes the data when the database is replicated and/or partitioned. The "mongoD" class manages the requirements of the clients and accesses to the MongoDB database. The "Event_time" class is used to obtain the simulation costs of the database operations.

The class "Message" is used to represent the data communication among different programs. There are two types of messages: "Request" and "Response". A "Request" is a message originated in the client. A "Response" is a message sent from the MongoDB to the client. The "NetPIPE" class represents a pipe network. It consists of channels through which messages are sent from one program to another. The "Channel" class is used to simulate the cost of sending messages through the network which considers the velocity of the switch and the size of the messages.

4 RESULTS

4.1 Experimental Settings

Experiments were executed on a cluster with five computers i7-6700 CPU 3.40GHz, with 32 GB of RAM. Communication was performed through a switch Cisco Catalyst 2960-S. We used MongoDB 3.4 and a client application implemented with the drive MGO for Golang. We run benchmark programs for all the database

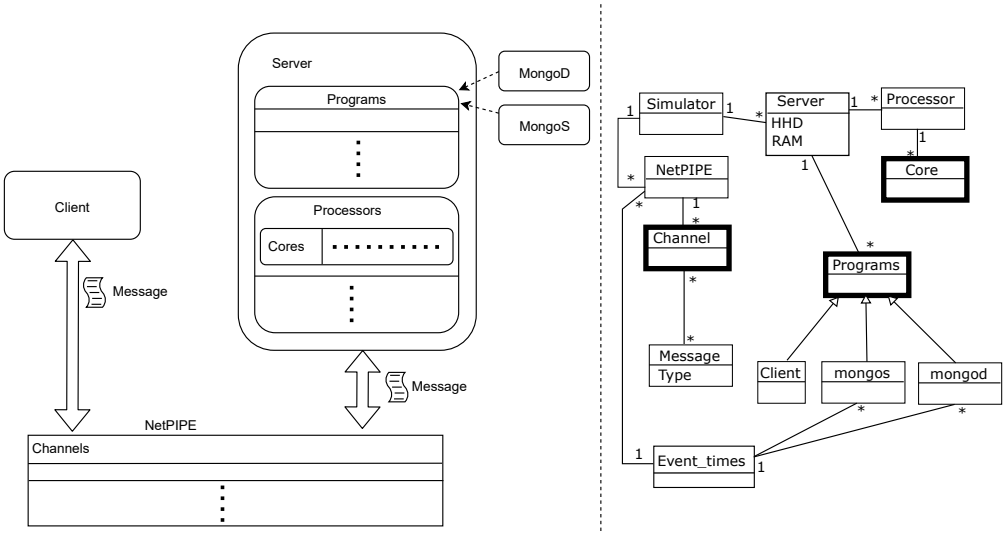


Figure 3: General scheme of our simulator. At right: the main class of our simulator. At left: message communication among the components of the simulator.

operations with data size ranging from 100 bytes to 16 Mb. The results obtained with the benchmark programs were used to feed the "Event_time" class of our simulator which is used at simulation time to estimate the cost of each database operation. The simulator reads the hardware characteristics from a JSON file.

4.2 MongoDB Simulator Accuracy

To validate our proposed TCPN-based simulation tool, we run a set of benchmark programs. These benchmarks are also used to properly tune the simulator cost parameters, which is critical to achieve a comparative evaluation. We implemented and executed benchmark programs for all operations supported by MongoDB with different size of data. Time was measured in nanoseconds using the $Now()$ and $Since(t\ Time).Nanoseconds()$ functions belonging to the time package of Golang.

