Performance and Comprehensive Comparison of NoSQL Data Models for Big Data

Laura Beaudoin
Department of Computer and Software Engineering
Polytechnique Montreal
Montreal, QC, Canada
laura.beaudoin@polymtl.ca

Reetesh Dooleea
Department of Computer and Software Engineering
Polytechnique Montreal
Montreal, QC, Canada
reetesh.chandranath-dooleea@polymtl.ca

Mohsen Rahmati
Department of Computer and Software Engineering
Polytechnique Montreal
Montreal, QC, Canada
mohsen.rahmati@polymtl.ca

Ming Xiao Yuan
Department of Computer and Software Engineering
Polytechnique Montreal
Montreal, QC, Canada
ming-xiao.yuan@polymtl.ca

***Abstract*—**The growing trend in using big data has increased the requirement for data storage in the industry. The limited pace of accessing relational databases (RDB) causes more preference for NoSQL databases. There are different NoSQL databases. We try to explain and compare the most important types of NoSQL databases including Document-oriented databases, graph databases, and key-value databases with YSCB benchmarking. For this paper, we concentrated on the specific implementations of MongoDB, OrientDB and Redis.

Keywords— NoSQL databases, big data, document-oriented databases, graph databases, key-value databases, benchmarking, YCSB, MongoDB, OrientDB, Redis, throughput, latency, CRUD, runtime

I. INTRODUCTION

The volume of information in the world is said to double every two years by experts. These massive websites, like Google, Facebook, Twitter, and even YouTube, produce a tone of information every day. Therefore, it is a challenging task to effectively store and retrieve this information. Databases are the primary solution to this demand for organized storage and retrieval of information since they can handle larger numbers of transactions and results of testing [1] [2]. Without processing Big Data, we risk missing the results of their invisible data. It is one of the major issues faced by huge businesses and big data analysis [3].

The study [4] found that the Big Data was typically defined with five Vs (Volume, Velocity, Variety, Veracity, Value). Volume is related to the fact that every second, a significant number of data sets are produced. Rapid reproduction is referred to as velocity, and it describes the pace at which new data is created, as well as the rate at which it is transferred. Variety is about gathering numerous types of data, both structured and unstructured such as texts, social media chats, images, sensor data, videos, voice recordings, and conventional structured data. Veracity is related to consistency, precision, quality, and dependability of Big Data. It describes data that is incorrect, has outliers, or has missing values. Finally, the usefulness of the obtained data for your organization is referred to as big data value.

Relational databases weren't built to handle change. In Relational databases, data modelling must be done in advance and, depending on the system, it might take months or even years. It takes a lot of time and resources to make changes afterwards, and database modelling efforts can take years and cost millions of dollars. Big data requires a database infrastructure that is adaptable and forgiving since it is continuously changing [5]. Users have a variety of alternatives for addressing the issue associated with such data in order to solve it. For instance, users can use different database technology, such as NoSQL databases, to store and handle large scale datasets [6]. But NoSQL databases has more flexible data structures including graphs and key-value data structures, so they are better with continuously changing in Big Data.

NoSQL databases can handle Big Data better than SQL databases because NoSQL databases have better method scaling which is horizontal instead of vertical in SQL. It means we could have more machines to do the task in horizontal scaling but whereas vertical scaling is the process of enhancing an existing computer with greater power (such as CPU and RAM) [7]. So, because of processing data in parallel, horizontal scaling is better with adding data continuously in Big Data.

There are four basic types of NoSQL databases including key-value store, document-based store, column-based databases, and graph-based store. First, the foundation of a Key-value database is a hash table, which contains a distinct key and value for each piece of information. The saved values can be integers, strings, JSON, XML, HTML, binaries, pictures, videos, and a select few other types of data [8]. The items are recognized and retrieved using a key. For the second type, semi-structured data is stored, recovered from, and managed using a document-based NoSQL database [9]. In the case of column-oriented databases, data is kept in cells that are arranged in columns of data. Lastly, graph-based NoSQL databases represent and store data using nodes and the relationships between nodes [10].

As for the following sections, notably in section two, we will discuss related works of NoSQL databases. In section three, we will explain the general description and structure of graph-based

databases, document-based databases, and column-based databases. In that section, additionally we will additionally describe advantages and disadvantages of each type. Also, a specific implementation of each type will be discussed. In the methodology section, the benchmarking will be defined and executed. Also, we will talk about used packages and environment configuration to be able to perform the benchmarking. Furthermore, the GitHub repository used for the benchmark can be found here. In the result section, we will analyze and explain the results obtained from benchmarking three times each database for different workloads. In the last section, the conclusion will be presented.

II. Related Works

In this section, we discuss the research areas related to our work which explore the characteristics of NoSQL data models and how to test their performance with benchmarking.

Khazaei, Hamzeh & Fokaefs, Marios & Zareian, Saeed & Beigi, Nasim & Ramprasad, Brian & Shtern, Mark & Gaikwad, Purwa & Litoiu, Marin [11] compare the benefits and weaknesses of different NoSQL databases like document stores, graph databases, key-value data stores and wide-column stores with a big data approach. It also gives some examples of concrete implementations of those different NoSQL databases models.

