Independent project (degree project), 15 credits, for the degree of e.g. Bachelor of Computer Science.
Spring Semester 2020.
Faculty of Computer Science.

Performance comparison of different NoSQL structure orientations

Filip Fredlund and Liridon Smailji

Author
Filip Fredlund and Liridon Smailji

Title
Performance comparison of different NoSQL structural orientations

Supervisor
Nazila Hasanzade

Examiner
Dawit Mengistu

Abstract
This study proposes a performance comparison between the different structures of NoSQL databases; document, key-value, column and graph. A second study is also conducted, when looking at performance comparison between three different NoSQL databases, all of the same structure; document based, the databases that are tested here are; MongoDB, OrientDB and Couchbase. Performance tests are conducted using a benchmarking tool YCSB (Yahoo! Cloud Serving Benchmark), and by looking at time to execute and throughput (operations/ second). Beside benchmarking literature reviews are conducted to be able to understand the different NoSQL structures, and to elaborate our benchmarking results. Every NoSQL structure and database in our benchmark is tested in the same way, a loading phase of 1k, 10k and 100k entries, and a running phase with a workload of approximately 50% reads and 50% updates with 1k, 10k and 100k operations. The finding of this study is that there are differences in performance, both between different structures and between same structured NoSQL databases. Document based OrientDB was the highest performing database at high volumes of data, and key-value store database Redis performed best at low volumes of data. Reasons for performance differences are both linked to specific trademarks of the structural orientation, the usage of the specific attributes of CAP theorem, storage type and development language.

Keywords
Database, NoSQL, Performance, Benchmark, Graph, Key-value, Column and Document.

Contents

1. Introduction

This paper conducts a study on performance, on the structural orientations of different NoSQL databases using the benchmarking tool Yahoo! Cloud Service Benchmark. This study will also look at performance differences on different Document based NoSQL databases.

1.1 Background

Databases are used to store information and large amounts of data. Traditionally, applications and other programs use relational databases (RDBMSs). Since the introduction of Big Data, and other new semi, or non structured data types and the fact that we now store, and collect much more information than before, the traditional databases started to show it's limitations. NoSQL databases were developed to fix the limitations of horizontal scalability in relational databases. NoSQL databases are not a replacement for SQL databases, but more of a remedy for where relational databases fall short.

What makes an application or a software great is user-friendliness and rapidness. As a user, you don't want to sit and wait for information being loaded on the screen. Performance of a database has a huge impact on loading time for an application and also the user-friendliness. When a developer chooses what database to run with the application, it is crucial that the chosen database is up for the task, otherwise there is a large risk for the application to be unpopular and unsuccessful, even if the application itself is really good and useful. This is truer now more than even in our information society, where the user simply expects everything to just work instantly, without caring for the developers point of view regarding data storage.

NoSQL stands for 'Not only SQL' and could be described as an umbrella term for a variety of databases, these databases could be categorised according to their structural

orientations and the four main sub-categories are commonly defined as: Key-value stores, Column stores, Document stores and Graph stores [1].

This paper is focused on testing different structural oriented NoSQL databases and comparing their performance. We will also describe the main characteristics of these different NoSQL databases. In this paper we will also test three different NoSQL databases of the same structural orientation and compare their performance and see if there are any special characteristics that differ between them.

1.2 Aim and Purpose

The purpose of this study is to compare the different structural orientations of NoSQL databases and compare their performance in CRUD operations. We want to know this because if we as developers choses the right and fastest database, we can provide the user a better experience when using an application, a very annoying thing is a large waiting time when using an application or a software. By studying the structural orientations we aim for a better understanding of when to use what type of database for best performance. Since there are multiple different structural orientations in NoSQL databases, there has to be both an upside to it, when using it correctly and a downside, when using it wrong. By using a benchmark tool, in this study Yahoo! Cloud Service Benchmark we aim to put the different structural orientations through tests, to see if performance differs between the different orientations. We test the different orientations with the same data and the same operations. Our aim is to draw a conclusion to what structural orientation is the fastest in performance. The second part of this study will dig a bit deeper into one of these structural orientations: document based NoSQL database. The purpose of this part is to see if there are any differences in performance between three of the most popular NoSQL databases: MongoDB, OrientDB and Couchbase, that are all using the document orientation. By studying these databases and also benchmark testing them using CRUD operations our aim is to be able to draw a conclusion on why, and what database has the best performance. By getting the result of performing these tests on the same orientations we aim to acquire knowledge and understanding on why the performance differs.

1.3 Research Question 1

Does structural orientation alter performance in NoSQL databases?

Are there any differences in speed and scalability between the different structural orientations (Key-value, Column store, Document store and Graph Store). To answer this question we will do literary research to fully understand the different structures and perform benchmarking tests with CRUD operations to see a result measured in time and throughput.

1.4 Research Question 2

Do different NoSQL databases with the same structural orientation (Document store) have any differences in performance?

Within the same structural orientation, in this case we have chosen; document store can we find any performance differences. To answer this question we have selected three different databases (MongoDB, Couchbase and OrientDB) on which we will do literary research on the differences and also do benchmark testing, to get actual results in time and throughput on the CRUD operations.

2. Method

To conduct this study we used two different methods, the first is literature review where we looked at other studies in the field of NoSQL and Benchmarking databases. Second method is testing different NoSQL databases with a benchmarking tool.

2.1 Literature Review

This part of the study focuses on reading and analyzing different articles regarding different design aspects of NoSQL, such as CAP theorem. Since each NoSQL structure uses different strategies on how to implement consistency, partitioning and availability of CAP theorem principles does it affect the performance of NoSQL databases. As such this part focuses more in-depth on each structural orientation and how they work and what impact it might have on performance. To perform our literature review we have to search for peer-reviewed articles, published journals and other academic studies on known articles databases like ACM (through HKR website). To acquire the right information related to our study we have used search words: NoSQL, performance, benchmark, comparison and CAP theorem.

2.2 Benchmark Tests

This part of the study focuses on performance testing with Yahoo! Cloud Serving Benchmark (YCSB) [2]. The testing of the study consists of two different research questions, where one is about testing different structures of NoSQL databases, and the other is about testing three different databases of the same structure of NoSQL. The testing will be done by conducting different operations, such as create, read and update and look at speed and scalability with different amounts of data. The testing will be done by measuring the amount of operations with throughput (operations per second) and the time to execute the workload. All tests will be run on the same computer with the following specs:

- CPU: Intel core i7-8700 3,20 GHz, 6 cores, 12 threads
- RAM: 32 GB
- OS: Windows 10
- Disk: Fixed hard drive

2.3 Limitations

Because of this paper being a degree project for school, we have a limited time to conduct our research and testing phases, around 20 weeks. We hosted our databases locally and not in the cloud clustered.

In this paper we will only take into account for measuring speed and performance, the time and throughput (operations/ second) we get when benchmarking the different databases. Because of time limitations we will only perform loading and running phases of 1k, 10k and 100k entries and operations on each database, and we will only use one benchmarking tool YCSB (Yahoo! Cloud Serving Benchmark), we opted to go for this benchmarking tool since it's widely used and well known.

In the running phase we will only run one type of workload which is approximately 50% read and 50% updates. Exactly the same operations will be executed on every database. YCSB performs and evaluates more simple read and write operations on a primary key and does not take the specific structural benefits into account [3].

To conduct the benchmarking tests we have to use some older versions of some of the databases, this since YCSB couldn't run its tests on the newer updates.

3. Literature Review

In this chapter we are going to provide information that we found during our literary review phase about NoSQL databases and it's structural orientations.

3.1 The CAP Theorem

The CAP theorem consists of consistency, availability and partition-tolerance. Consistency meaning how accurate the data will appear, the latest updated version of data or if there was a conflict. Availability meaning that it will be available for response. Partition which in itself stands for when the data is being allocated and written to somewhere across the network of nodes. According to CAP theorem it states that a distributed network can only contain two of these three attributes [4].

Partitioning impacts performance of NoSQL stores over the network. These NoSQL stores use virtual nodes and physical nodes with help of random data placement methods, which in turn can lead to unbalanced partitioning and overall can make the performance worse over time of these NoSQL stores. Using horizontal scaling improves the overall performance of NoSQL stores [5].