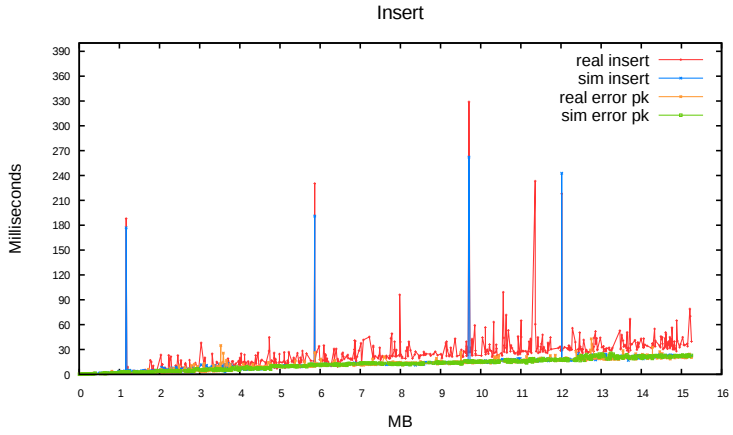
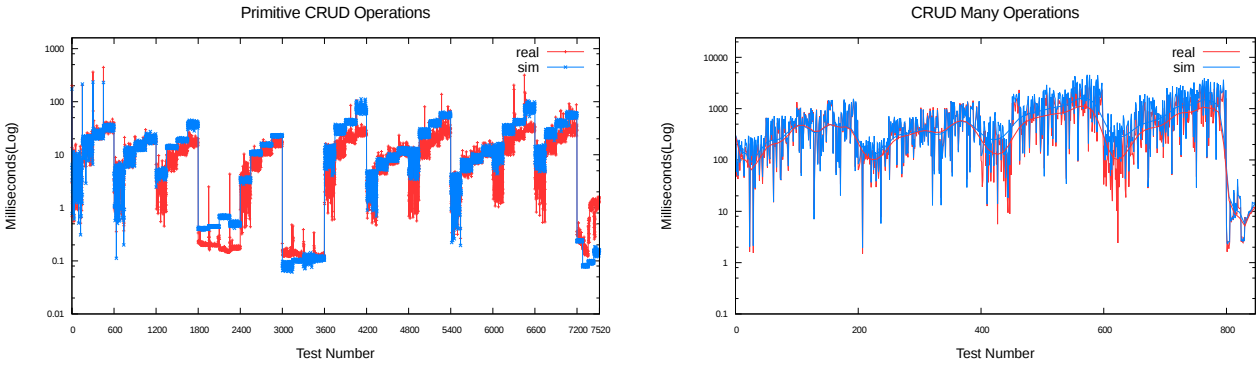


Figure 4: Results in milliseconds reported by a real execution of MongoDB and our simulator for the insert operation.

In Figure 4 we show the execution time in milliseconds reported by a real execution of the MongoDB (real) and our proposed simulator (sim). In this experiment, we executed insert operations with different size of data ranging from 100 bytes to 16 Mb. The insertions were performed on four different collections. In the first collection we inserted data with sizes between 100 bytes and 4 Mb. In the second collection we inserted data with sizes between 4Mb and 8Mb, in the third collection data sizes between 8Mb and 12Mb and in the last collection data sizes ranging from 12Mb to 16Mb. There are peaks in Figure 4 representing insert operations when the collection does not exist - note that the x-axis is not ordered chronologically but by data size. We also show results when executing insert operations with a primary key error. That is, we insert a document with a primary key that already exists and therefore the document is not inserted in the collection. Both insert operations report execution times tending to increase as the data size increases.

Figure 4 shows that the simulation results achieve good agreement with the results from the actual implementation of MongoDB for insert operations. To verify these results, we compute the mean square error of the deviation or mean square difference defined as $\varepsilon_m = \sqrt{(\sum(x_i - \bar{x})^2/(n(n-1))}$. It is the measure of the differences between values obtained by the simulation and the values reported by the real execution. In this experiment we computed $\varepsilon_m = 10,2$. Additionally, we obtained a Pearson correlation of 86%.



(a) (b)

Figure 5: Execution times reported by a real implementation of MongoDB and our proposed simulator for a trace with: (a) 7520 CRUD operations with one document. (b) 2448 CRUD operations with many documents.

Figure 5 shows the results reported by a real execution of the MongoDB and our simulator for a trace with different CRUD operations and with different data sizes (from 100 bytes to 16MB). Figure 5.(a) show results for CRUD operations with a single document. Figure 5. (b) show results reported by CRUD operations with many documents. Also, we include the interpolation function of the points (spline) in order to graphically show how the real and simulated curves fluctuate.

