Testing Cloud Benchmark Scalability with Cassandra

Veronika Abramova
University of Coimbra
DEI - CISUC
Coimbra, Portugal
veronika@student.dei.uc.pt

Jorge Bernardino
Polytechnic Institute of Coimbra
ISEC - CISUC
Coimbra, Portugal
jorge@isec.pt

Pedro Furtado
University of Coimbra
DEI - CISUC
Coimbra, Portugal
pnf@dei.uc.pt

***Abstract*— NoSQL databases were developed as highly scalable databases that allow easy data distribution over a number of servers. With the increased interest of researchers and companies in non-relational technology, NoSQL databases became widely used and a common belief emerged defending that those engines scale well. This means that the use of more nodes would result in reduced execution time of requests and the system would scale adequately, by adding nodes proportionally to data size and load. However, sometimes, adding nodes may not result in improvement of request-serving time. Therefore, it is useful to investigate how different factors, such as workload, data size and number of simultaneous sessions influence scaling capabilities. We will review the architecture of Cassandra, which is known for being one of the most efficient NoSQL engines, and analyze its scalability, using the Yahoo Cloud Serving Benchmark. The results will allow a better understanding of scalability and scalability limitations in that type of environment.**

Keywords*— *NoSQL, Scalability, YCSB, Cassandra, Benchmarking

I. INTRODUCTION

The technology of NoSQL databases has emerged as a new non-relational database model, and quickly caught attention of research community and enterprises. NoSQL databases are expected to respond all the demands of different web applications and enterprise systems concerning fast and efficient data storage and retrieval. They bring a set of additional features that distinguish them from standard Relational Database Management Systems (RDBMS), such as not requiring a rigid database schema to be defined and have an easy horizontal scalability. Meanwhile, the concept of Big Data has also appeared and it corresponds to the data whose volume, velocity and variability are difficult to process using traditional data management tools and techniques [11]. For example, huge amount of data extracted from the web and stored for future querying.

NoSQL databases are designed to provide shared-nothing horizontal scalability that is represented by distribution of data over different servers [4]. Thus, NoSQL databases were developed to easily scale and to be able to efficiently distribute data. As the number of servers increase, the system is capable of serving requests more efficiently. All the requests are executed in parallel, resulting in higher throughput and, depending on the workload, in lower query execution time.

In addition to scalability, speedup is a concept that is highly connected to horizontal scaling and query distribution. Increasing the number of servers results in further data distribution, while requests are executed in a parallel way. Speedup is represented by the amount of reduction of execution time of the requests, as a consequence of the parallelization of query processing.

From enterprise point of view, high execution time of client's requests may lead to a gain loss. Different performance problems as well as request delays are generated by the usage of badly chosen infrastructure and have a high economic impact [10, 16]. A company with success may expect to be able to serve more clients' requests in less time by scaling data and adding more and more servers. Lower execution time of requests results in higher number of server clients and, consequently, in higher profit. On the other hand, one of the main concerns of companies is to be able to fulfill all the requests. The execution time itself is acceptable if it is always below a certain limit, and the main goal is to be able to deal with the Big Data phenomenon and unexpected database load. Both of those issues are highly important and determine whether scale or not.

There have been published a number of papers that contain NoSQL databases comparison and performance analysis. It is important to notice that most of the tests, presented by those papers, are executed in high performance environment and most of them do not evaluate database scalability. Instead, a large number of papers compare latency and throughput obtained for each database with defined number of nodes in cluster. In contrast, we evaluate scalability and how execution time is affected by database and cluster sizes.

In this paper we study scalability properties of NoSQL databases, specifically using Cassandra, to test data scalability, load and total (data and load) scalability. We first review the architecture of Cassandra and then we investigate how it scales with workload, data, number of simultaneous requests and mix of all those. We analyze, experimentally, different query workload response time and throughput while adding nodes. For the experimental work, we used Yahoo! Cloud Serving Benchmark (YCSB) that provides a common set of workloads for evaluating the performance of different NoSQL databases.

The remainder of this paper is organized as follows: section 2 reviews related work. Section 3 discusses a set of memory and index structures of Cassandra, since those are directly related to the performance of the typical access patterns. Section 4 describes the experimental setup and section 5 discusses the experiments. In that section we analyze the behavior of Cassandra when we change the main factors that allow us to conclude on scalability of the engine. Section

978-1-4799-5069-0/14 $31.00 © 2014 IEEE
DOI 10.1109/SERVICES.2014.81

6 contains experiments conclusions and section 7 concludes the paper.

