Practical Course: Cloud Data Bases

YACOUBA CISSE, 03698456
LUCA CORBUCCI, 03726968
FABIAN DANISCH, 03683295

Abstract: A $< key, value >$ database is a non-relational database that allows storing a unique $< key, value >$ pair, where the key is used as a unique identifier. In this report we will explain how we developed a distributed $< key, value >$ database using Java. We will also inspect the design choices we had to face during our implementation and analyze the performance of our system.

1 INTRODUCTION

When talking about Data, the subjects Big Data and Big Data Analysis are always mentioned. However, people sometimes forget that this also entails a database that is capable of storing big quantities of data. The application areas for a database are infinite. For example, they are used to store data in back end systems or collect data from IoT sensors. Databases are even used to analyze data.

The rise of databases started in 1970 with relational systems [1] and in 2009 NoSQL [2] systems were developed. When developing something and a database is being deployed, it is crucial to deeply understand what data is used and how it should be stored to achieve the most optimal performance. This information affects the choice of the type of database that is going to be deployed.

In this report, we will explain how we developed a distributed $< key, value >$ database and we will analyze the performance of our software.

The motivation that led us to develop a distributed $< key, value >$ database is that we wanted to deeply understand how a system like this works and which are the main pro and cons of using it.

A distributed $< key, value >$ database is a NoSQL database where we store a collection of $< key, value >$ pairs. In this database the key of each pair is unique and we allow the following operations:

- Insertion of a new pair
- Deletion of a pair
- Update of the value corresponding to a key
- Lookup of a pair

As we said, this database is distributed. This means that we have a network with multiple servers. Each server is responsible for a subset of keys. The data from each key-range is stored on the responsible server, whilst 2 additional servers are used to store the data-replicas to provide data-availability for the users.

The main features of a distributed $< key, value >$ storage are the following:

- High performance: these systems allow to store the data on disk and in a cache. This is an important feature if the user does a lot of lookups in the database.
- Scalability: these systems are distributed which allow the client to balance the load of each server and to distribute data around the servers of the network.
- Simple to use: it is not necessary to describe a schema for this database, the client can only store $< key, value >$ pairs, which is very convenient in some use cases.

2 RELATED WORKS

The most known $< key, value >$ database are the following:

- Cassandra [3]: offers replication of data and uses a simple query language. It has a horizontal scaling system that increases the performance.
- Redis [4]: offers a lot of possible data structures to store data.
- Aerospike [5]: offers high performance, scalability, and simplicity for all the operations.
- Amazon DynamoDB [6]: offers high performance and scalability. It is developed by Amazon and has great integration with the AWS services.

3 USE CASES

During the development of our project, we thought about some possible use cases for our distributed database. Our goal was to think about a use case that could exploit all the best features of a $< key, value >$ database.

We thought that a Leaderboard System for a multiplayer game could be a good use case for our system because of the following reasons:

- We can quickly add or update the score of a user;
- We can quickly retrieve a score of a specific user from the database giving up the consistency of the data on the servers. For this use case in our opinion is better to have the velocity property than the consistency.

After thinking about the possible use case we decided to develop all our extensions to the project based on the necessities of a leaderboard system. We will talk further about our extension in Section 5.

4 IMPLEMENTATION CHOICES

4.1 File Storage

One of the first things that we implemented was the persistence of the $< key, value >$ pairs on the disk. To implement this feature we wanted to use a BTree [7] or a B+Tree but we did not find any implementation of this data structure in the Java basic libraries. We did not have enough time to implement and test one of these data structures based on the description that we found on the paper so we decided to use another solution.

We were inspired by the idea of consistent hashing [8] and we used two data structures to implement the persistence of the $< key, value >$ pairs on disk:

- To store the data on disk, we used a HashMap [9]. This data structure allows us to add and retrieve data on average in

$O(1)$. HashMap implements Serializable and this was really useful in our case because we needed to store the data on disk. The only problem was to read from the HashMap on the disk. We thought that when more and more pairs are added into the HashMap, the performance would become too slow, so we decided to put a limit on the number of keys contained in each HashMap. When a HashMap is full, we split it into two HashMaps to balance the amount of data that we need to load in the memory when we need to access the HashMap and also to avoid collisions.

- To deal with this "rebalancing" procedure, we use a TreeMap [10] to store the last key of each HashMap. When we want to execute an insertion of a new key or a lookup, we can easily find the responsible HashMap, with which we are then able to insert or extract the key from this structure without wasting too much time to load the data in the memory.