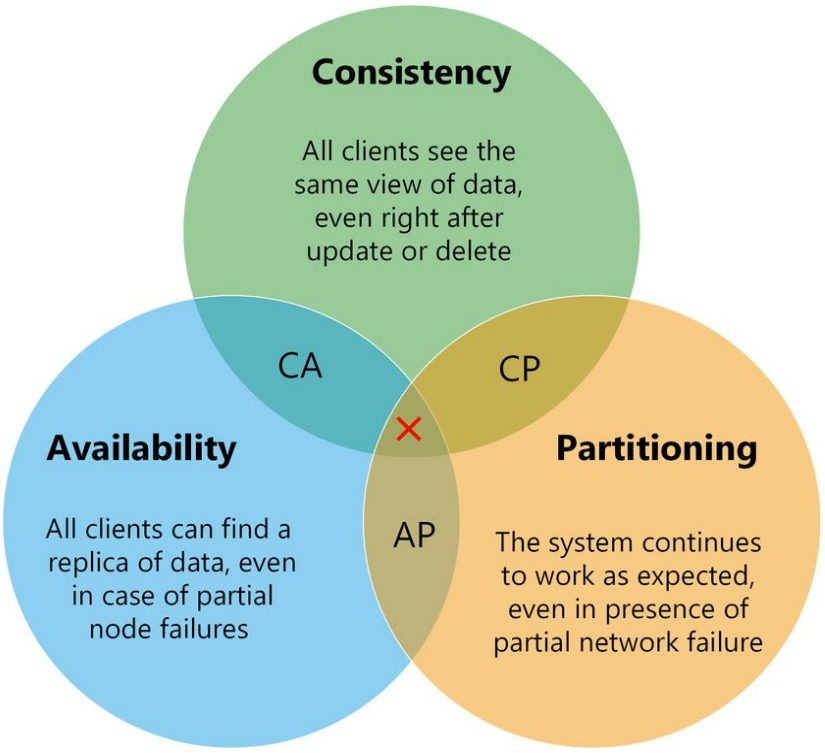

Figure 1: The principles of CAP Theorem for NoSQL illustrated here as an example [6].

When it comes to scaling, it basically means how much the performance can increase when additional space is used, by scaling it up. The opposite can also be said, when decreasing the additional space of a database and how much performance decreases in turn. And this is why NoSQL stores use clusters to store data over many nodes or computers [5]. Storing that data is done horizontally, instead of vertically, meaning the data is stored on several computers instead of one computer. NoSQL uses the horizontal scaling, which stores data across several computers, and can be accessible even if one of these nodes are down, while SQL scaling is done vertically, meaning only on one computer which is also limited to how much it can scale [7].

According to the CAP theorem, it states that a network doing partitioning must choose between either availability or consistency with partitioning [8].

By loosening the consistency to make it weaker, such as eventually, it allows horizontal scaling to perform better. Strong consistency compromise on scalability, while eventual

consistency is better suited for scaling horizontally [8]. Eventual consistency allows these NoSQL databases to achieve a higher throughput and availability with some trade-offs to users and developers if a conflict occurs [9].

In NoSQL databases, the data is stored over several nodes for the purpose of maintaining the ability to provide availability and scalability. Depending on the consistency, the way a transaction happens depends on how these NoSQL databases impose their rules and guarantees, for enforcing on how data is handled on all nodes. In NoSQL there are two types of consistencies, eventual consistency and strong consistency [10].

Strong consistency is when an application is writing to the database and the changes are made visible immediately afterwards the writing is finished. However, when it comes to eventual consistency, the changes are not made visible immediately after writing is finished and is not as accurate as strong consistency but will eventually be as accurate overtime having had all the changes done shown over every node [10].

For decades SQL databases have been depending on a strong consistency model for queries showing immediate visible changes. For developers having worked with SQL there is a necessity for at all times to have immediate changes visible when a query has finished [10].

3.2 Key-Value Store

Key-value stores are among the most popular and simplest NoSQL to use. The data is represented as (key,value) and are stored in efficient and highly scalable key-based lookup structures. A key can be something simple such as a filename, hash or URI. A value represents the data, and what type and size it has such as either string, document or image [7].

Key-value store requires the use of the key, in order to perform a query for accessing the key's value. This type of store is useful for unstructured data as key-value does not enforce a schema for key-value pairs. In some cases, it may be enough to only access the primary key of databases such as key-value and it also depends on the application's use case [10].

Key-value has the ability of retrieving (reading data) or updating data based on a single key or a set of limited keys, however key-value does not provide custom querying for values outside of these keys, where the developer using key-value store for their application is instead encouraged to implement their own custom indexes to maintain and query values outside of these keys [10].

For update queries using key-value the following happens: Firstly, find the record of the desired data to be updated. Secondly, update that desired data and also update the index for it. From the application's side using the database of key-value stores, when an update occurs the entire record may have to be overwritten even if it is just one small attribute changed in the entire record [10].

Consistency in key-value is eventual and there are some trade-offs such as when updating or deleting it is not unusual for conflicts to occur due to the fact that every node containing the data is not updated immediately and as such there is a risk of conflicts to occur. If a conflict occurs the responsibility is either pushed to the user's side, the application or instead uses the most recent value kept and made visible instead [10].

Some differences in how different database providers of key-value have adapted to consistency is to make it tunable, meaning that developers can choose between strong or eventual consistency. Redis is eventual by default [11] and uses CP of CAP theorem [8].

3.3 Column Store

Column stores, which are also known as column-family stores or wide-column stores are inspired by Google Bigtable, where data is represented in tabular form. Tabular form refers to rows and columns of a fixed size, each row has columns inside. Column stores are in fact more advanced than key-value due to the reason that it is also an extended version of key-value store [7].

Data is retrieved with the use of one primary key per column family similar to key-value stores. Column stores use eventual consistency for their databases. Just like key-value, this type of store can also have conflicts when updating values if two users do it at the

same time and can push the responsibility back to the developers or keep the latest version [10].

Cassandra uses AP by default when it comes to the CAP theorem, availability and partition, this means it has eventual consistency by default and also has the option to tune the consistency between strong or eventual. If tuned to use strong consistency it will instead of AP be CP [8].

3.4 Document Store

Document stores is an extended version of key-value stores, in which the value is represented as a document and encoded in the formats of JSON, BSON, XML [7].

In SQL, data is stored in rows and columns whereas the data in a document store are stored in documents as the name suggests. JSON is the most popular format for document store amongst developers. Each field contains a value, it can be anything such as string, array or whatever else data one might need to store. Typically you have foreign keys, tables, records, associations and more in SQL, but in document stores it is all stored together in one single document. The schema for a document store is dynamic in the sense that it can contain different kinds of data [10].

Document databases are general purpose meaning it has a wide use for different kinds of applications thanks to the flexibility that is provided from document stores. This type of database has the ability to make a query on any field within a document and provides a rich set of index capabilities amongst other things depending on which provider. Example with MongoDB, you can update data be it complex manipulations so long as it matches an element in the document, be it an array or nested arrays all within the same transaction of the update query [10].

Document databases have tunable consistency, meaning that it can be configured to either run on strong consistency or eventual consistency. By default document stores such as MongoDB use strong consistency [10].

Both MongoDB and Couchbase Server are CP [11] but tunable, as in giving the option of configuring the consistency between strong or eventual consistency but are by default using strong [12]. OrientDB is CA as default but is tunable in consistency [8].

Between the three document store databases we have opted to benchmark, there are some differences besides CAP theorem. MongoDB and Couchbase are both written in C++ while OrientDB is written in Java. Couchbase and OrientDB use storage types JSON and MongoDB uses BSON [8].

3.5 Graph Store

Graph stores which represents datasets in the form of vertices and edges, where vertices are entities and edges are the relationships between the entities. Graph stores traverse through entities and it needs to be efficient which is why graph-oriented NoSQL emerged. Graph stores are usually coupled with either SQL databases or document stores of NoSQL [7].

Graph stores provide rich query models of both simple and complex relationships which can be used to make either direct or indirect changes to the data. Graph databases are not general purpose, and the main appeal of this one is to make it easy navigating through data between entities and their relationships in an application [10].

Since OrientDB is multi-model with document and graph, it means that it is also using CA when it comes to CAP theorem and is tunable in consistency and by default uses strong consistency [8].