Since the transaction data for the e-commerce sector frequently needs to be connected, Apache HBase is an excellent option in the broad column store NoSQL database category [12]. In the news industry, the suitable NoSQL database category is the document store, and the better choice is MongoDB [12].

Additionally, the most well-known NoSQL databases that are assessed using YCSB include MongoDB, CouchBase, Cassandra, HBase, and Redis [13, 14]. Cassandra performed better than the other while the workload was executed as 50% Read and 50% Write operations. Additionally, Surya et al., who worked with YCSB, MongoDB, CouchBase, Cassandra, and HBase, discovered that MongoDB performed best while working with 100% Read and 100% blind Write, whereas HBase performed better when working with tiny database collections [13].

Redis is particularly suitable for loading and executing workloads for small datasets, but Redis has poor performance for the issues of vast amounts of datasets, as mentioned by EnqingTang et al. who evaluated performance between different NoSQL databases including Redis, MongoDB, and CouchBase using YCSB [14].

The study used a variety of NoSQL databases, including Redis, MongoDB, and CouchBase, and found that Redis is best for doing high-performance analyses on small datasets, MongoDB is best for handling big amounts of data, and CouchBase is better for fault-tolerant database environments [15]. However, CoucheBase is used to handle concurrent read and write operations in huge data sets, however because of its poor performance, Cassandra is advised in this situation [16].

The effectiveness of relational and non-relational database management systems was assessed using two models, one of which was deployed on a small scale of data and the other on a big size. To determine the benefits and drawbacks of both types of DBMS, performance was assessed based on CRUD operations. The amount of the data and the length of the queries were used to assess performance [17, 18].

III. Databases Description

This section will cover 3 different types of NoSQL databases with their benefits and weaknesses, and it will also provide examples of concrete implementations of those models and how they correspond to the theorical database model.

Before describing the databases, it is important to understand the principles of the CAP theorem. CAP consists of Consistency, Availability, and Partition tolerance. According to the CAP theorem, you can guarantee either consistency or availability if a distributed database has a network failure. When the value of the information returned must be correct, consistent databases should be utilised. Also, utilizing availability databases is advised when the service is more crucial than the data [19].

A. Document-oriented databases

As a first solution to avoid relational databases, document-oriented databases are used. They intuitively store data as documents, instead of rows and columns. According to Digital Ocean, documents can be described as “a self-contained data entry containing everything needed to understand its meaning, similar to documents used in the real world” [3.1].

Many languages can be used to represent the document entities, such as JSON, XML or YAML. JSON is the most common [3.1]. Here is an example of what a document can look like:

```
{
    "_id": "sammyshark",
    "firstName": "Sammy",
    "lastName": "Shark",
    "email": "sammy.shark@digitalocean.com",
    "department": "Finance"
}
```

Fig. 1. Example of document entry [3.1]

What is important to note about document-oriented databases is that they do not need to have a uniform structure between records (documents) [3.1]. That means that all the documents do not need to have the same field/value pairs. Take note that those fields can contain arrays or even nested objects.

That type of database is commonly used by businesses to store customers, for example, which can be retrieved for different purposes. In this case, the document database is used as the central holder information. Document-oriented databases also work well to manage inventory or even product development [3.2].

For the advantages, we note that not only the field (key) is searchable, the value also is. In addition, this type of data stories allows inserting, retrieving and manipulation of semi-structured data [11]. Because of the flexibility of the structure of documents, it is also used for its great flexibility and adaptability of the database. It also helps with the ability to manage data, structured or unstructured. Finally, the last big advantage is that it has a good scalability, because it is easy to scale horizontally documents which are independent [3.1]. Those elements all facilitate storing of data when the database gets bigger, which is frequent when talking about big data.

On the other hand, this database type has some issues with consistency-check, because for example, you could duplicate the same information in multiple documents. Also, there is a limitation related to the atomicity, because if like we just

explained the same information is duplicated, you will need to make two requests [3.3]

Now that we know what document-oriented databases are made of, we can look at a concrete implementation of such a database, like **MongoDB** which is widely used [3.1]. MongoDB has consistency with partition tolerance (CP) based on CAP Theorem [19].

First, the language internally in MongoDB is JSON, like typically seen in document-oriented databases, but it can also accommodate BSON (Binary JSON), which is less performant than JSON, but supports more data types, which is less restrictive [3.1, 11].

Like document-oriented databases, MongoDB provides a document query mechanism by providing atomic operation on fields. It is also possible to nest document structures, like it is supposed to, but it can also retrieve them and manipulate them, which reduces the number of queries you must make [11]. It also supports a Master-Slave replication architecture to maintain multiple copies of data [11].

Unfortunately, like other NoSQL databases models, MongoDB, like document stores, does not ensure the ACID properties. On the other hand, it follows the BASE ones [11].

With that information, it is safe to say that MongoDB is a form of document-oriented database since it has the same structure and has the same benefits and weaknesses.

B. Graph databases

Now, for our second NoSQL database, let's look at graph databases. They represent data structured as a graph with nodes and edges, which are the relationships between data. This makes the processing of data and calculation of properties of the graph easier [11]. Once again, graph databases are not using a predefined schema like we used to have in relational databases, which make it flexible [3.4].

To store data, as mentioned, we have nodes and relationships between them. The nodes can contain attributes to describe them. In the figure below, you can see an example of a graph database.