4.2 Data Replication

There are three main properties that a user would like to have in a distributed system:

- Consistency: in our case, this means that on each node we always have the most recent value for each key;
- Availability: in our case, this means that the user always receives a reply from the server when he/she asks for a key;
- Partition Tolerance: the damage of a part of the network does not affect the rest of the network.

Unfortunately, as stated in the Cap Theorem [11], it is impossible to have more than 2 of these features at the same time in a distributed system. When we develop a distributed system we should always understand what the final purpose of the system is. With that in mind, we choose the two most suitable properties that we have just listed.

In our distributed system we decided to have Availability and Partition Tolerance. Due to that, it could happen that if a user wanted to retrieve the value of a specific key, he would get an old value and not the most recent one. More specifically we do the following things when we need to change or add a key:

- When we add a new key, it is stored on the master server and it is immediately forwarded to two slave servers (the servers that follow the master in the ring of the consistent hashing);
- When we update the value of a key we make use of the Eventual Consistency [12] and wait a certain amount of time before sending the updated data to our successor. The idea is to wait because in that period of time we could receive other update requests. Doing so, we can order more update requests by only sending a single message to our successor.

We implemented the replication of the data using recursion:

- When a pair is added or updated on the master server, it sends a message to its successor on the ring for each key.
- When the successor receives the data, it stores the pair and then forwards it to its successor based on the information contained in the received message.

We chose to implement it this way because we can reduce the workload on the master. He only has to send a single message to its successor, then the message will be spread in the network using a "peer-to-peer" strategy.

This implementation allows us to easily extend the original project specification, adding more than 2 replicas of our data.

5 OUR EXTENSION TO THE ORIGINAL PROJECT

In section 3 we mentioned our extensions to the project, here we want to explain them more deeply.

5.1 Renaming a Key

One of our extensions is to offer users the possibility to rename the key of a $< key, value >$ pair. In order to implement this functionality, there were some hindrances we had to face.

- Due to the consistency of our system, we always store replicas of our data on 2 additional servers. So if we rename a given key of a $< key, value >$ pair, we also have to make sure that our replicas will get informed of the changes.
- Another obstacle whilst renaming is the responsibility of the servers. Every server is responsible for a different key-range. After renaming, we could face the case, that the responsible server changes. Due to that, we have to transfer the $< key, value >$ pair to the new responsible server. This obviously also affects the replicas from the old server and the new one.
- The last hindrance was the subscription to a $< key, value >$ pair. When renaming a key we also have to rename the corresponding subscription file. If the key has to change the server, then the subscription data gets removed on the old server and replicas but also has to be transferred, to the new responsible server and its replication server.

5.2 Subscription system for $< key, value >$ updates

Clients that want to stay updated about the value of a certain key face the problem of having to send the same get-commands over and over again. Therefore, we implemented a subscription system notifying the client about updates on certain keys. The user can now choose for which keys he wants to receive updates by sending a subscribe-command. The servers store the information about which client subscribed to which key. If the user chooses to not receive updates on a key anymore, he/she can do so by sending an unsubscribe-command.

Every time a put-update is sent to the server by an arbitrary client, it will notify the subscribers with the new value of the key. Clients will also get notified when a key that they're subscribed to gets deleted, and the server automatically unsubscribes all the clients from said key.

The update-communication takes place via another connection between client and server. Here, the client listens to incoming requests on port 6969 and prints out subscription changes on the user's console. As the server stores the IP address of the subscribed client, the subscription will be upheld if a client disconnects from the network under the assumption that its address doesn't change.

In case of a server shutting down, the subscription data will be sent to its successor, which is now responsible for the $< key, value >$

pairs. Also, if a new server joins the network, it will receive the subscription data to the corresponding keys from its successor.

For our use case, the subscription can be useful for tracking the highscores of your best friends and worst rivals.

5.3 Protect a key by Password

Based on the "Leaderboard" use case we thought that a user could be interested in hiding his/her score and in maybe only sharing it with certain users. That's why we decided to develop a password protection feature for the $< key, value >$ database.

Our goal with this extension was to extend the original CRUD API without overturning the user experience and adding a lot of new commands. From the user perspective, this feature is only visible when he/she wants to add a new key to the database because he/she can choose between a standard put command or a put command protected via password. For all the other operations the user will continue to use the original commands and the system will ask for the password when he/she tries to interact with a protected $< key, value >$ store.

In the end, we were able to develop this feature by adding only one more command.

6 PERFORMANCE ANALYSIS

When we finished developing the distributed database, we ran a couple of benchmark tests to understand the performances of our software.