3.6 Related Work

During the literature review phase, there has been a lot of research done on databases both SQL and NoSQL and comparisons in between the different types of databases. Having found a lot of previous work related to NoSQL databases that compares for example MongoDB vs Cassandra and performance NoSQL vs SQL, but no work that compares performance of the structural orientations of NoSQL like in this paper.

There have been papers explaining the differences in NoSQL and some testing done with Yahoo Cloud Serving Benchmark as it is a popular tool for NoSQL databases, which this study also uses [2].

Many tests have been done on databases when it comes to performance and comparisons on both NoSQL and SQL databases. Most commonly is comparing NoSQL with SQL or comparing the two most popular NoSQL databases. MongoDB and Cassandra have been tested against each other multiple times and have often been used in studies as these are popular databases, one is document and the other column, this study also compares these databases amongst others from this. The results of this study is that Cassandra the column based NoSQL database has higher performance in loading phase and MongoDB document based is faster during update commands [13].

While most comparisons look at the number of operations and latency to compare databases this article finds that latency is not important since it does not use a cluster cloud environment, because the tests are hosted locally.

This study will look at speed (time of execution) and throughput (operations done per second) and compare these numbers with database size (scaling the operations) and what happens when scaling up which results in larger amounts of data.

There have been of course comparisons between SQL and NoSQL which this study does not focus on, but most of these studies show relevant information such as the CAP theorem, performance testing with benchmark tools, cloud computing but uses more advanced operations than CRUD which is looked into more detail below.

The article “The Battle between NoSQL Databases and RDBMS” [1], focuses on NoSQL databases such as key-value, column, graph and documents while comparing it to SQL. The article reviews the basics of NoSQL and how the data is stored and does the comparison with advantages / disadvantages for SQL vs NoSQL. It also discusses which usages these databases have, which is comparable to this study that compares different NoSQL databases as well with scaling and throughput with performance in mind. This study does not focus on advantages and disadvantages but more on how different

structures work and if it differs in performance when comparing these together. This study's conclusion is that in our new information world, NoSQL databases permit more reliable deployment and can more easily be suited for organizations requirements.

“On the State of NoSQL Benchmarks” [3], it is about performance due the fact that it uses YCSB to benchmark it is related to this article which also focuses on performance of different nosql databases while theirs focus on more advanced things such as nestled document search rather than just CRUD. It is concluded in this article that further development of YCSB or development of a completely new tool is required to assess the performance between different NoSQL databases since YCSB does not take the specific characteristics into account.

“Consistency Models of NoSQL Databases” [8], is an article which focuses on the implementation of consistency models done by different databases of NoSQL. They look at databases such as Redis, MongoDB, Cassandra, Neo4j and OrientDB and compare their usage of consistency. Their conclusions are as follows: if a database is configured in favor for strong consistency then the database will result in less availability when it comes to partitioning according to the CAP theorem. As for default consistency that each NoSQL structure uses by default, the preferable choice for high consistency and partition-tolerance is MongoDB. As for high availability the preferable choice is Cassandra. For non-distributed databases and no partition tolerance, the preferable choices are OrientDB and Neo4j which offer high consistency. It shares some similarities with this study because it studies almost the same databases, benchmark with YCSB but their focus is on consistency and this study focuses on performance.

The article, “An experimental study on tuning the consistency of NoSQL systems” [9], focuses on finding middle ground on tunable consistency to see if there’s any benefit of how to configure the consistency for which usage the database applies for and depending on the need of the consistency. The conclusion is that whether or not by analyzing Cassandra to see if it can satisfy four session guarantee consistencies by analyzing if and why inconsistency occurs. They propose on how to use the length of write pending

queues as an indicator of the consistency and how to improve the consistency. According to their findings their proposed three methods can help on how to tune the consistency when it comes to the experience of developers/users, by controlling consistency with some trade-offs between consistency, latency or throughput. In this study no tuning is done, benchmarking is done on default value for consistency.

The article “Scalable Database Management in Cloud Computing” [12], focuses on big data with scaling over the cloud with clusters in mind. Their focus is on scaling different NoSQL databases with big data and research design decisions taken by each vendor. In the conclusion, they review existing storage solutions for Big Data for disparities and can determine which are appropriate for different applications on the cloud. The major scalable databases such as Bigtable, Dynamo and PNUTS achieve scalability by using single key-value pairs at one end and single record-level on the other end. There are applications such as online gaming that require multiple keys and record access instead of single key-value. There is a gap between traditional databases and present cloud databases. The gap needs to be fixed if the cloud is to be able to handle all kinds of applications. This study focuses on performance testing different NoSQL with different volumes of data.

4. Benchmark Tests

This chapter will provide documentation and detailed guidance on how the benchmark testing on performance with scalability was done using YCSB and local databases.

4.1 Setup

The setup consists of using Windows 10 for the benchmark tests. Using a released version of YCSB and afterwards installing python. Then adding environment variables for Java, in this case Java SDK Version 8 for being able to run YCSB via python. Python is responsible for launching YCSB through the terminal, such as Windows Command Line and Git Bash. Git Bash which is a UNIX based terminal is similar to Linux's terminal. Afterwards some testing was needed to confirm if YCSB was working, by navigating directly into the folder by using the command: ‘cd’ via the Git Bash terminal and inputting: ‘cd yscb-0.17.0’, after that the command to start YCSB is: ‘./bin/ycsb’ in the Git Bash command line.

4.2 Testing Phase

YCSB has three default command options such as: ‘load’, ‘run’, ‘shell’. Only ‘load’ and ‘run’ is going to be used to make the benchmark tests with the workload A. In order for YCSB to work with a database either hosted locally or through the cloud, before being able to try out the benchmark tool, it is required to install the database(s) and have it running, such as on a local server for hosting the database.

When it comes to using YCSB it is important to have created the database before using YCSB in order for the benchmark tool to recognize the database. There already are workloads that can be used for operations such as insert, read and update, in this case it was decided to use workload A as mentioned before, which consists of ‘load’ and ‘run’ command options, where ‘load’ consists of 100% inserts and run consists of approximately 50% reads and 50% updates. The reason for picking workload A is that using workload A worked well with this study’s research questions regarding

performance and scalability due to the fact that it can benchmark these types of queries to see the overall runtime and throughput of these queries done and how many operations.

After navigating into the YCSB folder it is then possible to use the command ‘./bin/ycsb’ and then deciding if to use ‘load’ or ‘run’ command options when benchmark testing a database, such as in this case: OrientDB graph, Cassandra column, Redis key-value, MongoDB document, OrientDB document, Couchbase document databases.

The next part is adding the binding name used by YCSB for the database in question, such as: ‘orientdb’, ‘cassandra-cql’, ‘redis’, ‘mongodb’, ‘couchbase’, and since OrientDB is a multi-model it was possible to use two different structures of database storage such as document and graph, OrientDB was used twice as a document instance and as a graph instance in these benchmark tests. Another part is also specifying ‘-s’ which is used for printing out the results if it succeeded or failed in error. Then before being able to specify the workload you need to add ‘-P’ also which means specifying the workload file such as adding afterwards ‘workloads/workloada’.

Then the part is to also add ‘-p’ after each variable in order to avoid syntax errors such as adding that part after having added the workload before adding the url connection of the database.

The next part is specifying the connection of the database in order for the operations of insert, read and update to work, by specifying the database name and then adding the localhost and port would look something like this: ‘mongodb.url=mongodb://localhost/ycsb’ and then if needed adding credentials for authentication.

The ‘load’ command option has an option for specifying the amount of inserts, this option is called ‘recordcount’ and example would be if the desired operations is 10 000 then it can be done like this: ‘recordcount=10000’ which would run the workload A with 10 000 inserts and would enable the ‘run’ command option benchmark testing.

The ‘run’ command option consists of approximately 50% each of read and update operations and if the total desired amount of operations for read and update is 10 000 then

it can be done by using something called ‘operationcount’ for ‘run’ command option and the example would look like this: ‘operationcount=10000’ for reading and updating approximately 50% each.

The examples below show how it looks on Git Bash terminal when using YCSB and testing a database for the ‘load’ and ‘run’ phase.

