Development of a NoSQL database with a client-server model

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor thesis
Bachelor of Engineering, Information and Communication Technology.
Spring semester 2022
Sergio Becerra Flores

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and Communication Technology.		Abstract
Author	Sergio Becerra Flores	Year 2022
Subject	Development of a NoSQL database with a client-server model	
Supervisors	Petri Kuittinen	

This thesis deals with different topics, such as the differences between SQL and NoSQL databases and the main keys of the architecture used.

For this thesis I have chosen to usee a non-relatonal database, NoSQL, so I can modify everything I wanted from start to end, this database is focused in a layer system, where the first layer is the client-server distribution, this is where the communication between the clients is produced, then there is the document store layer, this will be used to store the different types of documments inside the database, and in the lowest level layer will be the Key-Value store, whch will be the layer in charge of storaging the information received on the upper layers. Using this type of organization simplifies a lot the architecture of database.

In order to keep data consistant, the database uses a Write-Ahead Log which supervises that every data introduced in the database remains safe and also that every single user has the same information.

For the communication between the client and the server I have chosen passive replication, this type of communication ensures that the data there will be no lost-data in the communication process.

This thesis has been mainly inspired because of multiple subjects in my home university such as Cloud Computing and Distributed Storage and Processing Systems and the result has been pretty much what I was looking for, even though I still have some more ideas about that I want to implement in a future.

Keywords Database; NoSQL; Client Server; Write Ahead Log; Replication.
Pages 42 pages and appendices 2 pages

Index

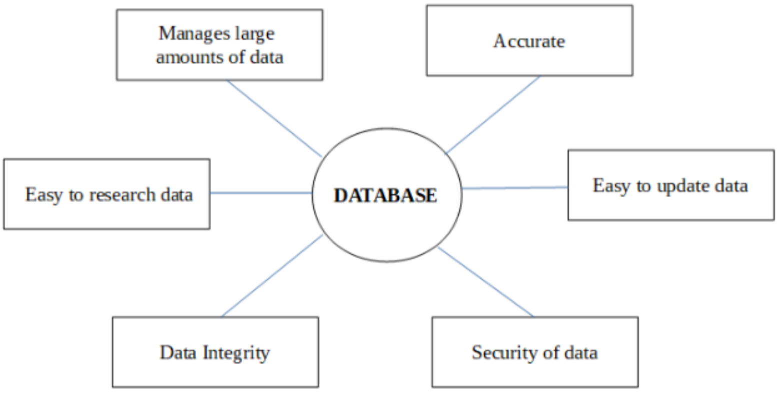
1. Introduction.

Nowadays, data flow is one of the most important parts of our technology, whether it is sending an email or making a payment using a bank application, everything requires a database that can support the workload, as Chris Smith (2019) comments in his article: "Why databases are so important in our lives", everything uses a database behind it.

In addition, this data flow increases every day, so the load is greater, and that is why it is necessary to improve our databases, since not only has the size of the data we use been growing, but also the amount of data we send and receive every day. (Khvoynitskaya, 2020)

That is why using technologies such as Cloud Computing is becoming more and more common and necessary, since they allow us to host a huge amount of data in a very simple way and will end up being used by most companies in the not-too-distant future.

A good database has to be able not only to store the data at a high speed, but also to be able to display it at a high speed, also one of the most important parts of a good database, probably the most important one is to ensure that the data will not be compromised, that is, it will not be accessible by anyone we do not want and will not be lost at any time, such as the great information leak that occurred in 2019 by the Facebook company, which affected 533 million users filtering a total of 146 gigabytes of information. (Tyas, 2022).



Lastly, one of the most important characteristics that a database must have is that it must be easily updatable, since today we live in an environment that fluctuates daily, and this implies that a database that today is the latest technology tomorrow can be completely obsolete.

This thesis tries to create a reliable, fast and easily accessible database so that multiple users can store their data within it, using a NoSQL relational database, with a client-server model to be able to connect to multiple users.

In addition, it ensures the integrity of this using a Write-Ahead Log (WAL) protocol that can recover any lost data the moment it is lost.

2. Differences between SQL and NoSQL databases.

What are the differences between these 2 types of databases, and which one should we use for every case ?

Those are the questions that you should make to yourself if you are thinking about creating a database.