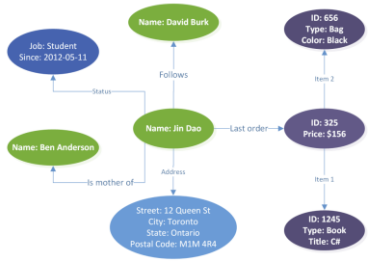

Fig. 2. Example of a graph data model [11]

Graph databases are very performant and flexible because we do not have an extra layer of abstraction for graph elements needed to fetch graph data. Those qualities are particularly useful in the era of big data [11].

Also, since this type of NoSQL database is very simple and easy to use, it's growing in popularity. It is used in multiple scenarios, like fraud detection, master data management, recommendation engines, etc. [3.4].

There are multiples reasons why it would be a good idea to use a graph database. First, graphs are everywhere, so it can be implemented intuitively. Second, it supports simple modeling. Third, just like the document-oriented model, you can have structured or unstructured data, or a mix of them both, which is convenient. Also, the querying is simple when comparing to similar query in SQL. You also can achieve a better performance because they use a simple index. Finally, you can easily get a visualization of your data. All of this leads to many advantages, like performance, an object-oriented mindset, flexibility, etc. [3.4].

However, there are also some drawbacks of using this type of database. For example, there is no standard query language, which helps with the flexibility, but you probably will have a bigger learning curve. It is also not appropriate for transactions, because they are not good at processing a lot of transactional data and are not efficient to query the entire database. Lastly, since it's not yet as common as relational databases, it might be harder to get the support you need to develop it [3.4].

It's now clearer what a graph database consists of. To go further, we will explore a specific implementation which permits this type of database, like **OrientDB** [11]. OrientDB has both consistency with availability (CA) and partition tolerance with availability (PA) according to your configuration [20]. You may make a CA by creating one master node and the number of synchronously replicated slaves. If you add asynchronous servers, eventual consistency can be possible if the client reads from the replicated asynchronous server [20].

It can be considered as a Graph database system because it supports relationships, which is permitted by document pointers. To do so, we have JSON documents containing document pointers, pointing to other documents. With this implementation, pointers are edges [11]. Because of that, we can say that it combines the power of graphs and the flexibility of documents [3.5].

Because of that specific implementation, OrientDB is very scalable, performant and supports multimodel databases. It reduces the complexity and makes it easier to maintain consistency in data, just like graph databases have been described above [3.5]. Also, thanks to the presence of authentication, password, and data-at-rest encryption, it is sage to use it.

Although OrientDB can provide scalability, the Community Edition does not support it. Another limitation is that it doesn't have an import tool, the documentation is outdated, and the graph editor has bugs. However, those issues are related specifically to graph databases, so it is not necessary to analyse them further.

Finally, just like graph databases, OrientDB has the same types of use cases confirms that it is a graph database [3.6].

With all those resemblances in implementation and strengths and the various resources leading to the conclusion that OrientDB can in fact be used as a graph database.

C. Key-Value pair-based database

The key-value pair-based databases include a group of key-value pairs in which each key acts as an unique identifier [25]. Keys and values can be both basic and complicated composite objects.

Figure 3 shows an example for a key-value database which is a MAC table for forwarding network packets.

MAC table

Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1

Fig. 3. Example of key-value database [26]

What we could mention as an advantage is easy designing and implementing [26]. Also, The built-in redundancy of this database type improves dependability and is a highly beneficial feature [27].

Unlike relational databases, and like previous databases we analyzed, we do not have predefined structure for key-value databases [27]. Despite lacking a query language, key-value stores do offer a mechanism to create and remove key-value pairs [26]. Values are unable to be searched for or inquired on. It is possible to query only the key [26].

One of the most popular implementation for key-value database is **Redis** as an open source and in-memory database. Imagine it as a dictionary with any number of keys, each of which may have a value set or retrieved. Redis can prepare consistency with partition tolerance (CP) [19].

Redis, however, goes beyond a straightforward key/value store, because it is truly a data structures server that supports a variety of values [28]. It is suitable for usage as a message broker, database, or cache. Also, Redis supports a variety of data structures and integrated replication. Moreover, keys can have a size up to 512MB [28].

As the first disadvantage, data intended to be written to a slot will be lost if its master slot holding those slots is down [3.11]. Another disadvantage is that since it runs in memory, it needs a lot of RAM. Hence RAM servers shouldn't be used or at least, should be prevented [29].

There are similarities between advantages and disadvantages of key-value database and Redis. So it confirms that we can consider Redis as a key-value database.

IV. Methodology

In this section, the term benchmarking will first be defined alongside the *Yahoo! Cloud Service Benchmark* (YCSB) tool. Afterwards, the environment configurations and used packages will be elaborated as well as the steps to perform the benchmarking with the help of a Shell script file.

A. Benchmarking and YCSB

Benchmarking a database is an essential strategy to validate the performance of our big data model. Hence, benchmarking refers to the process of measuring and comparing software performance to identify opportunities for improvement [30]. In parallel, the *Yahoo! Cloud Service Benchmark* (YCSB) is an open-source NoSQL benchmark suite, allowing users to measure the performance of NoSQL data models with the help of simple CRUD (Create, Read, Update and Delete) operations on synthetically generated data [31]. Moreover, the YCSB directory comes with 6 built-in workloads where each workload has a different configuration or proportion of each CRUD operation. The tool itself is used in two steps: firstly, the data to be inserted in the database needs to be loaded and secondly, the chosen workload, which defines the operations to be executed against the dataset, is executed. Thus, this tool will allow us to perform benchmark suites on our three chosen NoSQL data models.