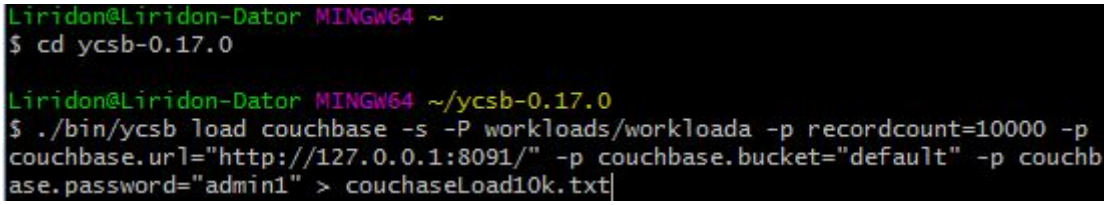

Figure 2: The example shown above is using Couchbase, once the benchmark test finishes in success or failure, it saves the results into a text file.

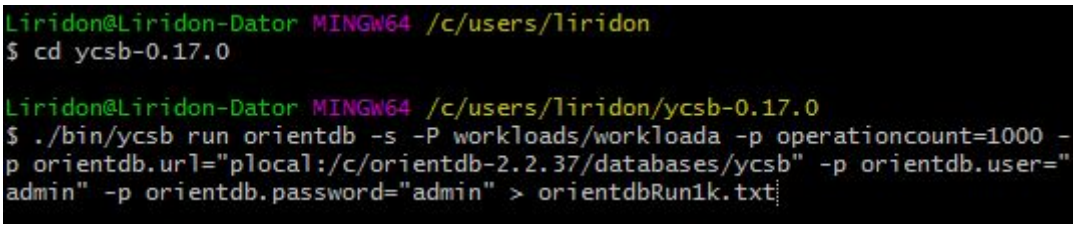

Figure 3: The example shown above is using OrientDB, once the benchmark test finishes in success or failure, it saves the results into a text file.

And this is how the benchmark testing works with YCSB, the hard part is making the database work with YCSB rather than YCSB itself being hard as each database is very different from each other. All benchmark tests were done on the same computer and all databases were hosted locally.

5. Results

This chapter is going to provide the results from the benchmark tests. The following diagrams below show the load and run phases of 1 000, 10 000 and 100 000 operations done for each database. The load phase consists of 100% inserts operations and the run phase consists of approximately 50% each of read and update operations.

5.1 Comparison of Different NoSQL Structures

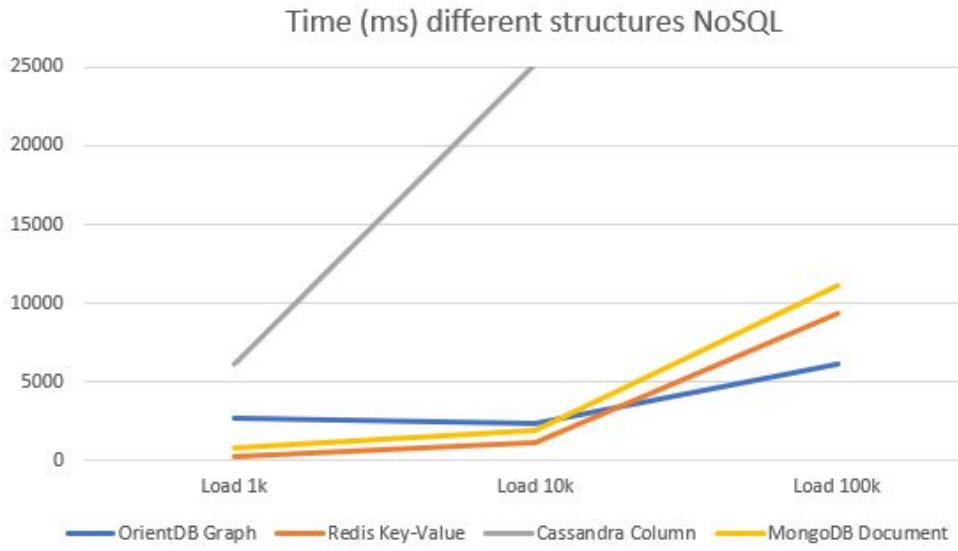

Figure 4: Time (ms) taken for the load phases of 1 000, 10 000, 100 000 operations.

The finish time of load test for load 1k, load 10k, load 100k: Cassandra was the worst of all these load tests. In third place was OrientDB for load 1k and load 10k while OrientDB is in first place at load 100k. Redis performed better overall compared to MongoDB. Both Redis and MongoDB beats OrientDB in load 1k and load 10k but loses in the end to OrientDB as the operations are scaled to load 100k instead, where OrientDB beats both Redis and MongoDB. Cassandra did even show up after load 10k and performed badly compared to Redis, MongoDB, OrientDB. For load 1k the following databases had the finish time in ms: OrientDB (2700 ms), Redis (179 ms), Cassandra (6153 ms), MongoDB

(772 ms). For load 10k the following databases had finish time in ms: OrientDB (2300 ms), Redis (1069 ms), Cassandra (25257 ms), MongoDB (1905 ms). For load 100k the finish time was in ms: OrientDB (6105 ms), Redis (9330 ms), Cassandra (190102 ms), MongoDB (11080 ms).

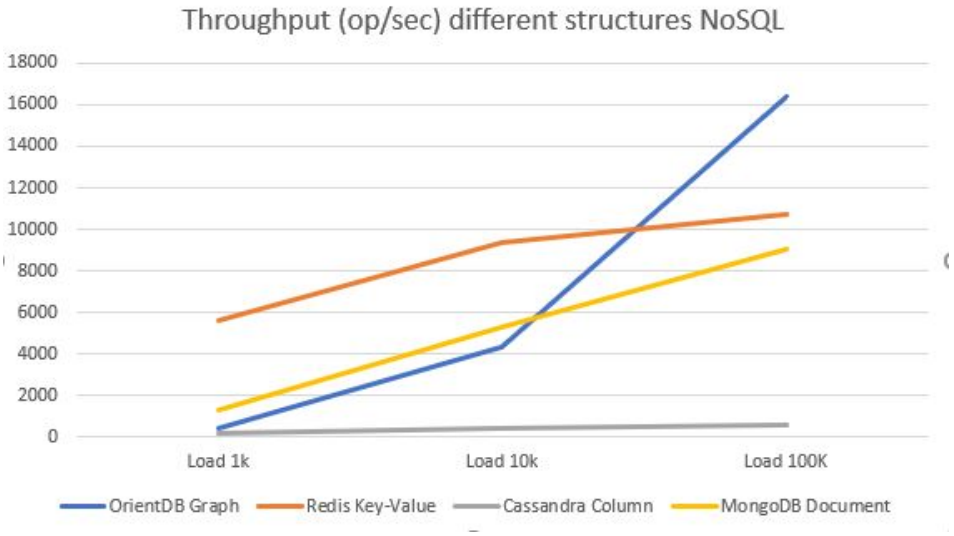

Figure 5: Throughput (operations done per second) for the load phases of 1 000, 10 000, 100 000 operations. The amount of operations done per second.

The throughput on load tests of OrientDB, Redis, Cassandra and MongoDB, this figure above shows the throughput done per second. Beginning with Cassandra we can barely see any at all due the fact that it's below even 1 000 operations for load 1k, load 10k, load 100k and it shows it does not scale very well especially if looking at the other databases and drawing a comparison. One can see that these other databases are indeed scaling well compared to Cassandra. As can be seen again Cassandra, OrientDB and MongoDB loses in throughput to Redis on load 1k and load 10k, followed up by MongoDB in load 1k and load 10k shows that MongoDB is second best while OrientDB is third and Cassandra in last spot. In the load 100k it can however be seen that OrientDB scales the best in throughput as it performs much better in performance than Redis, MongoDB, Cassandra where Redis is second place and MongoDB in third while again Cassandra is in last spot. The throughput for the databases was for load 1k: OrientDB (370.370 op/sec), Redis

(5586.592 op/sec), Cassandra (162.522 op/sec), MongoDB (1295.336 op/sec). For load 10k the throughput was: OrientDB (4347.826 op/sec), Redis (9354.536 op/sec), Cassandra (395.929 op/sec), MongoDB (5249.343 op/sec). For load 100k the throughput was: OrientDB (16380.016 op/sec), Redis (10718.795 op/sec), Cassandra (526.033 op/sec), MongoDB (9025.270 op/sec).