Just to see a little bit of history, relational databases started being used in the 80´s whereas the non-relational databases which started being used around 2012 (Bartholomew, 2010), until this day, relational databases are still the most popular type of database, but it doesn´t mean that it is the best option for every situation as I will now explain.

2.1. Relational databases.

Relational databases are a collection of data elements organized into a set of formally described tables, from which the data can be accessed and reassembled in many different ways without having to reorganize the database tables. The standard user and application program interface to a relational database is Structured Query Language (SQL) (Silva, Almeida & Queirozn, 2016). SQL commands are used both for interactive queries and for getting information from a relational database and collecting data for reports.

They are based on the organization of information in small parts that are integrated by means of identifiers; unlike non-relational databases which, as the name implies, do not have an identifier that serves to relate two or more data sets. They are also more robust, that is, they have a greater storage capacity, and are less vulnerable to failures, these are their main characteristics. (Hammes, Medero, & Mitchell, 2014)

Other relational databases that are commonly used nowadays are:

Oracle:

Oracle is mainly used for companies to administrate high loads of data, which makes employees being able to focus on work operations and be more efficient (Lahiri, Chavan, Colgan, Das, Ganesh, Gleeson, & Zait, 2015)

IBM DB2:

IBM DB2 is able to prevent unauthorized access, provides utilities for data backup and recovery, and offers performance tools and data management capabilities. (Haderle & Jackson, 1984)

And of course, the most used one is **MySQL**:

The most common advantages and disadvantages are (Denton & Peace, 2003):

Advantages:

MySQL is free and open to use.

Easy to use and install.

Low cost in requirements for the preparation and execution of the program.

Speed when performing operations and good performance.

Low probability of data corruption.

Environment with security and encryption.

Disadvantages:

Being Free Software, many of the solutions for the deficiencies of the software are not documented or present official documentation.

Application performance should be controlled/monitored for failures.

It is not the most intuitive of the programs that currently exist for all types of developments.

It is not as effective in applications that require constant write modification to the DB, due to the data management.

2.2. Non-relational databases.

Non-relational databases are specifically designed for specific data models and have flexible schemas for building modern applications. They are widely recognized because they are easy to develop, both in functionality and performance at scale (Bhat & Jadhav, 2010). They use a variety of data models, including document, graph, key-value, in-memory, and lookup.

This type of databases are those that, unlike relational databases, do not have an identifier that serves as a relationship between one set of data and others. As we will see, the information is normally organized through documents, and it is very useful when we do not have an exact scheme of what is going to be stored.

This way of storing information offers certain advantages over relational models. Among the most significant advantages the most important ones are:

1. They run on machines with few resources: These systems, unlike SQL-based systems, require hardly any computation, so they can be set up on less expensive machines.
2. Horizontal scalability: To improve the performance of these systems, it is simply achieved by adding more nodes, with the sole operation of indicating to the system which nodes are available.
3. They can handle a large amount of data: This is because it uses a distributed structure, in many cases through Hash tables.
4. It does not generate bottlenecks: The main problem with SQL systems is that they need to transcribe each statement in order to be executed, and each complex statement also requires an even more complex execution level, which constitutes a common entry point, that before many requests can slow down the system.

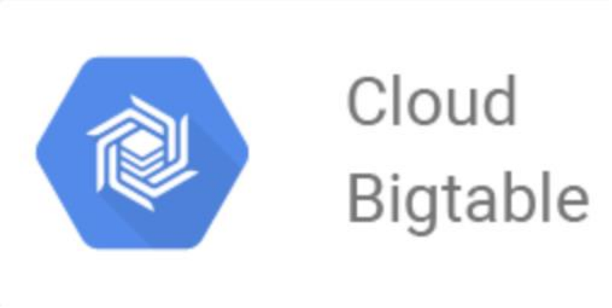
Some of the most common non-relational databases used nowadays are:

Cassandra:

Cassandra's main goal is to be able to manage a large data load across multiple nodes. Cassandra replicates and distributes the information from the first moment through all its nodes. (Cassandra, 2014)

This works in a similar way as my thesis does and has been an inspiration for some of my approaches to this project.