In Figure 5.(a), we executed a trace composed of 600 tests for each operation. The first 600 tests are INSERT operation, the next 600 tests (601-1200) are INSERT operations with a primary key error. Tests 1201-1800 correspond to FIND operations. Tests 1801-2400 correspond to NOT FIND operations (when searching for an item that does not exist in the collection). Tests 2401-3000 correspond to FIND ONE ID operations (like the FIND operation but searching an indexed item). Tests 3001-3600 correspond to NOT FIND ID operations. Again, it is similar to the NOT FIND operation, but it uses an indexed item. Tests 3601-4200 correspond to UPDATE ONE operations, which are used to update only one item. If the collection has more than one item matching the query, only the first one is taken into account. Tests 4201-4800 correspond to NOT UPDATE operations (when trying to update an item that does not exist in the collection). Tests 4801-5400 correspond to UPDATE ONE ID operations. It is like the UPDATE ONE operation but using and indexed item. Tests 5401-6000 correspond to NOT UPDATE ID operations, which are similar to the NOT

UPDATE operation but using an indexed item. Tests 6001-6600 correspond to UPSERT NO ID operations. These operations search for a one, and only one document to update. If the document does not exist, it creates the document. Tests 6601-7200 correspond to UPSERT ID operations which are similar to the UPSERT NO ID operation but using an indexed item. Tests 7201-7280 correspond to NOT DELETE operations (Tries to delete a nonexistent document). Tests 7281-7360 correspond to NOT DELETE ID operations, which are like the NOT DELETE operation but using an indexed item. Tests 7361-7440 correspond to DELETE ONE ID operations, used to delete a document with an indexed item. Tests 7441-7520 correspond to DELETE ONE operation. They are similar to the DELETE ONE ID operation but without an indexed item.

In Figure 5.(b) we execute a trace with a subset of 200 operations involving the management of many documents (e.g. the UPDATE MANY operation is used to update many documents matching the query). In both cases, operations with one document or many documents, Figure 5 shows that our simulator is capable of reproducing the behavior of a real implementation of MongoDB. For operations with one document, we obtained a relative error $\varepsilon_m = 13.7$ and a Pearson correlation of 78%. For operations with many documents, we obtained a higher Pearson correlation of 97.3%, because instead of maintaining a linear function according to a number of retrieved documents, we used a logarithmic factor which was obtained with linear regression. Therefore, both the real and the simulated results present similar tendency.

4.3 Performance Evaluation

In the following experiments we evaluate the performance reported by our MongoDB Simulator Tool. We evaluate the running time, utilization levels and the memory usage in bytes with different number of processors P. We simulate a network with transfer velocity of 100 Mbit per second. The aim of this section is to show how our simulator tool can be used to evaluate the scalability of MongoDB deployed on a cluster of computers.

In figure 6 we show how our simulator tool can be used to evaluate the throughput. The x-axis show the simulation time advance. We executed a total of 10,000 CRUD operations with different document sizes on a single processor. The inter arrival time is set to $\Delta_t = 1$, so the database is saturated reporting a level of utilization close to 99%. As expected, operations executed on small documents finish first. Operations executed on documents of larger sizes require more time to finish. We also include the interpolation function of the points (spline) for each range of document sizes.