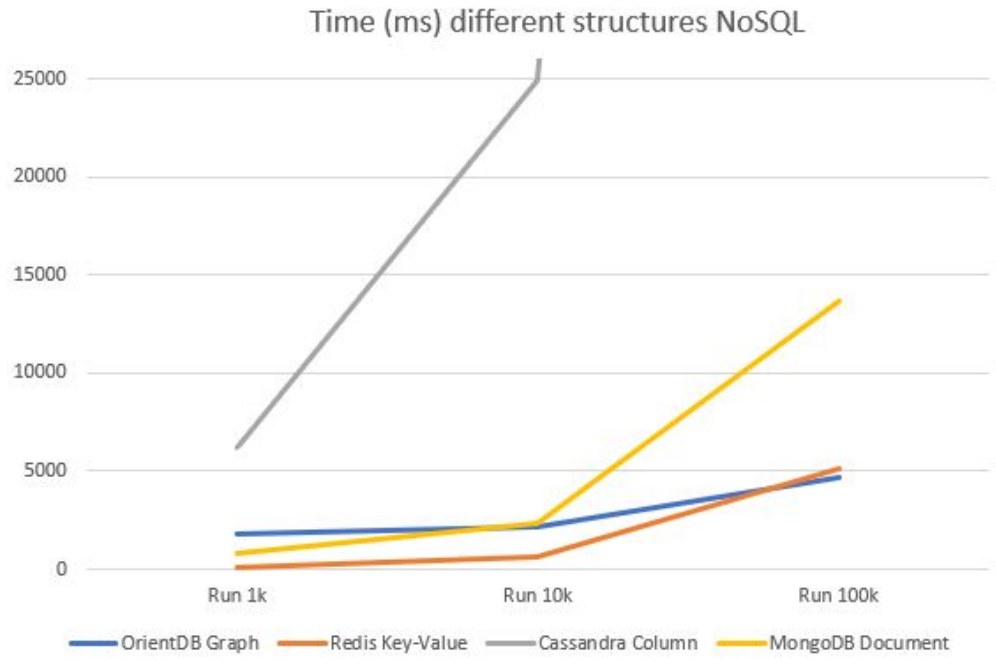

Figure 6: Time (ms) taken for the run phases of 1 000, 10 000, 100 000 operations.

The finish time of run tests for run 1k, run 10k and run 100k: yet again Cassandra does seem to change in terms of finishing last with the worst results of all databases. Cassandra is not visible on the finish time of run 100k. Redis is in first place in terms of finish time for run 1k and run 10k while it is in second place on run 100k where OrientDB is in first place on run 100k but in third place for run 1k and second place for run 10k. MongoDB is in second place for run 1k and third place for run 10k and run 100k. The finish time for each database for run 1k in ms: OrientDB (1794 ms), Redis (133 ms), Cassandra (6251 ms), MongoDB (821 ms). For run 10k the finish time in ms: OrientDB (2178 ms), Redis

(674 ms), Cassandra (24948 ms), MongoDB (2314 ms). For run 100k finish time in ms: OrientDB (4659 ms), Redis (5139 ms), Cassandra (187554 ms), MongoDB (13720 ms).

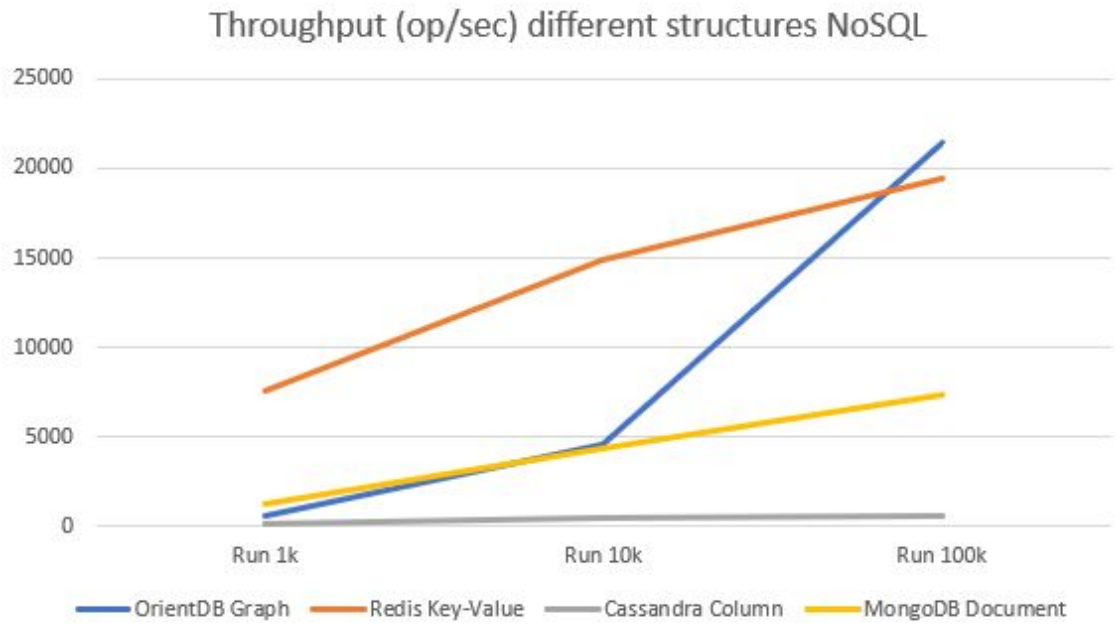

Figure 7: Throughput (operations done per second) for the load phases of 1 000, 10 000, 100 000 operations.

The throughput on run tests of OrientDB, Redis, Cassandra, Mongodb for the figure above shows Cassandra is yet again in last place and has very low throughput below 1 000 operations per second, while Redis is in the first place for both run 1k and run 10k and in second place for Run 100k where OrientDB is in first place on run 100k. MongoDB is in second place at run 1k and third place for run 10k and run 100k. OrientDB is in third place for run 1k and second place for run 10k. The throughput for run 1k for the databases was: OrientDB (557.413 op/sec), Redis (7518.796 op/sec), Cassandra (159.974 op/sec), MongoDB (1218.026 op/sec). For run 10k the throughput for the databases was: OrientDB (4591.368 op/sec), Redis (14836.795 op/sec), Cassandra (400.833 op/sec), MongoDB (4321.5211 op/sec). Lastly for run 100k the throughput for the databases was: OrientDB (21463.833 op/sec), Redis (19459.038 op/sec), Cassandra (533.179 op/sec), MongoDB (7288.629 op/sec).

5.2 Comparison of Document Structure NoSQL Databases

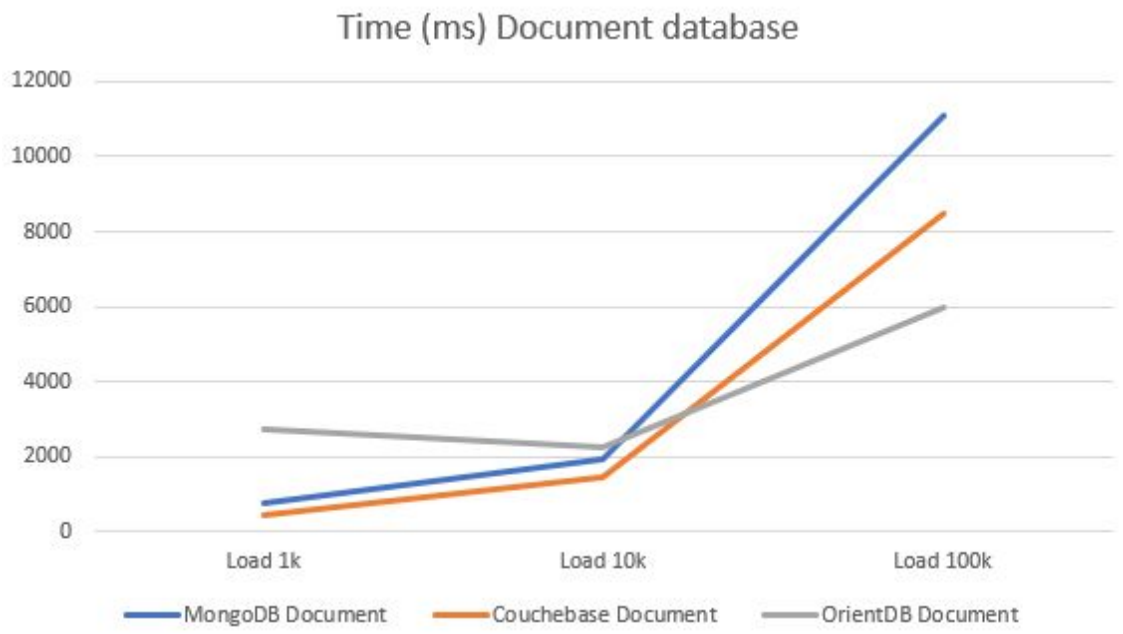

Figure 8: Time (ms) taken for the load phases of 1 000, 10 000, 100 000 for Document Databases.