The following table shows the 6 built-in default workloads integrated in the YCSB directory which were also used in our benchmarking experiments.

TABLE I. DEFAULT YCSB WORKLOADS PROPORTIONS

	Workloads					
Operations	***A***	***B***	***C***	***D***	***E***	***F***
Read	0.5	0.95	1	0.95	0	0.5
Update	0.5	0.05	0	0	0	0
Scan	0	0	0	0	0.95	0
Insert	0	0	0	0.05	0.05	0
Read-Modify-Write	0	0	0	0	0	0.5

It's interesting to explicit the meaning behind every operation. The read operation is to fetch information from a database. The update operation is to update one or more records, which are already available in the database. The scan operation is to scan on an index that is not guaranteed to return zero or one row. The insert operation is to add a new record into the database. Finally, the read-modify-write operation is a series of operations that fetch the data, modifies it, and writes the updated data into the database.

B. Environment configurations and specifications

All procedures were made possible with the Amazon EC2 instance on a virtual machine provided by the Teaching Assistants. An EC2 instance of type *t2.large* with a vCPU of 2 and 8GiB memory was configured. An SSH client enabled a connection to the Linux instance via a generated SSH key. The operating system (OS) image used was Ubuntu of version 22.04.1 with 64-bit (x86). Once everything was set up and running, base packages such as Java, Maven, Docker, Docker-compose, Curl and so on were installed via the command line. Afterwards, the latest YCSB, i.e., ycsb-0.17.0, was fetched and extracted to a local working directory. The download command for this tool and the general benchmarking steps are detailed in the README file of the provided GitHub repository located here. Once again, as for our working directory, all our folders, files, benchmarking script and results were added to our public GitHub repository found on this link.

C. Benchmarking workflow

The whole benchmarking process was highly dependent on two common tools: Docker and Docker-compose. Docker is a software platform that allows developers to build, test and deploy applications based on containers [32]. As for Docker-compose, the latter is simply a tool for creating, defining, and running multi-container Docker applications at once [33]. This

strategy would enable us to run the YCSB tool with the selected databases using containers. In our case, these tools allowed us to define and run several nodes for our databases with a simple command.

The number of nodes chosen for this experiment was six for all databases. Hence, six nodes were each created for MongoDB, Redis and OrientDB using three distinct *docker-compose.yml* files in which each node was configurated and defined based on the database distributed architecture explained in the previous sections. It is also important to note that the */etc/hosts* file was modified to add the IP addresses of all six nodes.

Once all three *docker-compose,yml* files were produced, a Shell script file was then created in order to handle the state of the containers and to perform the benchmarking automatically rather than having to execute each instruction manually in the terminal. The script then generates the output result of the load and run instructions as text files and saves them in a distinct folder. The strategy of the script was for it to be as generic as possible to reuse it to benchmark different databases. To achieve this behavior, the script must be run with three arguments:

1) The database (argument -d) where the options are either mongodb, redis, or orientdb.

2) The workload alphabet (argument -w) where the options are either a, b, c, d, e, or f.

3) The execution number (argument -e) which can be 1, 2, 3 or any number since it is just a number used to reference the naming of the saved output file.

An example for running the script in order to benchmark the Redis database for the workload *a* for the 2nd time is as follows:

./script.sh -d redis -w a -e 2

The script begins with a validation of all its shell arguments. Then, the benchmarking begins. Firstly, we run the docker-compose up command with the associated docker-compose file to build and start the containers and create six nodes. Secondly, we wait for about 10 seconds to make sure the nodes have enough time to communicate with each other. Thirdly, we load the data, i.e., the data to be inserted in the database. For this step, we chose a record count of 1000 operations. We also save the loaded output in a text file. Fourthly, we execute the chosen workload, in which the proportion of the CRUD operations are defined. Once again, we save the output run file in a text file. At the end, we clean everything: we run the docker-compose down command which stops and removes all containers and nodes. The goal of this step is to be able to restart this whole cycle again with fresh nodes and containers before performing the benchmarking again. Finally, this process is repeated three times to generate three different run outputs. Afterwards, the three files are exported to an Excel file in which the average throughput and other metrics are calculated. At the same time, the Excel file produced charts based on our average results upon which we are then able to interpret and analyze.

V. RESULTS

After repeating the benchmarking three times for each database and generating three output run files for each experiment, the average metrics were then computed in an Excel file.

A. Average results of workloads

In this section, we present the average benchmarked results and observations. Hence, the following tables display the average results obtained from our experiments for each workload. It is to be noted that only metrics relevant to a workload are considered and presented.