BigTable:



BigTable is mainly used for storaging large amounts of data with a key-data structure and very low latency.
It has great scalability, easy administration and the capability of changing the cluster size without downtime. (Chang, F., Dean, J., Ghemawat, Hsieh, Wallach, Burrows & Gruber, 2008)

HBase:

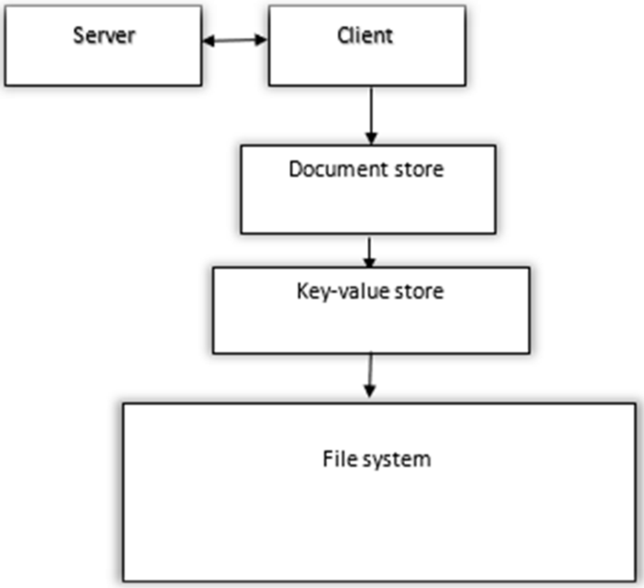
Apache HBase is mainly used to have real-time access to big amounts of data, it runs on top of Hadoop Distributed File System (HDFS) and provides a fault-tolerant way to store data sets. (Mehul, 2011)

3. Architecture used in the system.

The architecture is composed of 4 layers:

- The files that contain all the data necessary for the key-value store.
- The key-value store will be in charge of storing the information in memory.
- The document store that organizes the information for your clients and allows any type of document to be stored in the database.
- Lastly, the server-client communication is the key for multiple users accessing the same database in real time.

Architecture layers



The key-value store will be the layer which will be storing as fast as possible in the database, while the document store will rely on the functionalities of the key-value store to support any type of file that the user wants to enter.

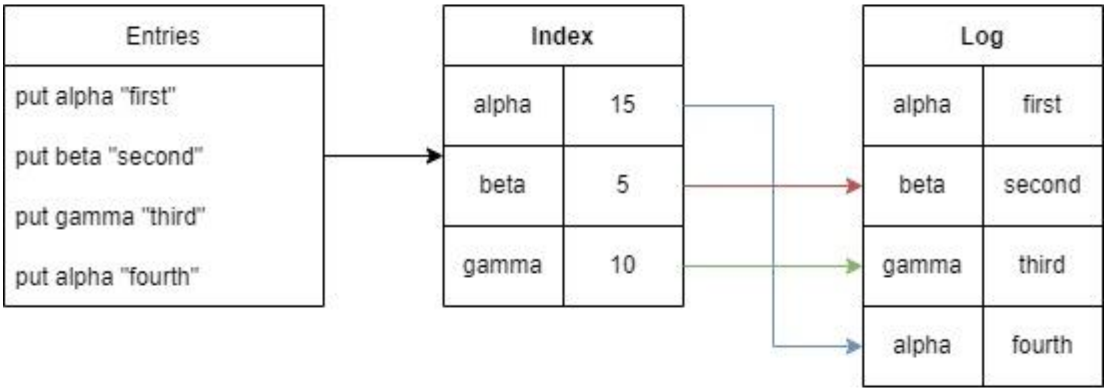
On the other hand, the Server-Client communication will allow interconnectivity among the users of the database, ensuring that the data they are using or storing is reliable by using the Write-Ahead Logging system.

3.1. Key-value store

This store will be based on logs and segments and presents a dictionary-like API to its clients.

To speed up the process of reading and writing the database, I have implemented an in-memory index that points to the key-value pair. In this index the key is registered with the position of the file in which the key-value pair is registered.

Key-Value Storaging



This way the indexing is faster and simpler, however, this method could lead to a log file that is too large and would not be useful for a quick search, so I have opted for using a segmentation method.

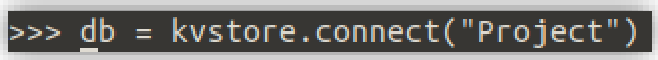
Each time segmentation occurs, a new segment linked to the previous one will be created, creating a chain of segments. Each segment will be made up of an index and a log file, and each time it is written to the database, it will be written on this new segment created, leaving the previous segment archived for reading, that reading will be carried out by searching the segments from most recent to oldest, until reviewing the last one.