II. Related work

A literature study about the challenges of scaling whole applications in cloud environments is presented in [12], along with an overview of state of the art efforts towards that goal. As the most well-known and popular papers in benchmarking cloud serving systems, we would like to present the results obtained by B.F. Cooper and Datastax enterprise [1, 2]. Both of these papers evaluate latency and throughput in a parallel cluster environment. Similarly to our study, those authors evaluate scalability of tested databases. Cooper et al. show that there is almost no difference in latency presented by Cassandra while varying the number of nodes, which characterize a good horizontal scalability. However, it is not possible to fully understand how that scalability, defended by authors, has affected execution time or the number of operations per second executed by database. On the other hand, according to the Datastax paper, Cassandra increases the number of operations per second that can be envisioned as performance increase and reduced execution time. Similarly to the previously analyzed paper, Cassandra's latency keeps stable and does not differ in different cluster environments. Performance evaluation of Cassandra, HBase and Riak was also presented by Konstantinou et al. [5]. These authors presented throughput results and scalability discussion as well as hardware usage shown during execution of workloads. Pirzadeh et al. [6] evaluated performance of Cassandra, HBase and Project Voldemort using YCSB. However, authors focused their analysis on execution of scan operations that allowed determining scan characteristics of those NoSQL databases. In our evaluation we also executed scan workload to understand how it is affected by data distribution. Beside scans we focused our evaluation on more basic operations, such as, get (read) and put (update). Okman et al. [15] had a different approach in Cassandra evaluation. Instead of evaluating performance and execution time of requests, authors focused their study on one of the main problems of NoSQL databases: security. Authors compared and studied Cassandra and MongoDB and concluded that these databases have no data encryption and their authentication mechanisms are highly vulnerable. Fukuda et al. propose a method for Cassandra's performance increase by reducing execution time of requests. Authors approach was successful and they were able to reduce execution time of simple operations as well as of range queries [17]. Another performance increasing method was tested by Feng et al. [18]. For reduction of execution times were used new indexes that allow faster execution of requests. Authors defend that Cassandra should have index optimization and the use of CCIndexes provides high performance increase. Both of previously related papers focus their studies on performance increase but, differently from those authors, we evaluate Cassandra's standard performance by using available mechanisms and without creating additional indexes. This approach shows us how Cassandra would behave in a normal company environment, using standard and not modified version of Cassandra. Alternative performance enhancement of Cassandra was proposed by Elif et al. [19].Authors' strategy was combining Cassandra with MapReduce engine that is originally used by Hadoop framework [20]. Performed tests demonstrated that Cassandra may be combined with other mechanisms in order to increase its performance and there is a possibility to reduce hardware usage by combining this database with MapReduce.

An interesting approach to benchmarking was presented by Patil et al [7]. In this paper the authors developed the YCSB++ benchmark as a set of extensions to YCSB to improve performance understanding and debugging of advanced features such as speedup techniques and function shipping filters from client to servers. They present HBase evaluation results as well as hardware impact of benchmarking.

In this paper we evaluate Cassandra's scalability and verify how execution time is affected by database and cluster sizes, which is not done in the referenced works

In the next section we describe Cassandra's architecture used for data storage and retrieval.

III. Cassandra Architecture

In our evaluation we used YCSB for database testing. This benchmark performs a set of CRUD (Create, Read, Update, and Delete) operations based on key. Each database record is uniquely identified by its key that is used to retrieve values of its attributes. YCSB generates a random record key that is used for record retrieval and posterior execution of operations over it. It is important to understand that execution time is directly affected by the used index and internal mechanisms of databases. In this section we will present the indexes and memory mechanisms that are used by Cassandra.

Apache Cassandra is a NoSQL database that is classified as belonging to Column Family type databases. Cassandra was originally developed by Facebook and nowadays is under control of Apache Software [8]. This database was designed to handle large amount of distributed data and requests while providing no single point of failure and automatic data distribution over a cluster.

Execution of client requests is divided into parts that are known as stages. Those stages are part of SEDA – Staged Event-Driven Architecture that was developed to handle request demand [9]. Thus, Cassandra is able to adjust the number of threads that are used during query execution accordingly to the load of the system. That means that, in order to reduce execution time, the engine creates a high number of threads, as much as are allowed by the system. Execution time is also reduced by extensive use of volatile memory for data storage and retrieval. During read operations, before executing any disk I/O operations that are known for being inefficient, Cassandra verifies if requested records are already mapped in memory [14]. Fig. 1 shows Cassandra's cache system and data retrieval process. For faster access to the records, Cassandra uses two mechanisms: Row cache and Key cache. Key cache is enabled by default and it is responsible for mapping primary index with all the keys that exist in the database. Similarly, Row cache simulates behavior of in-memory databases by mapping entire records. This approach enables a possible drawback since it increases hardware and system load that may be unnecessary. Therefore, by default, this option is disabled and it is necessary to change Cassandra configuration in order to use this mechanism.