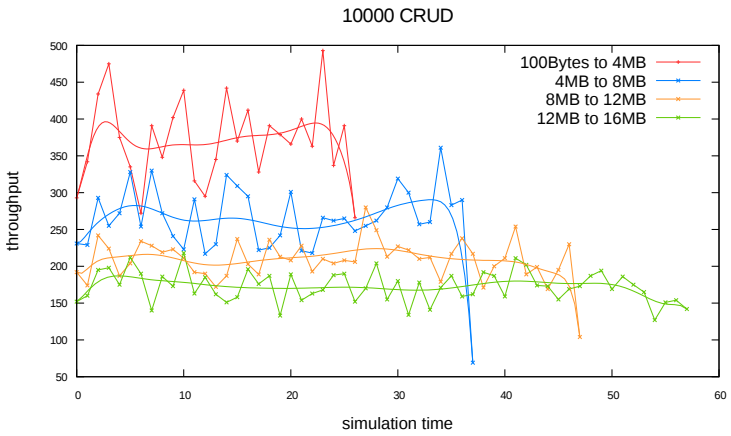


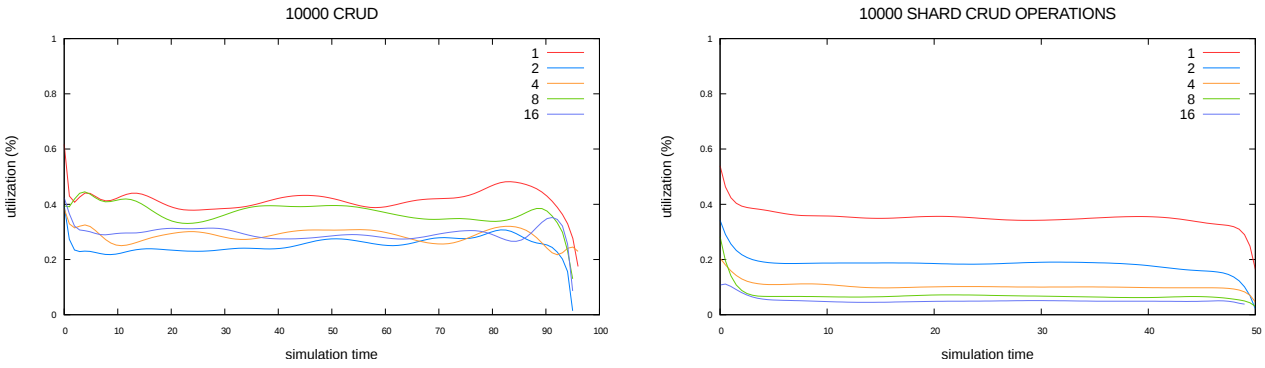
Figure 6: Throughput obtained with CRUD operations executed with documents with different sizes on a single processing node.

# Nodes	Broadcast CRUD		SHARD CRUD	
	Throughput	Avg. Resp. Time	Throughput	Avg. Resp. Time
1	20.62	0.0485	39.22	0.0255
2	35.79	0.0279	46.34	0.0216
3	64.64	0.0155	60.90	0.0164
4	124.62	0.0080	93.10	0.0107
16	228.13	0.0044	155.50	0.0064

Table 1: Throughput and average response time for CURD operation when they are sent to many MongoDB instances (broadcast) and when they are sent to a single MongoDB instance (SHARD).

Figure 7 shows the utilization level obtained with different number of processing nodes. The x-axis show the simulation time advance. We set the inter-arrival time Δ_t so that the utilization level with 1 processing node is 40% in average. We keep fixed the inter arrival time as we increase the number of processing nodes. In Figure 7.(a) we show results obtained for 10,000 CRUD operations with broadcast. That is, each operation is executed in several MongoDB instances located in different processing nodes. In this case, results show that more resources help to reduce utilization levels up to 20%. However due to each operation potentially hits many processing nodes -if the operation has only the sharded field the operation is directed to a single node, otherwise it is broadcasted-, there is no direct relationship between the number of resources and the utilization. In this experiment, we show that with 8 processing nodes the utilization is similar to the reported with one processor. Also, in Table 1 we show the throughput and the average response time for the 10,000 simulated operations. Although broadcast operations are more unpredictable because their performance depends on how the data is distributed in the processors, the results show that throughput operations tend to increase by 90% when using $P = 16$. While operations that visit a single processor report an increase in throughput of 74%. The average response time presents the same tendency. That is, with more processors the broadcast operations present lower response times.

In Figure 7.(b) we show utilization levels reported with different number of processors for 10,000 CRUD operations without broadcast. That is, operations are solved by a single MongoDB instance. In this experiment we show, as expected, that the utilization level decreases with more processing nodes. Also, Table 1 shows that the throughput tends to increase with more processing nodes and the average response time of the total number of operations tends to decrease.



(a) (b)

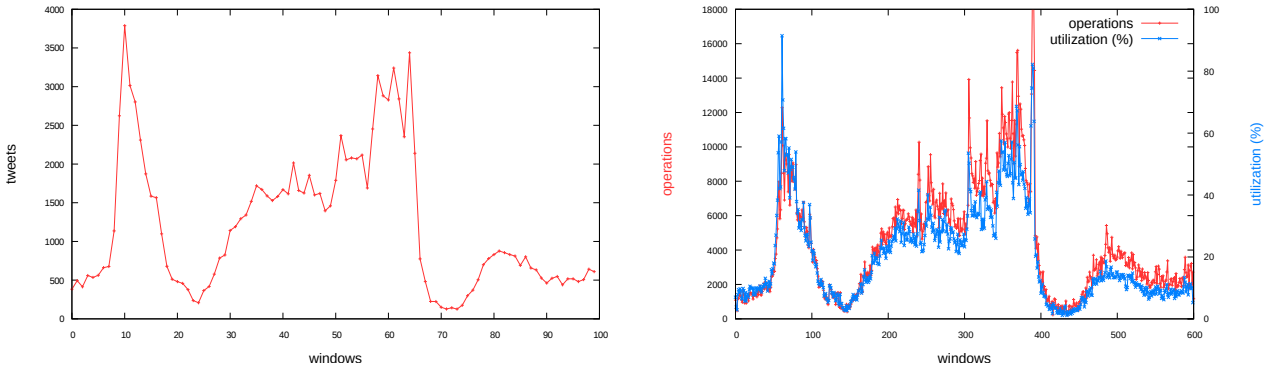
Figure 7: Utilization reported by different number nodes to process 10.000 CRUD operations: (a) Operations sent to many MongoDB instances. (b) Operations sent to a single MongoDB instance.