We tested our project on a laptop, its configuration is shown in Table 1

Table 1. Detailing System Configuration

Processor	Intel Core i7-6820HQ
Clock Speed	2.70GHz
Number of Cores	4
Number of Threads	8
RAM	16Gb
Java Version	13.0.1
Operating System	macOS Catalina 10.15.2

When we tested our project, we considered the fact that we can run at most 4 threads concurrently and with more than 4 threads we start using hyper-threading.

All the tests were executed 5 times and we took the average of the results that we obtained so that we could have a more reliable final result.

6.1 Testing our system with variable numbers of Servers or Clients

The first idea that we had for testing our distributed database was to change the number of servers or the number of the clients in the network and performing the following operations:

- Insertion of 50.000 $< key, value >$ pairs
- Retrieval of 50.000 $< key, value >$ pairs

In the first test, we changed the number of active servers. We tried the following combinations:

- 1 Client 1 Server
- 1 Client 2 Servers
- 1 Client 4 Servers
- 1 Client 8 Servers
- 1 Client 16 Servers

The result of this test is visible in Figure 1, the completion time for the insertion decreases with an increasing number of servers, in this case, the real bottleneck is the single client.

The completion time for the retrieval of 50.000 keys, in this case, has a spike when we increase the number of servers to 2 and then decreases again after that. This curve shape is probably due to the fact that when we add the second server, the client may try to contact a server which is not responsible for that key, so that the client has to reconnect to the right one.

This assumption is confirmed if we consider that when we have 4 servers and 1 clients, we obtain an improvement in terms of completion time because having more than 3 servers activates the replication and that way the clients don't always need to redirect the query to the responsible server. When we add more and more servers the completion time increases again because the replication is only taking place on 3 servers which is why there is a higher chance that the client has to forward the query to the responsible server.

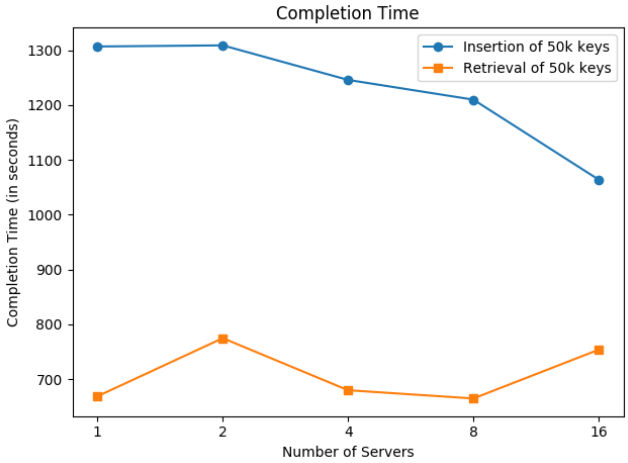

Fig. 1. Insertion and Retrieval of 50.000 keys using 1 server and a variable number of Servers.

We computed a similar test changing the number of clients, while the number of servers remains at 1. We tried the following combinations:

- 1 Client 1 Server
- 2 Clients 1 Server
- 4 Clients 1 Server
- 8 Clients 1 Server
- 16 Clients 1 Server

We can see in Figure 2 that the completion time for the insertion of 50.000 keys is slowly decreasing when we add more servers. The problem here is that, even if we add more clients, the server

that handles the requests is always 1 and so we don't have a big improvement adding more clients.

The completion time for the retrieval of 50.000 keys is more interesting because we have a different behavior compared with the one that can be seen in Figure 1. In this case, the completion time for the retrieval process increases when we add more clients. This is probably related to the fact that the single server has to handle a lot of requests in parallel.

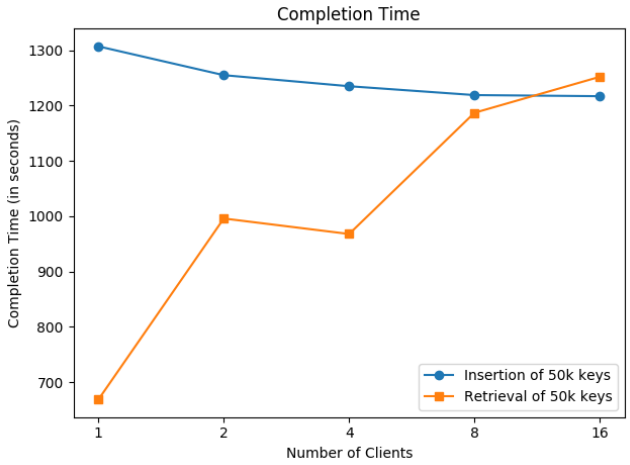

Fig. 2. Insertion and Retrieval of 50.000 keys using 1 server and a variable number of Clients.