1) Workload A (50% Read – 50% Update)

TABLE II. WORKLOAD A

	Database		
Metrics	***MongoDB***	***Redis***	***Orientdb***
Throughput (us)	153.0187607	1768.01023	157.345442
Latency Read (us)	1619.158992	222.472577	658.082302
Latency Update (us)	10123.87103	245.9139296	1806.225385

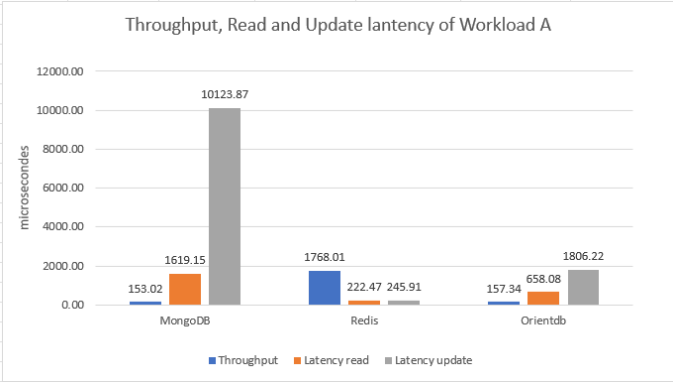

Fig. 4. Throughput, Read and Update Latency for Workload A

For workload A, the corresponding metrics are the throughput, the read latency, and the update latency. We can see that Redis is the best database to use in case of 50% read and 50% update and is the overall quickest among the 3 databases. Another point is that by dividing the throughput metrics, we also observe that it is about 11 times better than MongoDB and OrientDB for the throughput. We can see that the latter has the highest throughput, the lowest read latency and the lowest update latency.

2) Workload B (95% Read – 5% Update)

TABLE III. WORKLOAD B

	Database		
Metrics	***MongoDB***	***Redis***	***Orientdb***
Throughput (us)	413.8554384	1999.65426	125.984004
Latency Read (us)	1089.342496	167.782009	645.114403
Latency Update (us)	12600.32827	285.7580278	3737.539527

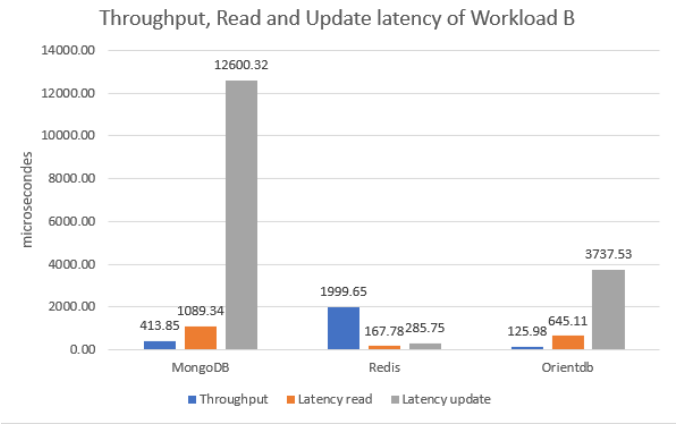

Fig. 5. Throughput, Read and Update Latency for Workload B

For workload B, the corresponding metrics are also throughput, read latency and update latency, but instead we have 95% read and 5% update. Redis is yet again the best database to use. It has the highest throughput, the lowest read latency and lowest update latency among the 3 databases. However, when MongoDB operates more read operations, Redis now is only about 5 times better than MongoDB as compared to workload A. On the contrary, while OrientDB has more read operations, Redis is now 15 times better than this database which shows that Redis is the best among the three for this workload.

3) *Workload C (100% Read)*

TABLE IV. WORKLOAD C

	Database		
Metrics	***MongoDB***	***Redis***	***Orientdb***
Throughput (us)	551.4689193	2031.80119	197.559563
Latency Read (us)	1034.076333	166.576333	642.063333

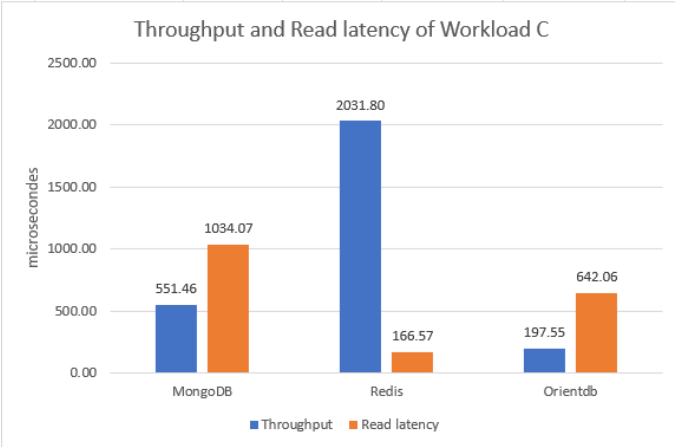

Fig. 6. Throughput and Read latency for Workload C

For workload C, the corresponding metrics are throughput and read latency and we have 100% read operations. Logically, the best database should be the same from workload B because we have majority of read operations. Based on the results, the best database is the same as the one for workload B, Redis. It has the highest throughput and the lowest read latency. Redis is 3 times better than MongoDB and 10 times better than OrientDB. This also shows that MongoDB performs best when it executes more read operations.