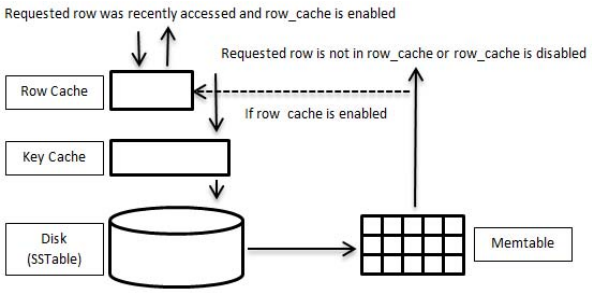


Figure 1 - Cassandra caching

On the other hand, during writes/updates, Cassandra uses a log to register executed operations and map new records in memory for posterior disk flush. This architecture reduces execution and waiting time of operations, since clients don't have to wait for data to become durable. Instead, the log ensures that all modifications will take part and data will be eventually modified on disk. As shown in Fig. 2, received modified data is mapped in memory, using memtable, all executed operations are registered in the log and if new data was inserted, the index is refreshed. Posteriorly modified data is flushed to the disk as an SSTable.

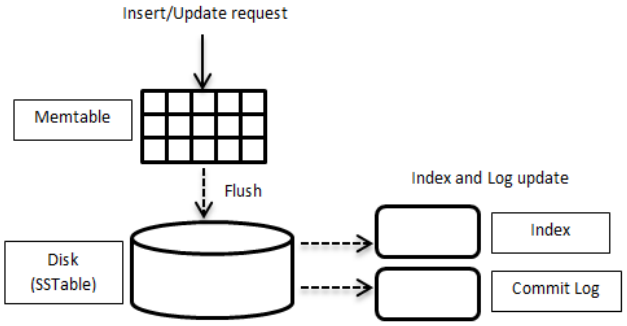


Figure 2 - Inserts/Updates in Cassandra

All stored data is initially accessed using primary index that contains record keys. Each record key is unique and identifies a hash object that stores all record information. The primary index is created automatically over all existing keys and automatically refreshed when new data is loaded. Posteriorly it is used for execution of CRUD (Create, Read, Update, and Delete) requests by accessing records using record key. YCSB will be explained in next section, but in a simplistic way, it executes get and put operations that run over primary index. When the benchmark generates a key for execution of operation, it is used for retrieving record fields. Data distribution directly affects primary index as well as stored records. Each record has a hash value that assigns cluster node that will be responsible for storage and management of that record. Each node has an assigned range of hash values that it is responsible for and each node knows exactly the ranges of other cluster participants [13]. When database receives a new request cluster nodes communicate with each other and the request is redirected to a specific node. This function allows to efficiently handling client demand, reducing execution time, by accessing the correct node instead of executing comprehensive research. Cassandra provides different data distribution mechanisms that determine how hash values are assigned to different nodes. Currently the default data practitioner is Murmur3 that allows even distribution of data among available nodes.

Over the next section we describe experimental setup that contains the description of testing environment, benchmark and executed operations.

IV. Experimental Setup

To perform the evaluation we choose the standard Yahoo! Cloud Serving Benchmark (YCSB). This is one of the most popular benchmarks for testing non-relational databases. It provides data generator and a set of workloads that are defined as a set of CRUD operations. Even though the database is synthetic i.e. data that is generated by the benchmark is a set of random characters, records have a predefined structure with a specified number of fields and are indexed by the key. All generated data can be easily imported into specific databases and workloads are easily executed via terminal. For our evaluation we generated records with 10 fields, each with a total size of 100 bytes plus record key. For data distribution, we used Zipf's law, which assigns more popularity to the records that are the head of distribution while most records will be unpopular (the tail). Since our main goal was to analyze performance of execution of requests, represented by workloads, we executed standard workloads that are available with YCSB: workload A, workload C and workload E. These workloads perform three types of operations, reads, updates and scans, as shown in Table 1.