6.2 Testing our system with different numbers of Clients and Servers

In this test, we evaluated our system while adding new clients and servers in each iteration to understand the scalability and the speedup of the distributed database. We tried the following combinations of servers and clients:

- 1 Server 1 Client
- 2 Servers 2 Clients
- 4 Servers 4 Clients
- 8 Servers 8 Clients
- 16 Servers 16 Clients

The result of this test is shown in Figure 3, we can see that the completion time for the insertion of 50.000 keys decreases when we add more servers and more clients and it increases only when we have 16 threads for the clients and 16 for the servers, this behavior is probably due to the fact that our testing machine does not have enough cores to run all these threads in parallel. The curve of the completion time for the retrieval of 50.000 keys has a similar behaviour of the curve that we showed in Figure 1. The reason is the same and is related to the replication of the keys on the servers in the network.

6.3 Testing different parameters for cache management

We wanted to understand how the cache replacement policy and the cache size can influence the completion time of our distributed

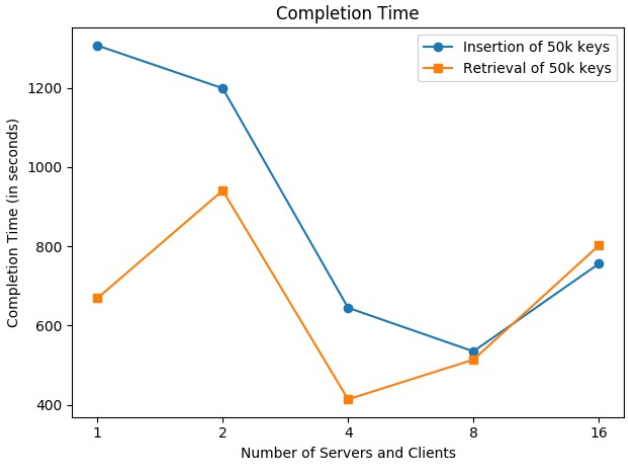

Fig. 3. Insertion and Retrieval of 50.000 keys using 1 server and a variable number of Servers and Clients.

system. This test is a bit different from the others. In this case, we populated our database with 50.000 keys and we did the retrieval of 1000 random keys for 10 times starting from a different key at each iteration. In this data set we have 800 distinct keys and 20 keys that are repeated 10 times.

In figure 4 we show the result of this experiment.

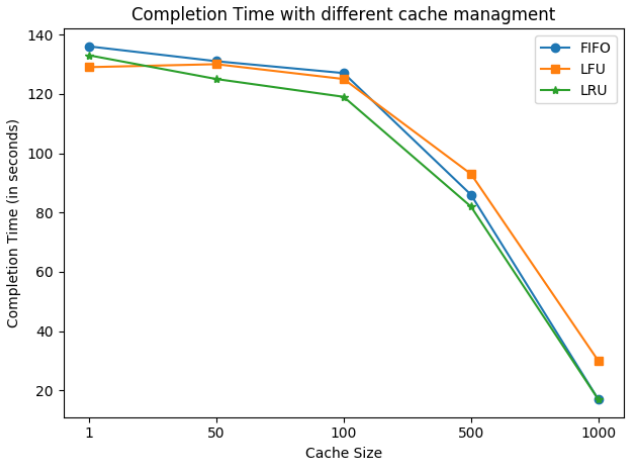

Fig. 4. Test of different cache size and replacement policy

We tried the 3 different cache replacement policies that we implemented (FIFO, LFU, LRU) and for each of these, we tried different cache sizes starting from 1 key (for each retrieval an access to disk) to 1000 keys (everything in the cache with just a single disk access). We can see that the completion time decrease when we increase the cache size because we don't need to access the disk as often as we did before. We couldn't make out visible differences between the different cache replacement policies using this data set.

6.4 Testing our system in a real world

We have to consider two aspects when we do a performance analysis in a distributed system:

- The cost of disk access
- The communication cost

In our previous benchmarks, the communication cost was negligible with respect to the cost of disk access because we were running the clients and the servers on the same machine. In a real-life distributed database we have a different situation because the communication happens through the network and the communication cost is higher than the cost of disk access and for this reason, it becomes the real bottleneck.

We ran our distributed database on a VPS to test how much the communication cost had an impact on the performance of our system. The VPS has the configuration shown in Table 2.