This is the list of commands that the API will present:

Connect(Path, opts={})

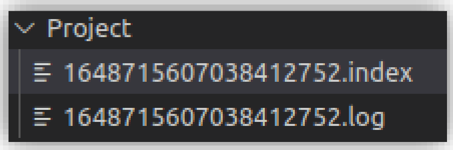
- Produces the connection against a database.
- With **Path** we indicate to the database that we want to connect
- With **opts** we can include additional options

We can either create or connect to the database “Project” with this command:



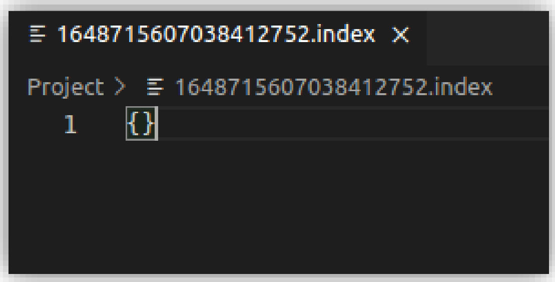
```
>>> db = kvstore.connect("Project")
```

An index and an empty log will be created, whose name is automatically generated using the time.time_ns() command that counts the seconds that have elapsed since January 1, 1970, 00: 00:00 (UTC).



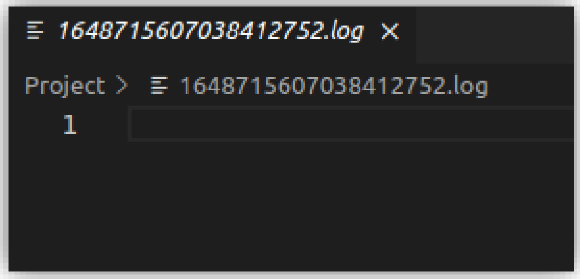
```
Project
  1648715607038412752.index
  1648715607038412752.log
```

Empty index that will later be filled with data.



```
1648715607038412752.index
Project > 1648715607038412752.index
{}
```

Empty log that will later be filled with data:



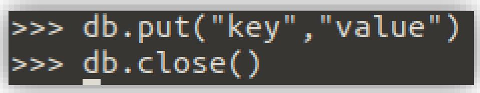
```
1648715607038412752.log
Project > 1648715607038412752.log
```

Once the database is created, we can start introducing data into it, we will now see the put() command to do that.

Put (key, value)

- Creates a new entry on the database associating the **key** with the **value** introduced.

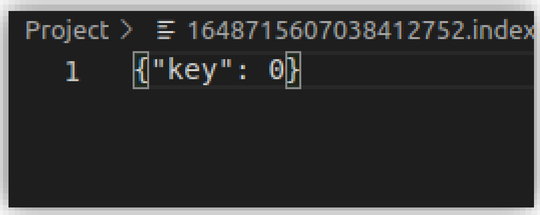
Using the "put" command alone will not update the database index, because it is loaded into memory and does not need to be saved until the connection is closed for later use, so using the close() command that we will see later will be necessary.



```
>>> db.put("key","value")
>>> db.close()
```

Entering a new key-value pair updates our database.

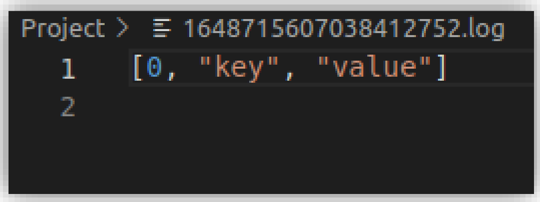
Index:



Project > 1648715607038412752.index

```
{"key": 0}
```

Log:



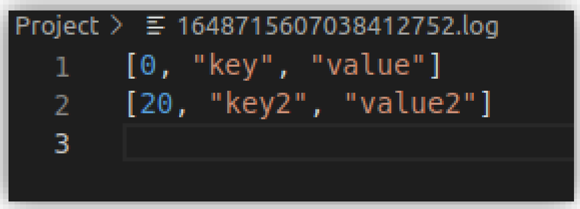
Project > 1648715607038412752.log

```
[0, "key", "value"]
```

As we can see, the information entered with the put() command is now reflected in the log and has the number 0 associated with the index to know where the line begins and to be able to index it easily.
This number is generated by counting the number of characters before this line, since there was nothing before, the number of characters is 0.