TABLE I. Executed workloads

Workload	*%Read*	*%Update*	*%Scan*
A	50	50	0
C	100	0	0
E	0	0	100

In order to better understand how scalability affects execution time of requests and if it is possible to achieve requests execution speedup by distributing data, we created three Cassandra scenarios: 1 node, 3 and 6 nodes. This approach enables us to verify the variation of performance according to the number of nodes that are used. Besides the variation of used nodes, we gradually increased the number of used records. Thus, we evaluated loading time presented by Cassandra using 1M, 10M and 100M records (1GB, 10GB and 100GB of data) generated by YCSB on 1, 3 and 6 nodes. Varying data size and the number of threads tested Cassandra scalability during executions. Thus, we evaluated performance while increasing the amount of stored data and request rates. These tests were performed in 1 node and 6 node cluster environments. The characteristics of nodes used are, as follows: Node_1 – Dual Core (3.4 GHz), 2GB RAM and disk with 7200 rpm; Node_2 – Dual Core (3.4 GHz), 2GB RAM and disk with 7200 rpm; Node_3 – Dual Core (3.4 GHz), 2GB RAM and disk with 7200 rpm; Node_4 – Dual Core (3.0 GHz), 2GB RAM and disk with 7200 rpm; Node_5 – Dual Core (3.0 GHz), 2GB RAM and disk with 7200 rpm; Node_6 – Virtual Machine with one Core (3.4 GHz), 2GB RAM and disk with 7200 rpm. It is important to notice that, for cluster

evaluation, were always used the same machines, meaning that during evaluation of Cassandra in 3-node cluster we used Node_1, Node_2 and Node_3 regardless of the number of records used. Single node evaluation was executed on Node_1 and all the machines described above were used for 6 node cluster testing.

Some of the Cassandra configurations are as follows: key_cache: 100MB, row_cache: 0; concurrent_reads: 32, concurrent_writes: 32, rpc_server_type: sync.

We started our evaluation by analyzing Scalability and Speedup shown by Cassandra for execution of each workload. We executed workload A (50% read, 50% update), workload C (100% read) and workload E (100% scan) with 1, 3, and 6 nodes for 1GB, 10GB, and 100GB datasets. All the experiments execute 10000 operations. These results allowed us to understand how execution time is reduced by adding more nodes to the cluster and while increasing the database size. As the second part of our evaluation, we tested Data Scalability. For that we varied the offered load against database in terms of data size, while maintaining the same environment, 1 server setup with 10M records (1, 10M) and 6 node cluster with 60M records (6, 60M). Third part of evaluation was focused on Total Scalability that was evaluated by comparing presented results while varying the number of load (threads) for each workload, and using 1 node setup with 10M records and 6 node cluster with 60M records while executing workloads using x6 and x10 threads. Finally, we evaluated Request caused Speedup for 6node cluster with 60M records and, separately, for 1 server with 10M stored records. This evaluation consists of variation of offered load against the database (request threads) to determine if Cassandra is capable to efficiently manage the increase of the requests.

V. EXPERIMENTAL RESULTS

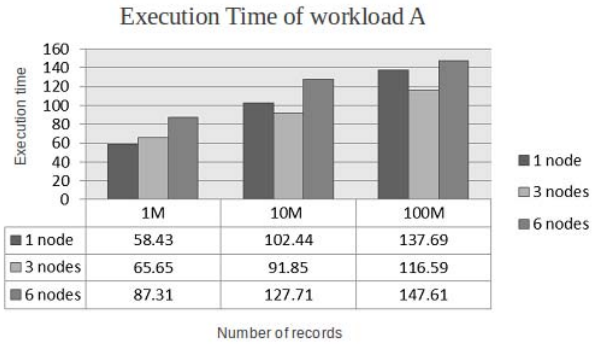
This section describes the results obtained during our experiments. We present our analysis of Cassandra's overall Scalability, Data Scalability and Total Scalability. Also we evaluate how performance is affected by database load. For this evaluation we compare obtained execution time of 10000 operations that is displayed in seconds.

A. Scalability and Speedup

We evaluated Cassandra's Scalability by increasing the number of cluster nodes and, at the same time, database size. We started by executing workload A, workload C and workload E over 1M records and then increased database size up to 10M records and up to 100M records. Cluster size variation started at 1server and posteriorly tests were executed in distributed environment, using 3 node and 6 node clusters.

WORKLOAD A:

Fig. 3 shows the results of execution of workload A in different environments, while varying data and cluster size.



	1M	10M	100M
1 node	58.43	102.44	137.69
3 nodes	65.65	91.85	116.59
6 nodes	87.31	127.71	147.61

Figure 3 - Workload A - Scalability and Speedup

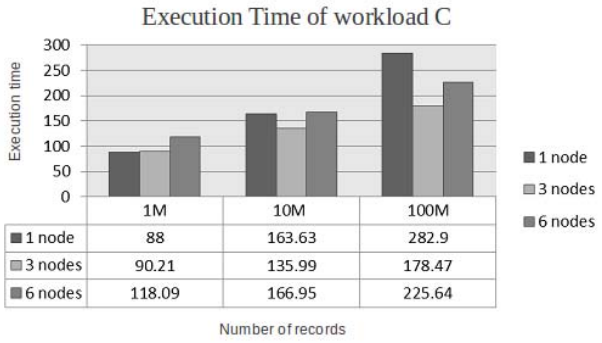
For the results obtained by executing 10000 operations over 1GB dataset we observed that increasing the number of nodes results in increased execution time. This is an example of a situation where increasing the degree of parallelism did not result in improved response times. It can be explained by the fact that the database size is reasonably small and since record access is based on key index, adding more nodes does not result in significant improvement in the index-based access times, while network communication, due to more nodes, creates additional overhead.