Table 2. Detailing System Configuration of our VPS

Processor	Intel Xeon Processor
Clock Speed	2GHz
Number of Cores	2
Number of Threads	2
RAM	4 Gb
Java Version	11.0.5
Operating System	Ubuntu 18.04.3 LTS

For this benchmark we were forced to decrease the number of keys because of the VPS's performance. We ran two tests to compare how much the communication cost impacts the completion time:

- Insertion of 5000 keys, Retrieval of 5000 keys (same order of insertion) with Client, ECS and Server on the VPS
- Insertion of 5000 keys, Retrieval of 5000 keys (same order of insertion) with Client on our local machine, ECS and Server on the VPS

The result of this benchmark is shown in the Figure 5. The blue bar is the completion time of our test run entirely on a single machine with the configuration shown in Table 2. The orange bar is the completion time of the test run wiwth the client on a local machine with the configuration shown in Table 1 with the Server and the ECS on the VPS. The difference between the two completion times is visible and we can say that the real bottleneck in a system like the one we developed is the communication cost over the network.

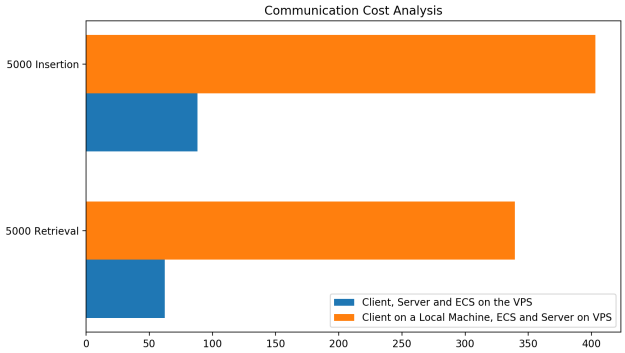

Fig. 5. Comparison between a test run on a VPS and a test run on a VPS with the client on a local machine

7 TEST REPORT

During the development of this project, we discovered the importance of writing good tests for our methods and for our classes. We tried, as far as possible, to adopt a Test Driven Development approach [13] and this allowed us to write our code considering all the possible failures and strange behavior of our application.

We tested our project on a MacBook Pro with the configuration written in this Table 1 and in a docker container with Ubuntu 18.04.3. To run the tests we used Maven and more specifically the command *mvn test*. All the tests were executed without any error.

8 CONCLUSIONS

8.1 Future Works

When we developed Milestone 5 we thought about a lot of possible extensions but in the end, we did not implement them because of the lack of time. Some of them are related to our use case and some of them are designed to add more features to the project and to extend it to other possible use cases. Even if we did not implement them we want to list them here as "Future Works" that we could develop in our spare time:

- When we use the leaderboard we could be interested to see the rank of the best users. Our idea was to implement a sort of Map Reduce [14] where each of our servers computes the N best users (Map Phase) and then sends them to one server that computes the K best users (Reduce Phase) and then sends them to the client.
- To use our distributed database for other use cases we wanted to add the 2 Phase Commit during the update process of a key to obtain the Consistency property at the expense of losing the availability. We thought that in some use cases (for example a finance context) the consistency should be more valuable than the availability.
- In our system, we have a single point of failure, it is the ECS, if it is down it is impossible to join the network. One of our ideas was to add more than one ECS to avoid the single point of failure problem.

8.2 What we Learned

During the development of this project, we had the opportunity to solve a lot of challenges and we also had the feeling to work like in a real company. One of the most challenging problems we dealt with was the balance of the work and the shared work. Using git we were able to understand how to work remotely on this project, which was very informative with a view to our working career. We learned how to use JUnit and how to write tests for our code. This was really useful because thanks to this we were able to find bugs in our code and to erase them.

The last thing that we learned is that in a big project like this we should have considered the possible future extensions since the beginning and we should have developed the milestones with consideration to them. If we had considered these possible extensions during the entire development of the project we would have had less trouble developing the final milestone. This is an important and useful takeaway message that we will never forget.

REFERENCES

[1] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.
[2] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley Professional, 1st edition, 2012.
[3] Cassandra - http://cassandra.apache.org.
[4] Redis - https://redis.io.
[5] Aerospike - https://www.aerospike.com.
[6] Dynamodb - https://aws.amazon.com/dynamodb/.
[7] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.
[8] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.
[9] Hashmap - https://docs.oracle.com/javase/8/docs/api/java/util/hashmap.html.
[10] Treemap - https://docs.oracle.com/javase/8/docs/api/java/util/treemap.html.
[11] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
[12] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.
[13] Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., USA, 2002.
[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.