4) *Workload D (95% Read – 5% Insert)*

TABLE V. WORKLOAD D

	Database		
Metrics	***MongoDB***	***Redis***	***Orientdb***
Throughput (us)	398.1076746	1520.05213	107.64434
Latency Read (us)	1044.816611	221.17479	683.112897
Latency Insert (us)	13915.15973	820.21625	6467.66174

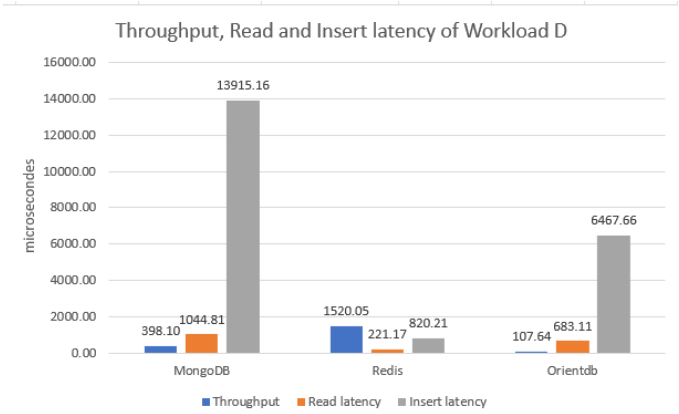

Fig. 7. Throughput, Read and Insert Latency for Workload D

For workload D, the corresponding metrics are throughput, read latency and insert latency. We have 95% read and 5% insert operations. Similarly, to workload B and C, most operations are read, it would be logical to assume that Redis is yet again the best database in terms of performance. It has the highest throughput, lowest read latency and lowest insert latency.

5) *Workload E (95% Scan – 5% Insert)*

TABLE VI. WORKLOAD E

	Database		
Metrics	***MongoDB***	***Redis***	***Orientdb***
Throughput (us)	337.754897	193.585906	34.3681573
Latency Insert (us)	12518.79099	1025.496409	4941.82577
Latency Scan (us)	1657.24857	5025.28068	4972.27001

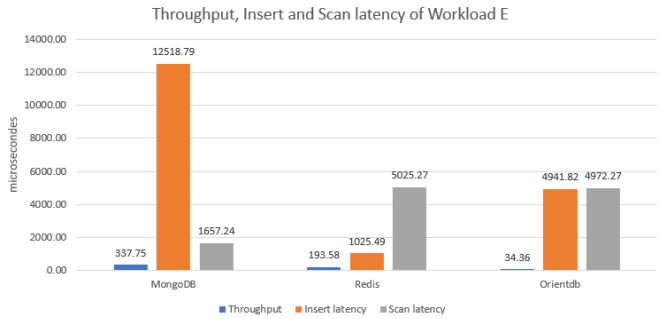

Fig. 8. Throughput, Insert and Scan Latency for Workload E

For workload E, the corresponding metrics are throughput, insert latency and scan latency. We have 95% scan and 5% insert operations. This workload is interesting to analyze because MongoDB has the better throughput and scan latency while Redis has the best insert latency. But if we consider the fact that most of the operations are scan operations, MongoDB would be the best database to use, plus it has a better throughput than Redis.

6) Workload F (50% Read – 50% Read-Modify-Write)

TABLE VII. WORKLOAD F

	Database		
Operations	***MongoDB***	***Redis***	***Orientdb***
Throughput (us)	138.4780251	1601.34721	20.5615467
Latency Read (us)	1243.812	162.343333	571.846333
Latency Read Modify Write (us)	11536.45515	415.6651371	2796.80755

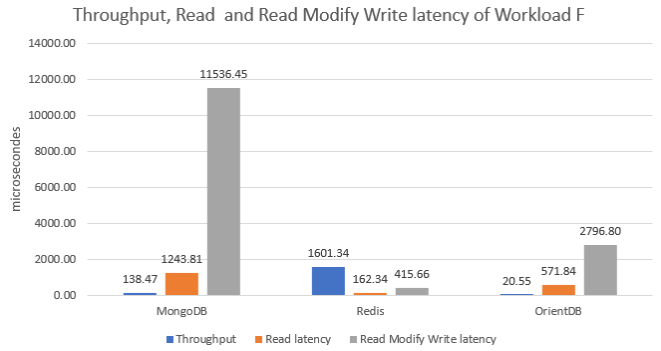

Fig. 9. Throughput, Read and Read-Modify-Write Latency for Workload F

For workload F, the corresponding metrics are throughput, read latency and read modify write latency. We have 50% read and 50% read modify write. Redis is yet again the best database to use because it has the highest throughput, the lowest read latency and the lowest read modify write latency. We observe that MongoDB is the lowest for the read-modify-write latency by far.

B. Interpretation and analysis

For the analysis, we will be comparing the performance of the three benchmarked databases based on the properties of the corresponding workloads. The values of the metrics used in this section are provided from three separate executions of the workload on each database.

1) Analysis of best database per Read and Write latency and Throughput

By comparing the performance of the 3 databases with similar workloads, the first thing we realize is that Redis is by far the outperforming database. It is definitively the best database to choose if we want to perform a lot of read operations. It is also the best database to choose if we want to perform a lot of write operations. In terms of update operations, from the data collected from workload A and B, Redis has the lowest update latency of them all. Furthermore, in terms of insert operations from workload D and E, even though it represents a small percentage of all the operations, Redis still has the lowest insert latency. In terms of throughput, Redis is also the outperforming database, having the highest throughput in 5 out of the 6 workloads. The only one where it is not the highest throughput is the workload E. So, if we want to perform a lot of scan operations and throughput is a key metric to evaluate, I recommend MongoDB, but Redis is still by far the most versatile database.