The finish time of load tests for document databases of MongoDB, Couchbase, OrientDB for load 1k, load 10k, load 100k for which placements these databases got. Couchbase performs best in load 1k and load 10k but only second best in load 100k, while MongoDB performs second best in load 1k and load 10k and is third place in load 100k. OrientDB is in last place for load 1k and load 10k but in first place for load 100k. The exact finish time in ms for these document databases was for load 1k: MongoDB (772 ms), Couchbase (430 ms), OrientDB (2741 ms). For load 10k, the finish time in ms was: MongoDB (1905 ms), Couchbase (1432 ms), OrientDB (2251 ms). For load 100k, the finish time was in ms: MongoDB (11080 ms), Couchbase (8485 ms), OrientDB (5988 ms).

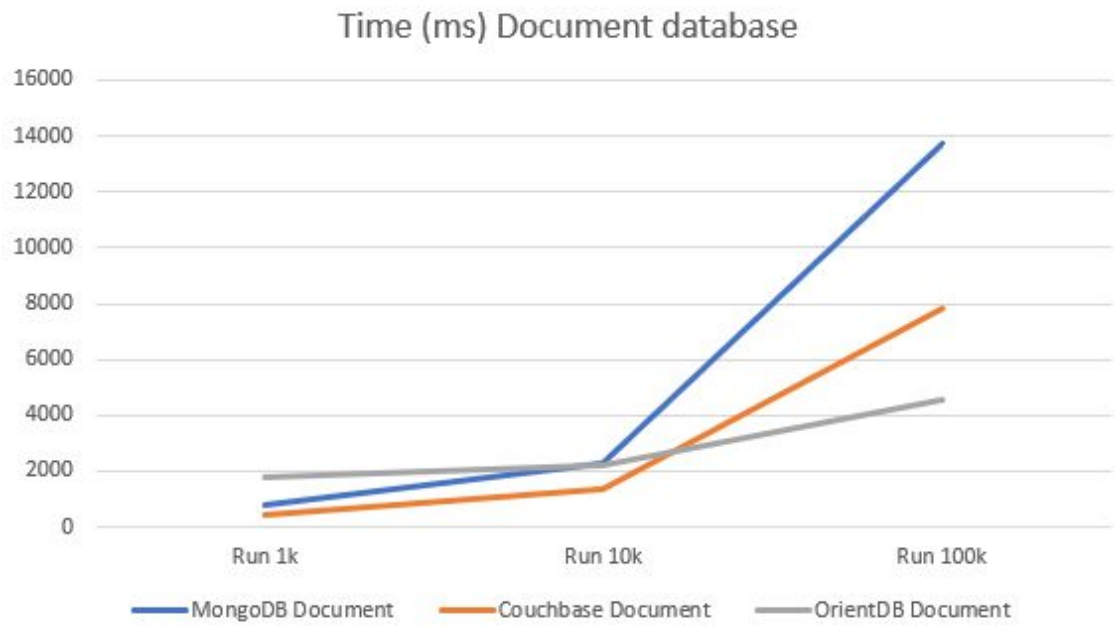

Figure 9: Time (ms) taken for the run phases of 1 000, 10 000, 100 000 for Document Databases.

The finish time of run tests for document databases of MongoDB, Couchbase, OrientDB for run 1k, run 10k, run 100k: for which Couchbase performs best in run 1k and run 10k while Couchbase is second best for run 100k. While MongoDB is second best in run 1k and third place for run 10k and run 100k. OrientDB is last place in run 1k and second place for run 10k and for run 100k, OrientDB is first place. The finish time here is for run 1k in ms: MongoDB (821 ms), Couchbase (418 ms), OrientDB (1780 ms). And for run 10k the finish time in ms was: MongoDB (2314 ms), Couchbase (1370 ms), OrientDB (2195 ms). Lastly for run 100k the finish time in ms was: MongoDB (13720 ms), Couchbase (7838 ms), OrientDB (4588 ms).

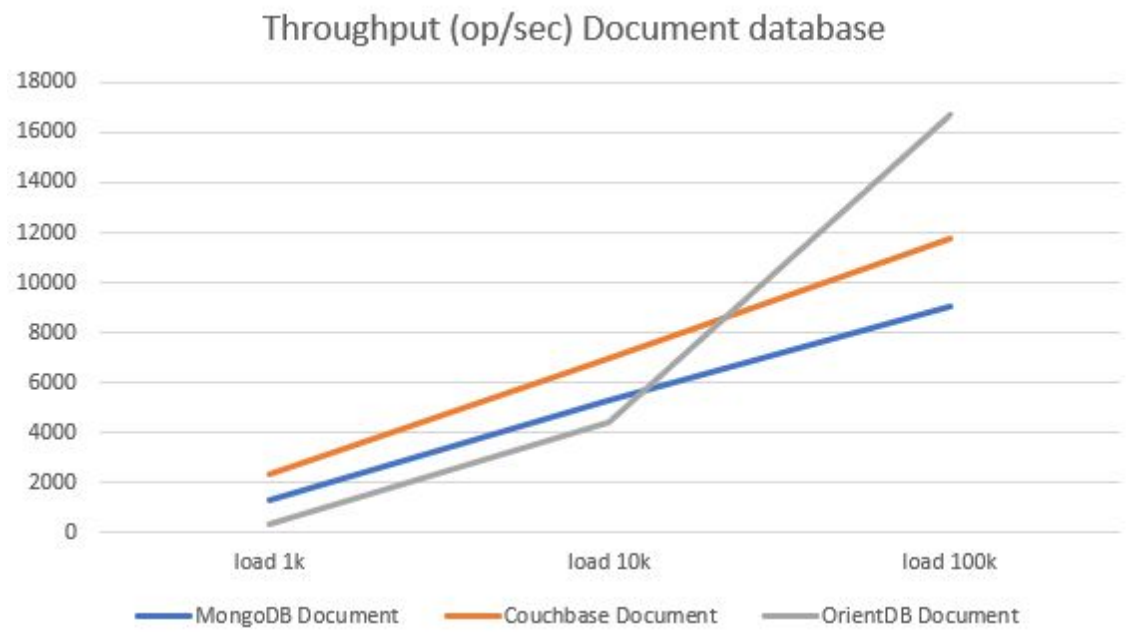

Figure 10: Throughput (operations done per second) for the load phases of 1 000, 10 000, 100 000 operations for Document Databases.

The throughput of load tests for document databases of MongoDB, Couchbase, OrientDB for load 1k, load 10k and load 100k are: Couchbase performs best for load 1k and load 10k but only in second place for load 100k. MongoDB is second place for load 1k and load 10k while MongoDB is third place for load 100k. OrientDB is last place for load 1k and load 10k but first place for load 100k. The throughput for these databases for load 1k: MongoDB (1295.336 op/sec), Couchbase (2325.58 op/sec), OrientDB (364.830 op/sec). The throughput for load 10k: MongoDB (5249.343 op/sec), Couchbase (6983.240 op/sec), OrientDB (4442.470 op/sec). Lastly for load 100k, the throughput was: MongoDB (9025.270 op/sec), Couchbase (11785.503 op/sec), OrientDB (16700.066 op/sec).