On the contrary, when we increased the database size to 10GB and 100GB, we noticed performance improvement when the number of nodes in the cluster increased from 1 to 3. This trend did not continue as we added another 3 nodes. Once again, the reason for this behavior has to do with the tradeoff between access time improvement when data is divided into more nodes, and increased overheads related to managing communication over more nodes. In the case of the 3 nodes the improvement in access time was greater, while the increase in overhead of communication was dominant over the improvement obtained by further dividing the data when 6 nodes were used.

In summary, the amount of parallelism should be tuned depending on the amount of stored data, otherwise the excess overheads can offset potential gains from further dividing the datasets.

Another relevant conclusion is that increasing the database size by x10 does not result in x10 execution time increase, in fact it results in a small increase in response time. This is, as expected, due to the data being accessed using an index, therefore the expected increase in access time should be a logarithmic one.

WORKLOAD C:



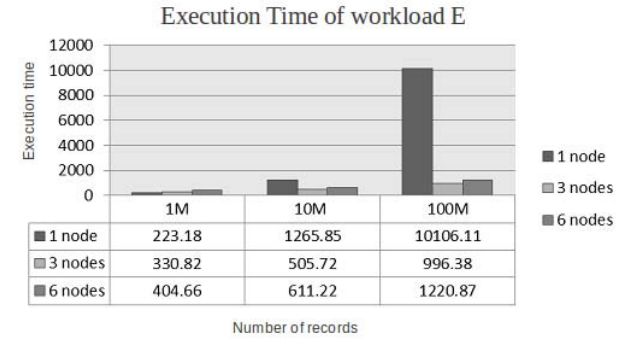
	1M	10M	100M
1 node	88	163.63	282.9
3 nodes	90.21	135.99	178.47
6 nodes	118.09	166.95	225.64

Figure 4 - Workload C - Scalability and Speedup

After observing the results of the execution of workload C, shown in Fig. 4, we noticed somewhat similar patterns to those of the execution of workload A. We detected a steady growth of execution time with database size of 1M records as we added nodes. Once again, since the database size is small, as we add more nodes the additional use of network generates more latency and negatively affects execution time, while the improvement from further dividing the data is smaller.

Overall, in comparison with the execution time of workload A, performance is worse. This workload is 100% read and Cassandra is not optimized for performing reads in comparison with writes. With the higher number of reads, executed by accessing the disk, the performance has decreased.

WORKLOAD E:



	1M	10M	100M
1 node	223.18	1265.85	10106.11
3 nodes	330.82	505.72	996.38
6 nodes	404.66	611.22	1220.87

Figure 5 - Workload E - Scalability and Speedup

Fig. 5 shows the results for Workload E that consists of scans. Although in the case of workload E we noticed the same pattern and concluded that there is no advantage in increasing the number of nodes with the small dataset, and that for larger datasets it is advantageous to increase the number of nodes but only up to a certain point, we also noticed that the execution of the workload E over 100GB of data and in one node scenario was very slow. We observed that in that environment Cassandra had an average execution of one scan operation per 10 seconds. This high execution time was not observed during previous tests. Scan operations take longer to execute compared to the execution of reads and updates. Unlike previous workloads, each scan is performed over intervals (from 1 up to 100) records, which means that, instead of 10000 get / put operations, 10000 scans are performed for this workload, each scan being represented by reading from 1 to 100 records. The higher execution time over 100M on 1 node may be due to the fact that the database is large and the operations are performed one by one in the single node case.

B. Data and Load Scalability

In this experiment we evaluate data and load scalability. Data scalability evaluates the effect of adding nodes to handle more data, while load scalability evaluates the effect of adding nodes to handle more load. Data and load scalability evaluate the effect of adding nodes to handle both factors simultaneously.

WORKLOADS A and C:

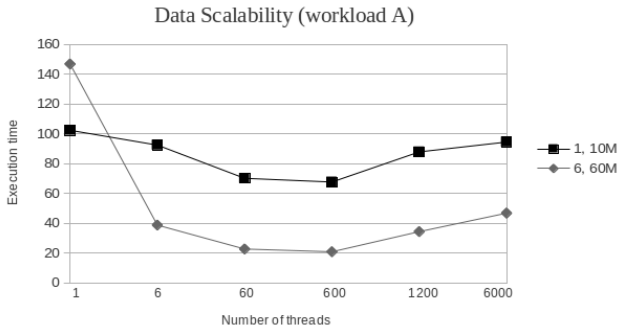


Figure 6 - Workload A - Data Scalability

Fig. 6 allows us to reach interesting conclusions. First of all, by looking at what happens with a single thread, we can conclude that the system exhibited sub-linear scalability. The results show that, in that case (1 thread), when the data set size was increased 6 times, increasing the number of servers 6 times resulted in 1.5 times higher execution time. This also happened for workload C shown in Figure 7, although in that case the degradation in execution time was not as large.