2) Analysis of the performance of each database against different workloads

It would be fair to compare the performance of the three databases against different workloads.

For MongoDB, we can see that workload C does 1 operation, i.e., only read operations, instead of 2 operations as compared to the other workloads. It greatly affects its throughput as the workload C has the highest throughput performance for MongoDB. The fact that workload C does 1 operation also affects its read latency. By doing only 1 operation, it has a lower average of read latency compared to the other workloads. For workload A and B, we can see that having a majority of read operations (workload B, C and D) comparing half-half (workload A) affects slightly MongoDB's performance as it's read latency only drops a little bit. Based on workload E and F, the insert latency varies slightly as they have the same percentage of operations in both workloads (5%).

For Redis, by doing 1 operation (workload C) comparing of 2 operations from the other workloads, Redis have a slightly higher throughput. For workload E, we realize that its throughput is significantly lower when it has a high percentage of scan operation. We realize also that having a majority of read operations (workload B, C and D) comparing half-half (workload A) slightly affects Redis's performance.

For OrientDB, we can see that the read latency varies slightly when we have a high percentage of read operations (workload B, C and D) compared to half-half (workload A). Furthermore, by having a smaller percentage of update operations (workload B), the update latency almost doubled compared to workload A. OrientDB's throughput are at their lowest for workload E and F as scan and read modify write latency seems to affect it greatly.

3) Analysis of the best workload for each database

By comparing the different workloads per database, it's undeniable to say that Workload C is better for MongoDB as its throughput is the highest compared to other workloads. It's read latency it's not the fastest of all the workloads but it's the best choice for it overall.

Workload B is the best workload for Redis because it has the highest throughput among workloads that has at least 2 operations. It also has the lowest average latency for read and update. Workload F could be another option because it has a relatively good throughput and read latency, but in the case of this paper, we will assume that the read operation is more relevant to a project than the Read Modify Write latency.

Our overall best workload for OrientDB is workload D because it has the best overall throughput and insert latency. Even though workload E has a better insert latency, its throughput is too low compared to that of workload D.

VI. CONCLUSION

To conclude, this paper describes the benchmarking process and analyzes the performance of three types of NoSQL data models notably MongoDB for document-oriented databases, OrientDB for graph databases as well as Redis for key-value pair-based databases. The *Yahoo! Cloud Serving Benchmark* or YCSB tool was used in order to perform the benchmarking based on six different kind of workloads. Moreover, tools such as Docker and Docker-compose, a manual scripting as well as different environments such as Amazon EC2 and Ubuntu operating system made this experiment successful. From our knowledge, the key-value pair-based database is supposed to be the quickest data model amongst the three chosen databases. Indeed, our benchmarking experiments and results have proven that Redis has the best performance amongst the other two.

REFERENCES

[1] Raut AB. NOSQL database and its comparison with RDBMS. International Journal of Computational Intelligence Research. 2017;13(7):1645-1651.

[2] W. Ali, M. U. Shafique, M. A. Majeed, and A. Raza, ''Comparison between SQL and NoSQL databases and their relationship with big data analytics,'' Asian J. Res. Comput. Sci., pp. 1–10, Oct. 2019.

[3] A. A. Safaei, Real-time processing of streaming big data, Real-Time System. 2017 53(1):1–44. https://doi.org/10.1007/s11241-016-9257- 0.

[4] LinkedIn Pulse. B. Marr, Big Data: The five Vs Everyone Must Know, 2014.

[5] InFocus Blog | Dell EMC Services. Big Data and NoSQL: The Problem with Relational Databases. www.infocus.emc.com/april_reeve/big-data-and-nosql-the-problem-with-relational-databases/. Accessed August 7, 2017.

[6] J. Pokorny, New Database Architectures: Steps towards Big Data Processing, Proc. IADIS Eur. Conf. Data Min. (ECDM'13), António Palma dos Reis Ajith P. Abraham Eds., IADIS Press. 2013 3–10.

[7] M. Wojcik, "Scaling horizontally vs. scaling vertically," Section, 24-Jul-2020. [Online]. Available: http://www.section.io/blog/scaling-horizontally-vs-vertically/. [Accessed: 08-Dec-2022].

[8] Tech Republic. 10 things you should know about NoSQL databases. http://www.techrepublic.com/blog/10-things/10-things-you-shouldknow-about-nosql-databases/. Accessed August 8, 2017.

[9] J. Bhogal and I. Choksi, Handling Big Data Using NoSQL. in Proceedings - IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2015, 393–398: IEEE.

[10] Ahmed, M. R., Khatun, M. A., Ali, A., and Sundaraj, K. 2018. A literature review on NoSQL database for big data processing. International Journal of Engineering & Technology. 7, 2 (Jun. 2018), 902--906.

[11] Khazaei, Hamzeh & Fokaefs, Marios & Zareian, Saeed & Beigi, Nasim & Ramprasad, Brian & Shtern, Mark & Gaikwad, Purwa & Litoiu, Marin. (2015). How do I choose the right NoSQL solution? A comprehensive theoretical and experimental survey. Journal of Big Data and Information Analytics (BDIA). 2. 10.3934/bdia.2016004.