In Table 2 we show how the network capacity affects the utilization of the MongoDB running on different nodes. With a single communication channel (1 CH) the utilization of the nodes is lower than 20% and the

#CH	MDB_1	MDB_2	MDB_3	MDB_4	CH_1	CH_2	CH_3	CH_4	CH_5	CH_6
1 CH	0.165	0.163	0.160	0.165	0.988					
6 CH	0.373	0.375	0.376	0.362	0.381	0.4	0.384	0.385	0.365	0.386

Table 2: Utilization reported with four processing nodes (MDB) when using a single network channel (CH) or using 6 communication channels.

network becomes a bottleneck. In this case we obtain a throughput of 109 requirements finished per unit of time. As we increase the number of communication channels up to 6, the workload is balances among all resources which report an utilization close to 40%. As expected, the throughput in this case increase to 257 requirements per unit of time.



(a) (b)

Figure 8: (a) Tweets collected for two days with Spanish words related to the #Teleton_2018. (b) Performance of MongoDB.

We also evaluate the performance of MongoDB with dynamic incoming traffic. To this end, we took a sample of tweets with Spanish words related to the #Teleton_2018 which is a TV show (https://www.teleton.org/) that takes place once a year in Chile to raise funds for the neediest people. Figure 8.(a) shows the number of tweets during two days and a half. The x-axis shows the time advance in windows of 30 minutes. This figure shows peaks of tweets during the day and during the night tweets tends to decrease. Also, after the TV show is finished, tweets tend to decrease until they disappear.

We used this trace to simulate insert and find many operations into MongoDB. The inserts represent the tweets submitted by users and the find many operations are used by TV show organizers to obtain statistics. We simulated the insertion of 2,167,974 Tweets and 31,592 find Many operations. Figure 8.(b) we show the number of operations simulated (red line) and the utilization level (blue line) reached by the MongoDB. The $x-$axis shows the time advance every 5 minutes. We show that our simulator can reproduce the behavior of real users. Also, the utilization level is guided by the incoming traffic.

Finally, in Figure 9, we evaluate the MongoDB performance upon the failure of processing nodes. The y-axis shows the utilization level and the x-axis shows the simulation time advance. We set the incoming traffic like Figure 8. We use five partitions (shards) and three replicas per partition. Processing nodes fail when $x = 60$, $x = 192$ and $x = 384$ to reach a total of 20%, 40% and 60% of total failed nodes. This experiment shows how failures impact on the utilization of the system.

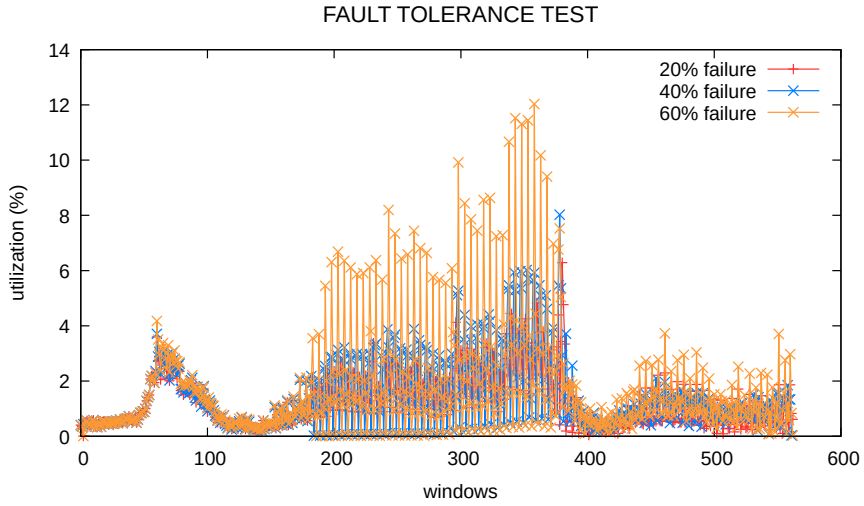


Figure 9: Utilization levels reported by the simulated MongoDB when different percentage of processing nodes fails.