On the other hand, as more request threads are added, results show that super-linear scalability happens. For instance, with 6 threads, the execution time decreased by almost 1.5 times when we added 6 nodes to offset a 6 times increase in data size. This advantage remains as the number of threads is increased. The system is taking advantage of the opportunity to handle more threads efficiently when it has more nodes. Even with a single node, Cassandra is able to improve response times slightly when more independent threads exist.

Although Cassandra proved very effective at using parallelism, at some point we would expect that adding more threads would result in worsening performance, due to the additional overheads incurred by managing more threads and limits on the amount of parallelism that exist. We observed this as we increased the number of threads from 600 to 1200, as expected.

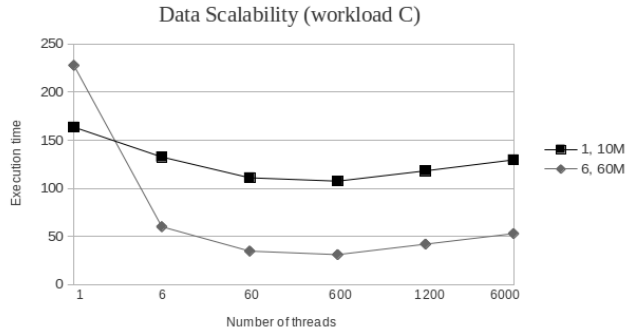


Figure 7 - Workload C - Data Scalability

Execution results of workload C are shown in Fig. 7. We observed similar behavior in comparison with execution of workload A. Even though performance breakpoint is the same (600 threads), overall execution time has increased. This is due to workload C being 100% read with no update operations.

WORKLOAD E:

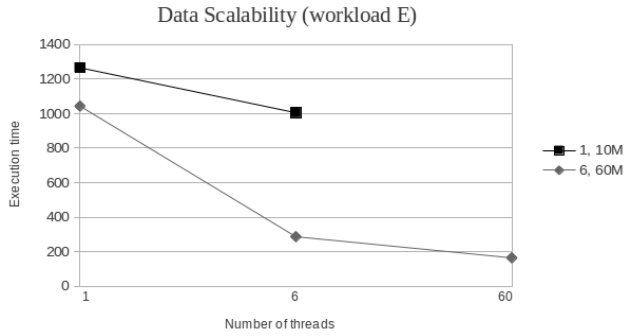


Figure 8 - Workload E - Data Scalability

As Fig. 8 shows, and contrary to the previous cases, workload E exhibited super-linear scalability even for 1 thread. Parallel execution of workload E was advantageous. This is explained by the fact that this workload executes scans and having many nodes scanning in parallel increases the throughput and, therefore, reduces the time to execute the whole scan. For this workload we were unable to run the remaining tests using the default configurations of the server. The system would timeout while waiting for long response from database. Since workload E consists of scan operations, and differently from previous tests (workload A and C), Cassandra had to wait for a long time for query answers as we added threads, therefore the system would timeout.

C. Total Scalability

Total Scalability was evaluated by varying the load (number of threads) and the data size simultaneously, while also adding nodes to offset the increased data and load sizes.

Fig. 9, 10, and 11 show how Cassandra scales when adding more processing threads and more stored data for 3 different workloads. Workload A is 50% read and 50% update, workload C consists of 100% reads and workload E performs scans.

In the first two columns we show the execution time (in seconds) for execution in one node with 10M records, each workload using one thread, vs. 6 threads with 10M records. In the second group of bars we increased the number of records and processing units (number of nodes) x6 and x10 the number of records and threads in the second bar (light gray). We conclude that Cassandra is super scalable, being up to 4 times faster in parallel environment with 10 times higher demand. Cassandra allows reducing execution time while adding more processing resources to handle more data and request load.