[12] J.-K. Chen and W.-Z. Lee, "An introduction of NoSQL databases based on their categories and Application Industries," Algorithms, vol. 12, no. 5, p. 106, 2019.

[13] S. Swaminathan and R. Elmasri, Quantitative analysis of scalable nosql databases, 2016 IEEE International Congress on Big Data (BigData Congress), 2016: IEEE.

[14] E. Tang and Y. Fan, Performance Comparison between Five NoSQL Databases, in 2016 seventh International Conference on Cloud Computing and Big Data (CCBD), 2016; 105–109: IEEE.

[15] J. Bhogal and I. Choksi, Handling Big Data Using NoSQL. in Proceedings - IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2015, 393–398: IEEE.

[16] P. Srivastava, S. Goyal, and A. Kumar, Analysis of various NoSql database, 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), 2015: IEEE.

[17] Z. Bicevska and I. Oditis, ''Towards NoSQL-based data warehouse solutions,'' Procedia Comput. Sci., vol. 104, pp. 104–111, Jan. 2017.

[18] C. Li and J. Gu, ''An integration approach of hybrid databases based on SQL in cloud computing environment,'' Softw., Pract. Exper., vol. 49, no. 3, pp. 401–422, Mar. 2019.

[19] D., & Johnson, J. (2020, December 9). CAP theorem for databases: Consistency, Availability & Partition Tolerance. BMC Blogs. Retrieved December 8, 2022, from https://www.bmc.com/blogs/cap-theorem

[20] "CAP theorem and OrientDB," *Google*. [Online]. Available: https://groups.google.com/g/orient-database/c/scktCwFBkPQ. [Accessed: 05-Dec-2022].

[3.1] M. Papiernik, "An introduction to document-oriented databases," *DigitalOcean*, 20-Jul-2021. [Online]. Available: https://www.digitalocean.com/community/conceptual-articles/an-introduction-to-document-oriented-databases. [Accessed: 26-Nov-2022].

[3.2] "Document databases make storing data easier," *InfluxData*, 16-Nov-2022. [Online]. Available: https://www.influxdata.com/document-database/. [Accessed: 26-Nov-2022].

[3.3] A. Williams, "Document database {definition, features, use cases}," *Knowledge Base by phoenixNAP*, 09-Jun-2022. [Online]. Available: https://phoenixnap.com/kb/document-database. [Accessed: 26-Nov-2022].

[3.4] "Graph database: How it works, its uses & benefits," *InfluxData*, 29-Nov-2022. [Online]. Available: https://www.influxdata.com/graph-database/. [Accessed: 26-Nov-2022].

[3.5] "Overview of orientdb," *Overview of OrientDB · OrientDB Manual*. [Online]. Available: https://orientdb.org/docs/3.0.x/misc/Overview.html. [Accessed: 26-Nov-2022].

[3.6] Diogo Fernandes Polytechnic of Coimbra - ISEC, D. Fernandes, P. of C.-ISEC, Jorge Bernardino Polytechnic of Coimbra - ISEC, J. Bernardino, H. Niederrhein, and O. M. V. A. Metrics, "Graph databases comparison: Allegrograph, arangodb, InfiniteGraph, neo4j, and orientdb: Proceedings of the 7th International Conference on Data Science, Technology and Applications," *Guide Proceedings*, 01-Jul-2018. [Online]. Available: https://dl.acm.org/doi/10.5220/0006910203730380. [Accessed: 26-Nov-2022].

[25] A. N. D. R. E. E. D. U. A. R. D. O. MENEGASSO, "NOSQL," Amazon, 2018. [Online]. Available: http://www.aws.amazon.com/nosql/key-value/. [Accessed: 08-Dec-2022].

[26] "What is a key-value database?," Redis, 15-Jul-2021. [Online]. Available: http://www.redis.com/nosql/key-value-databases/. [Accessed: 08-Dec-2022].

[27] A. Williams, "NoSQL database types explained: Key-Value Store: TechTarget," Data Management, 17-Aug-2021. [Online]. Available:

http://www.techtarget.com/searchdatamanagement/tip/NoSQL-database-types-explained-Key-value-store. [Accessed: 08-Dec-2022].

[28] Getting started. UVA Research Computing. (n.d.). Retrieved December 8, 2022, from http://www.rc.virginia.edu/userinfo/howtos/general/redis/

[29] Patel, H. (2020, September 29). Redis - what and why? Medium. Retrieved December 8, 2022, from http://www.medium.com/weekly-webtips/redis-what-and-why-pros-cons-ae2f5bc750f

[30] "Benchmarking definition - What is Benchmarking?," *Shopify*. [Online]. Available: https://www.shopify.com/ca/encyclopedia/benchmarking. [Accessed: 07-Dec-2022].

[31] "The Ultimate YCSB Benchmark Guide (2021)," *Benchant.com*. [Online]. Available: https://benchant.com/blog/ycsb. [Accessed: 03-Dec-2022].

[32] G. Patterson, "Docker," *Amazon*, 1980. [Online]. Available: https://aws.amazon.com/docker/. [Accessed: 03-Dec-2022].

[33] "Overview," *Docker Documentation*, 03-Dec-2022. [Online]. Available: https://docs.docker.com/compose/. [Accessed: 03-Dec-2022].