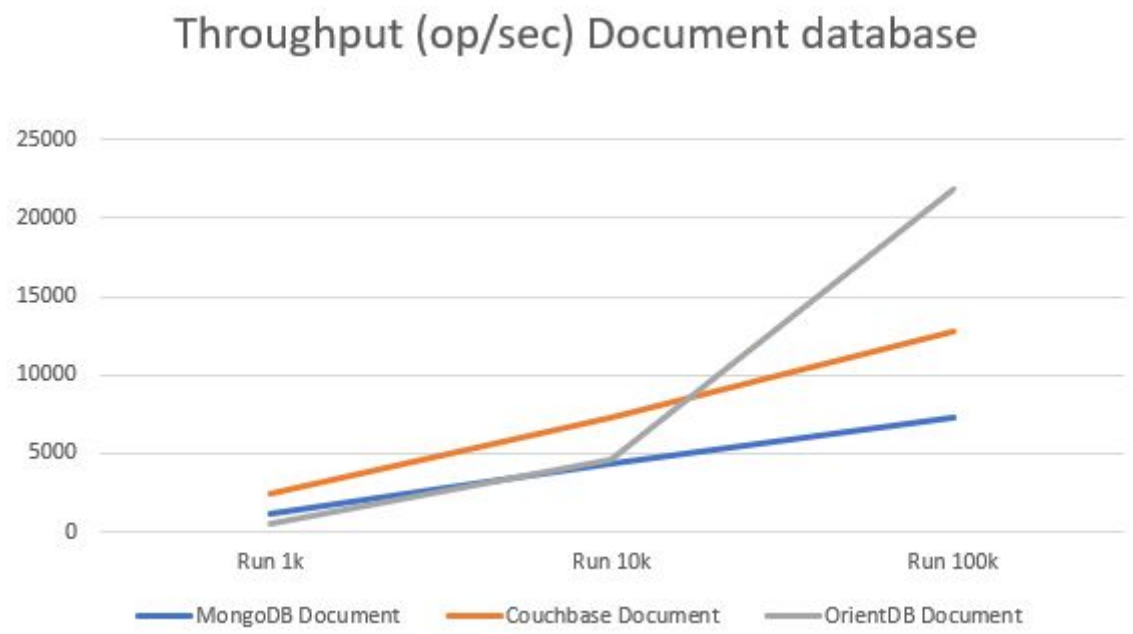

Figure 11: Throughput (operations done per second) for the run phases of 1 000, 10 000, 100 000 operations for Document Databases.

The throughput of run tests for document databases of MongoDB, Couchbase, OrientDB for run 1k, run 10k and run 100k are: Couchbase is in first place for run 1k and run 10k while Couchbase is in second place. OrientDB is in last place for run 1k and run 10k and first place for run 100k. MongoDB is second place at run 1k but last place for run 10k and run 100k. The throughput of these databases for run 1k: MongoDB (1218.026 op/sec), Couchbase (2392.344 op/sec), OrientDB (561.797 op/sec). The throughput for run 10k: MongoDB (4321.5211 op/sec), Couchbase (7299.270 op/sec), OrientDB (4555.808 op/sec). Lastly for run 100k, the throughput for these databases: MongoDB (7288.629 op/sec), Couchbase (12758.356 op/sec), OrientDB (21795.989 op/sec).

6. Discussion

In the following chapter we are going to discuss our results and findings with our literature review and previously done work in the field.

6.1 Different structural orientations

As we can see in figure 4 - 7 in results, there are differences between the different structural orientations, both when it comes to performance and scalability as we wanted to look at in research question 1. Between OrientDB, Redis and MongoDB the differences are not that large but one that was really bad in comparison was Cassandra database which is a column store type NoSQL database. In both loading and running phases the time it took to execute your operations was very high and it's scalability when performing many operations was horrible in comparison to the other databases. To give an example of this we compare OrientDB graph type with Cassandra column type and when loading 1000 entries in the database they both had pretty bad results in throughput (operations/seconds), OrientDB had 370 operations/ seconds and Cassandra had 162.5 operations/ seconds. When we tested to load 10 000 entries instead OrientDB did 4347.8 operations/ second while Cassandra remained to perform slow with only 395.9 operations/ second. We scaled the database even further and tested to load 100 000 entries and here OrientDB hit an remarcable 16380 operations/ second while Cassandra still only did 526 operations/ second. The same example can be made between the two in the running phase where we updated and read values instead of loading them but the results for Cassandra are even worse in comparison to OrientDB graph type structure. During our literature review phase of this paper we have come across other papers testing Cassandra with far better results then we got. The main reason our results could differ is the fact that we had to host our Cassandra database locally instead of on the cloud. Another reason for the bad results in this benchmarking test could be the usage of threads, Cassandra was the only database in our test with the option to select number of working threads when initializing the load or run phase, we chose not to do this since we didn't have the same

option for the other databases, and wanted to make the benchmarking test commands the same so that we dont give an edge to one database over another letting it work on many threads. Cassandra as default uses AP in CAP Theorem and from what we have discovered databases with availability as a trademark normally have good performance, which has led us to believe that our results from benchmarking Cassandra is not the real truth.

If we take a closer look at the loading phase where we input different amounts of data in the different structured typed databases we can clearly see some different behavior when it comes to performance. The key-value database Redis performs very well during low amounts of data but when scaled up the throughput starts to even out. The reason for Redis' good results on lower amounts of data could have to do with the fact the YCSB uses a primary key for its operations, this is exactly how key-value storage works in order to perform a query. So this could give Redis a better start on the lower data levels but it shows on higher levels that the throughput does not scale as well as graph based storage. The reason for this is that OrientDB graph based has the available trademark of CAP theorem which Redis does not.

On the contrary our graph database OrientDB has a lower throughput on lower amounts of data and scales up drastically when being tested on large amounts. Our document database MongoDB performs evenly during our test and scales up at an even linear pace.

When looking at the running phase where we perform read and updates on the inserted data we get similar results to the loading phase with some exceptions. Column-store is still by far the worst in both time and throughput, document store MongoDB still has an even speed across all tests and scales evenly across different amounts of data. Key-value stores perform a lot better in the running phase than in the loading phase, especially on large amounts of data and graph stores are still weak at low amounts of data and perform really well on the larger amount of data.

The reason for these results could be found in the CAP theorem, graph store OrientDB uses consistency and availability and uses strong consistency, the availability part in

OrientDB does so that less conflicts emerge and it will always be able to answer any request it gets. MongoDB which is a document store and Redis, our key-value store database both use consistency and partition, while MongoDB use strong consistency by default it takes a little bit longer time since it has to make sure that every node in the system gets the update, this causes the throughput to be a little slower per second. Redis on the other hand uses eventual consistency, while MongoDB slows down to make sure everything is updated before taking on the next operation. Redis simply does the operation without caring if every node is 100% up to date. Eventual consistency makes for faster throughput.

Since we did our benchmarks hosting locally the partition tolerance does not affect the performance in this case due to the fact that partition-tolerance is for distributed networks. Taking a look at Redis which is the simplest form of NoSQL and seeing that it performed best of all in the lower amounts of data up to 10 000 operations.

Time to execute both loading and running phases is highly linked to scalability and how good the database adapts to higher volumes of data. If the database scales and adapts to high volumes of data the running time in both phases goes down drastically. Column database does basically not scale at all and it's running time grows exponentially when the amount of data gets bigger. Our graph store database which adapts well to high amounts of data has almost the same time running 1k operations and 100k operations. What we could see from our results is that CAP theorem trademarks availability and eventual consistency are a factor for performance. Every database that possesses these trademarks is faster than a database with strong consistency.

6.2 Document structured orientation

When looking at our three document type databases in research question 2, we can spot similarities in performance at least at lower volumes of data. MongoDB and Couchbase perform nearly the same results at all levels of data but Couchbase is a little bit faster and has a little more throughput in both loading and running phase. The biggest difference between them is in the running phase on high volume of data where Couchbase scales just

a bit better than MongoDB. What really stands out in these tests are OrientDB at high volumes of data the throughput just excels. At our test with 100k operations OrientDB triples the throughput of MongoDB and dubbles the throughput of Couchbase and these databases are of the same structural type. What could be the reason for this, many things could be the reason but what we found in our literature review is that MongoDB and Couchbase is both written in C++ language this could be one reason for the near identically scaling performances. We can see that while MongoDB uses BSON storage type, Couchbase uses JSON this could be the answer to why Couchbase performs better on all phases. OrientDB stood out with it's exceptional performance when scaling to really large amounts of data and this was the only database written in the language of Java and using JSON storage type exactly like Couchbase which was the better C++ based database. Another thing that differs between all of these databases is their usage of CAP theorem, while MongoDB and Couchbase is using strong consistency and partition tolerance, OrientDB is using strong consistency and availability. The CAP theorem has an impact on performance on NoSQL databases. When using availability there are less conflicts to deal with since there is always an answer for an incoming request.