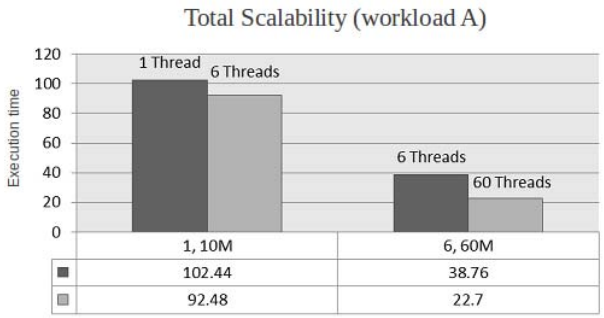


Figure 9 - Workload A - Total Scalability

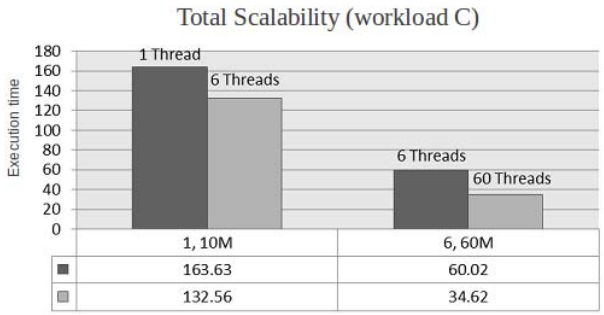


Figure 10 - Workload C - Total Scalability

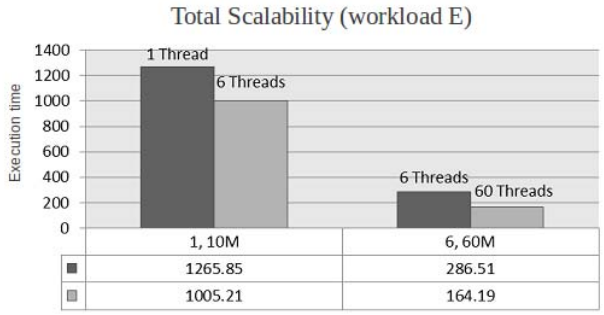
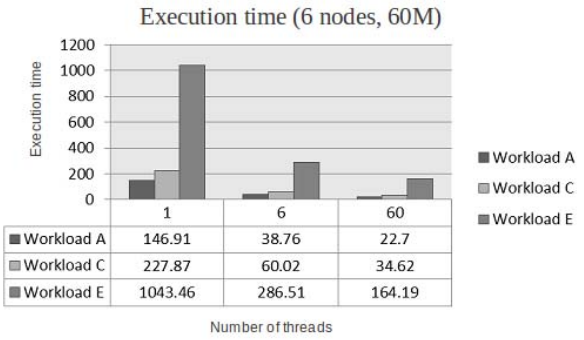


Figure 11 - Workload E - Total Scalability

D. Request caused speedup/slowdown – 6,60M

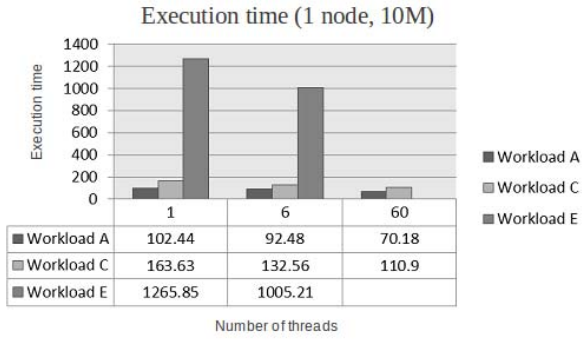
The evaluation of speedup obtained by the increase of the number of threads allowed us to verify Casandra's capability of handling bigger demand and how execution time of requests is affected by operation types (workloads).



	1	6	60
Workload A	146.91	38.76	22.7
Workload C	227.87	60.02	34.62
Workload E	1043.46	286.51	164.19

Figure 12 - Execution time of workloads (6, 60M)

We started this evaluation of Cassandra's request scalability by comparing execution times for each workload while varying the number of threads that were used during executions (Figure 12). These tests were performed in 6 node cluster environment and over 60GB of data. The final results show us that Cassandra scales well with the increase of load. With the increase of threads and regardless of the type of workload, the execution time continued decreasing. That also proves that parallel execution of operations in a 6node cluster environment is capable of handling high number of requests.



	1	6	60
Workload A	102.44	92.48	70.18
Workload C	163.63	132.56	110.9
Workload E	1265.85	1005.21	

Figure 13 - Execution time of workloads (1, 10M)

Similarly to the previous results, obtained while executing workloads in 6 node cluster, in single server environment we observed a constant decrease of execution time for each workload as shown in Figure 13. However, it was not possible to execute workload E with 60 threads due to Cassandra's timeouts.

VI. Lessons Learned

One important conclusion from the experiments is that Cassandra improves processing time when it has concurrent requests, and although it's scaling (more nodes to handle more data) is sub-linear when tried with a single thread, it becomes super-linear when there are a reasonable number of concurrent threads submitting requests.