5 CONCLUSION

In this paper we presented a simulation tool for the well-known mongoDB database. The simulator supports many configurations and operations provided by real MongoDB. Our simulator was designed is such a way that each component can be easily replaced or modified without affecting the implementation code of the other components. We validate our simulator against a trace executed on a real implementation of the database. The parameters of the simulator must be adjusted for different architectures by means of benchmark programs which can take hours to complete. Results show that our simulator is capable of reproducing the same behavior and results reported by the real implementation.

This simulator can be used by database managers and engineers in charge of data centers to test different applications before deploying them into a real MongoDB implementation. We showed how our tool can be used to evaluate different metrics such as throughput and utilization levels of the processing nodes. We showed that for operations requiring broadcasts the results not always follow the expected tendency (i.e. decrease utilization levels) as we increase the number of nodes. We evaluated the effect of the network capacity on the system performance. We also sowed that our simulator (1) can be fed with real traces to emulate user behaviour and unpredictable incoming traffic; and (2) can be used to find saturation points when failures occurs.

As future work, we plan to evaluate the impact of memory saturation on the MongoDB performance. That is, to evaluate the performance when the size of the dataset is larger than the memory size and when MongoDB accesses disk with unpredictable latencies.

REFERENCES

Abramova, V., and J. Bernardino. 2013. “NoSQL Databases: MongoDB vs Cassandra”. In *International C* Conference on Computer Science and Software Engineering*, C3S2E ’13.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2008, June. “Bigtable: A Distributed Storage System for Structured Data”. *ACM Trans. Comput. Syst.* vol. 26 (2), pp. 4:1–4:26.

Copeland, R. 2013. *MongoDB Applied Design Pattern*. OŔeilly Media.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. "Dynamo: Amazon's Highly Available Key-value Store". *SIGOPS Oper. Syst. Rev.* vol. 41 (6), pp. 205–220.

Mailewa Dissanayaka, A., R. R. Shetty, S. Kothari, S. Mengel, L. Gittner, and R. Vadapalli. 2017. "A Review of MongoDB and Singularity Container Security in Regards to HIPAA Regulations". In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, pp. 91–97.

Marzolla, M. 2004. "LibCppSim: A SIMULA-like, portable process-oriented simulation library in C++. In *ESM*.

Nambiar, R., M. Poess, A. Dey, P. Cao, T. Magdon-Ismail, D. Qi Ren, and A. Bond. 2015. "Introducing TPCx-HS: The First Industry Standard for Benchmarking Big Data Systems". In *Performance Characterization and Benchmarking. Traditional to Big Data*, pp. 1–12.

Wei-ping, Z., L. Ming-xin, and C. Huan. 2011. "Using MongoDB to implement textbook management system instead of MySQL". In *International Conference on Communication Software and Networks*, pp. 303–305.

AUTHOR BIOGRAPHIES

GABRIEL OVANDO-LEON is a PhD student in Computer Science at Universidad de Santiago de Chile (USACH). He holds a BSc in Computer Engineering from Universidad de Magallanes (2014). Currently, he is an active member of CITIAPS (Centre for Innovation in Information Technologies for Social Applications). His email address is juan.ovando@usach.cl.

LUIS VEAS-CASTILLO is a PhD student in Computer Science at Universidad de Santiago de Chile (USACH). He holds a MEng (2011) in Informatics from Universidad de Santiago de Chile and a MSc (2018) in Computer Science from Universidad de Santiago de Chile. He is an active member of CITIAPS (Centre for Innovation in Information Technologies for Social Applications). He has participated in research projects in CeBiB (Centre of Biotechnology and Bioengineering). His email address is luis.veasc@usach.cl.

VERONICA GIL-COSTA received her PhD (2009) in Computer Science, from Universidad Nacional de San Luis (UNSL), Argentina. She is a former researcher at Yahoo! Labs Santiago hosted by the University of Chile. She is currently an associate professor at the University of San Luis and researcher at the National Research Council (CONICET) of Argentina. Her email address is gvcosta@unsl.edu.ar.

MAURICIO MARIN is a former researcher at Yahoo! Labs Santiago hosted by the University of Chile, and currently a full professor at University of Santiago, Chile. He holds a PhD in Computer Science from University of Oxford, UK, and a MSc from University of Chile. His email address is mauricio.marin@usach.cl.