When choosing a database it helps to know how much data it needs to store because there are performance differences between all of them and the more data you enter, the more you see and feel the difference. Another thing you need to know is if your application always needs to display the latest input at all times, then you will have to choose a database with strong consistency and therefore sacrifice some performance. If you know from the beginning that your project will contain huge amounts of data and entries we can see from our results, that a NoSQL document based database will be the best performing since its ability to scale with large numbers of entries are very good.

6.3 Locally hosted databases

We have found that we have some different results than previously done work in this field. The main thing we had to differentiate from other papers is that we hosted our databases locally to run our benchmark tests. In comparison to other papers we

benchmarked a lot of different databases instead of just two, and other papers involving many databases focused more on the theoretical part then benchmarking. In order for us to make every database work with YCSB which is an open-source software and is not always 100% up to date with newer versions of the databases, we had to install and use sometimes deprecated versions of the databases to get them to work with YCSB. This also meant that we could not get all databases to work in the cloud so we made the decision to host them locally, and run our benchmarking tests on all of our chosen databases in the same way. We believe that in a clustered cloud environment with all the newest versions of every database, our results would have been somewhat different but we still got an result from both literature review and benchmarking, that there are differences in performance depending on what structural orientation you chose for your application or software.

7. Conclusion

The case presented in this thesis was to see if there were any performance differences between the different structures of NoSQL, in addition to this we wanted to see if there are any performance differences between different databases of the same NoSQL structure in this case document based orientation. The findings of this study is that there are differences in performance, at lower volumes of data Redis, a key-value based NoSQL database performed best and at high volumes of data OrientDB document based, was the highest performance NoSQL database. For this study it is important to mention that benchmarking is done with locally hosted databases and the same benchmarking tool is used for every database without taking their structural orientation into account.

It is proved that there are differences in performance between the different structures of NoSQL, some structures perform better on lower volumes of data while others perform much better on higher volumes of data.

Since hosting the databases locally the full effect on CAP theorem cannot be applied since partitioning does not work on a single node but attributes for consistency and availability seems to have an effect.

When looking at the research on our chosen document based NoSQL databases we clearly could see performance differences there as well, both on low volumes and high. What we found in research from other papers are their differences in development language, storage type and their specific use of the CAP theorem.

7.1 Further work

As further work more testing could be done with different types of operations, for example a workload with more reads then updates or more updates than reads. We could also do cloud cluster testing to see if the different databases behave in a different way when being hosted on the cloud instead of being hosted locally. Further we can find

benchmarking tools for every different structure that takes the different database specifics into account.

8. References

1. Mukherjee S. The Battle between NoSQL Databases and RDBMS. IJIRSET [Internet]. 2019 [cited 2020-03-22];8(5): 5780-5790. Available from:

http://www.ijirset.com/upload/2019/may/107_The.pdf

2. Cooper B. Yahoo! Cloud Serving Benchmark. Yahoo! [Internet]. 2010 [cited 2020-03-22]; Available from: https://ycsb.site/

3. Reniers V, Van Landuyt D, Radique A and Joosen W. On the state of NoSQL Benchmarks. ACM [Internet]. Available from:

https://dl-acm-org.ezproxy.hkr.se/doi/abs/10.1145/3053600.3053622

4. Frank L, Frank C, Larsson J, Pedersen R. The CAP theorem versus databases with relaxed ACID properties. ICUIMC [Internet]. 2014 [cited 2020-04-15]; 1-7. Available from:
https://dl-acm-org.ezproxy.hkr.se/doi/10.1145/2557977.2557981

5. Huang X, Wang J, Zhong Y, Song S, Yu P. Optimizing data partition for scaling out NoSQL cluster. John Wiley & Sons, Ltd [Internet]. 2015 [cited 2020-04-16]; 27(18): 5793-5809. Available from:
https://onlinelibrary-wiley-com.ezproxy.hkr.se/doi/full/10.1002/cpe.3643

6. Beigi N, Fokaefs M, Khazaei M, Gaikwad P, Ramprasad B, Litoiu M, Zareian S. How do I choose the right NoSQL solution? A comprehensive theoretical and experimental survey. Journal of Big Data and Information Analytics (BDIA) [Internet]. 2015 [cited 2020-05-09]; 1-33. Available from:
https://www.researchgate.net/publication/282679529_How_do_I_choose_the_right_NoSQL_solution_A_comprehensive_theoretical_and_experimental_survey

7. Chen L, Davoudian A, Liu M. A Survey on NoSQL Stores. ACM Computing Surveys [Internet]. 2018 [cited 2020-03-22]; 51(2); 40:0-43. Available from:
https://doi-org.ezproxy.hkr.se/10.1145/3158661

8. Bernardino J, Cabral B, Diogo M. Consistency Models of NoSQL Databases. MDPI AG [Internet]. 2019 [cited 2020-04-15]; 11(2): 0-19. Available from: https://www.mdpi.com/1999-5903/11/2/43/htm

9. Bai P, Huang X, Wang J, Yu P, Zhang J. An experimental study on tuning the consistency of NoSQL systems. John Wiley & Sons, Ltd [Internet]. 2017 [cited 2020-04-15]; 29(12): 1-16. Available from: https://onlinelibrary-wiley-com.ezproxy.hkr.se/doi/full/10.1002/cpe.4129

10. MongoDB. Top 5 Considerations When Evaluating NoSQL Databases. MongoDB [Internet]. 2018 [cited 2020-04-06]; 1-11. Available from: https://webassets.mongodb.com/_com_assets/collateral/10gen_Top_5_NoSQL_Considerations.pdf

11. Bernardino J, Cabral B, Carreiro P, Lourenco J, Vieira M. Choosing the right NoSQL database for the job: a quality attribute evaluation. Journal of Big Data [Internet]. 2015 [cited 2020-05-09]; 1-27. Available from: https://link.springer.com/article/10.1186/s40537-015-0025-0

12. Kaur P, Sharma G. Scalable Database Management in Cloud Computing. ICECCS [Internet]. 2015 [cited 2020-04-16]; 70: 658-667. Available from: https://doi.org/10.1016/j.procs.2015.10.102

13. Abramova V, Bernardino J. NoSQL Databases: MongoDB vs Cassandra. C3S2E '13: Proceedings of the International C* Conference on Computer Science and Software Engineering. NY: Association for Computing Machinery; 2013 [cited 2020-05-17]; 14-22. Available from: https://dl.acm.org/doi/abs/10.1145/2494444.2494447

9. Appendix

This chapter provides additional information that could be interesting for the reader.

9.1 Complete records of benchmarking different structures

Different structural orientations:	Orient Graph	Redis Key-Value	Cassandra Column	MongoDB Document
Load 1k				
Time (ms)	2700	179	6153	772
Throughput (op/sec)	370,37	5586,592	162,522	1295,336
Load 10k				
Time (ms)	2300	1069	25257	1905
Throughput (op/sec)	4347,826	9354,536	395,929	5249,343
Load 100k				
Time (ms)	6105	9330	190102	11080
Throughput (op/sec)	16380,016	10718,113	526,033	9025,27
Run 1k				
Time (ms)	1794	133	6251	821
Throughput (op/sec)	557,413	7518,796	159,974	1218,026
Run 10k				
Time (ms)	2178	674	24948	2314
Throughput (op/sec)	4591,368	14836,795	400,933	4321,5211
Run 100k				
Time (ms)	4659	5139	187554	13720
Throughput (op/sec)	21436,833	19459,038	533,179	7288,629

Figure 12: All data collected presented in an excel document, both load and run phases for benchmarking databases with different structural orientations.

9.2 Complete records of benchmarking document structure

Document structural orientation:	MongoDB	Couchbase	OrientDB
Load 1k			
Time (ms)	772	430	2741
Throughput (op/sec)	1295,336	2325,58	364,83
Load 10k			
Time (ms)	1905	1432	2251
Throughput (op/sec)	5249,343	6983,24	4442,47
Load 100k			
Time (ms)	11080	8485	5988
Throughput (op/sec)	9025,27	11785,503	16700,066
Run 1k			
Time (ms)	821	418	1780
Throughput (op/sec)	1218,026	2392,344	561,797
Run 10k			
Time (ms)	2314	1370	2195
Throughput (op/sec)	4321,5211	7299,27	4555,808
Run 100k			
Time (ms)	13720	7838	4588
Throughput (op/sec)	7288,629	12758,356	21795,989

Figure 13: Complete records of the data gathered when benchmarking document structured databases, both load and run phases.