Increasing the number of nodes does not guarantee response time improvement for the typical get and put access patterns. More nodes means less data per node, but "put" is insensitive to data size, and "get" is also relatively insensitive, since it is based on an index. Moreover, more nodes result in a slight increase in communication and housekeeping overheads, the overall result is a lack of improvement in response times. Increasing the number of nodes becomes important for scan operations on large data sets and when there are many concurrent request threads.

VII. Conclusions

In this paper we have analyzed the scalability properties of a cloud serving NoSQL engine for the typical access patterns. We used Cassandra as our texted experiment and YCSB as the benchmark. Our main goal was to consider the typical put, get and scan operations, review the memory and index structures of the engine, and define an experimental setup that allowed us to stress the system. We tested factors such as data size, number of nodes, number of threads and workload characteristics, and analyzed whether desirable speedup and scalability properties were met.

We were able to conclude that scaling the number of nodes does not guarantee performance improvement, even for relatively large datasets, and this is related to the access patterns and mechanisms. But we also concluded that Cassandra deals well with concurrent request threads and scales well with concurrent threads. We were also able to show, by executing workload E (100% scans), that scans benefit better from having more nodes than put/get operations.

Acknowledgment

Our thanks to iCIS – Intelligent Computing in the Internet Services Project for scientific contributions, resources and guidance during our performance evaluation.

References

[1] B. F. Cooper. A. Silberstein. E. Tam. R. Ramakrishnan. and R. Sears. 2010. "Benchmarking cloud serving systems with YCSB". In Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10). ACM, New York, NY, USA, 143-154.

[2] Datastax. 2013. "Benchmarking top NoSQL databases". A performance comparison for Architects and IT Managers. White paper.

[3] A community white paper developed by leading researchers across the United States. "Challenges and Opportunities with Big Data".

[4] R. Cattell. 2011. "Scalable SOL and NoSQL data stores". SIGMOD Rec. 39, 4 (May 2011), 12-27.

[5] I. Konstantinou, E. Angelou, C. Boumpouka, D. Tsoumakos, and N. Koziris. "On the elasticity of nosql databases over cloud management platform"s. In CIKM, pages 2385–2388, 2011.

[6] P. Pirzadeh, J. Tatemura, and H. Hacigumus. "Performance evaluation of range queries in key value stores". In IPDPSW, pages 1092–1101, 2011.

[7] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. L´opez, G. Gibson, A. Fuchs, and B. Rinaldi. "Ycsb++: benchmarking and performance debugging advanced features in scalable table stores". In SoCC, pages 9:1–9:14, 2011.

[8] http://cassandra.apache.org/

[9] M. Welsh, D. Culler, and E. Brewer. 2001. “SEDA: an architecture for well-conditioned, scalable internet services”. In Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP '01). ACM, New York, NY, USA, 230-243.

[10] Datastax. 2013. “Evaluating Apache Cassandra as a Cloud Database”. A performance comparison for Architects and IT Managers. White paper.

[11] D. Talia. “Clouds for Scalable Big Data Analytics”. IEEE Computer (COMPUTER) 46(5):98-101 (2013)

[12] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. “Dynamically scaling applications in the cloud”. SIGCOMM Comput. Commun. Rev., 45-52, 2011.

[13] P. Garefalakis, P. Papadopoulos, I. Manousakis, K. Magoutis. “Strengthening Consistency in the Cassandra Distributed Key-Value Store”. DAIS 2013:193-198

[14] E. Hewitt. “Cassandra - The Definitive Guide: Distributed Data at Web Scale”. Springer 2011

[15] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov. 2011. “Security Issues in NoSQL Databases”. In Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM '11). IEEE Computer Society, Washington, DC, USA, 541-547

[16] L. Beernaert, P. Gomes, M. Matos, R. Vilaça, and R. Oliveira. 2013. “Evaluating Cassandra as a manager of large file sets”. In Proceedings of the 3rd International Workshop on Cloud Data and Platforms (CloudDP '13). ACM, New York, NY, USA, 25-30.

[17] S. Fukuda, R. Kawashima, S.Saito, and H. Matsuo. 2013. “Improving Response Time for Cassandra with Query Scheduling”. In Proceedings of the 2013 First International Symposium on Computing and Networking (CANDAR '13). IEEE Computer Society, Washington, DC, USA, 128-133

[18] C. Feng, Y. Zou and Z. Xu. “CCIndex for Cassandra: A Novel Scheme for Multi-dimensional Range Queries in Cassandra”. SKG 2011:130-136

[19] E. Dede, B. Sendir, P. Kuzlu, J. Hartog and M. Govindaraju. “An Evaluation of Cassandra for Hadoop”. IEEE CLOUD 2013:494-501

[20] J. Dean and S. Ghemawat. 2010. “MapReduce: a flexible data processing tool”. Commun. ACM 53, 1 (January 2010), 72-77.