For example, if I enter another value, the number associated with the second entry will be 20, since there are a total of 20 characters in the first entry.

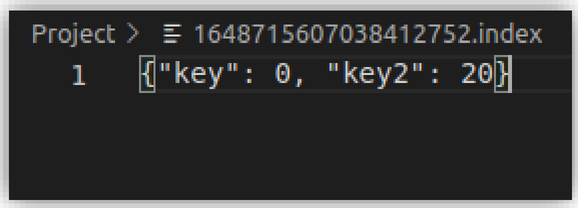
Log



Project > 1648715607038412752.log

```
[0, "key", "value"]
[20, "key2", "value2"]
```

Index



Project > 1648715607038412752.index

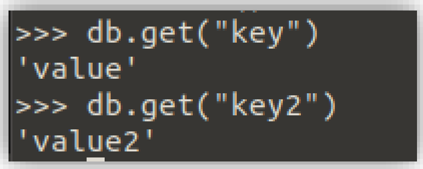
```
{"key": 0, "key2": 20}
```

Get(key)

- Returns the **value** associated with the desired **key**

For the use of the **get()** command, it searches the index for the first word that matches the entered **key** and returns its **value**.

Example:

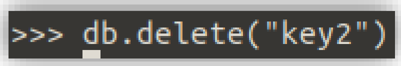


```
>>> db.get("key")
'value'
>>> db.get("key2")
'value2'
```

Delete(Key)

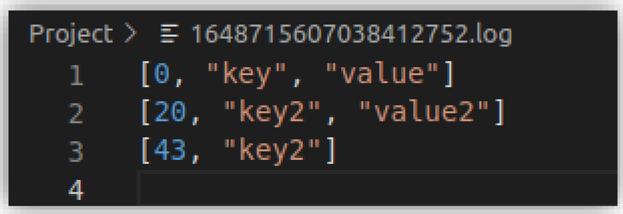
- Removes the key-value pair associated with the given **key**

When we delete, for example, “key2” which was previously inserted in the database with this command:



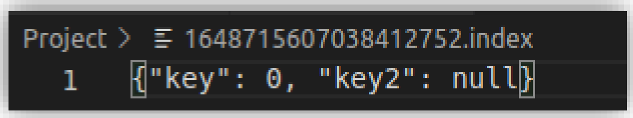
```
>>> db.delete("key2")
```

The log will be displayed like this:



```
[0, "key", "value"]
[20, "key2", "value2"]
[43, "key2"]
```

And the index will have a null pointer so we can no longer look for it with the **get()** command.

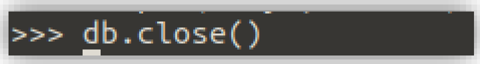


```
{"key": 0, "key2": null}
```

Close()

- Closes the connection against the database.

As I said before, this command is necessary for the index to refresh, it has to be used before closing any connection to ensure that no data will be lost



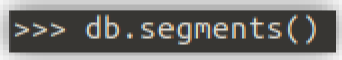
```
>>> db.close()
```

This will be automatically used by the document storaging commands later on.

Segments()

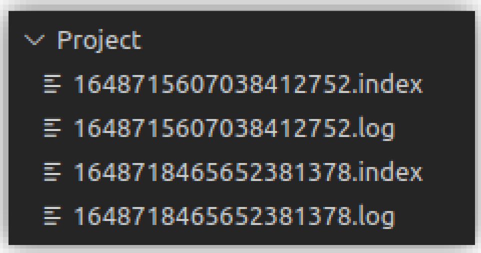
- Start a new segment in the database

When the data starts to be massive, the log and the index become massive too, this will lead to longer times t oread and write into the database, the solution for this is the segmentation, which creates a new log and index so the data can be better distributed into the database, eliminating to entries that we no longer use, such as duplicated keys or deleted ones.



```
>>> db.segments()
```

A new index and log are created:



```
Project
  1648715607038412752.index
  1648715607038412752.log
  1648718465652381378.index
  1648718465652381378.log
```

This new index and log are now empty but will be the one we will be writing on now.

Having multiple logins and indexes in a database is a complicate task, since I want to have them located in memory so the process is faster, this means that when the database starts running it has to load every single segment and this can be a really slow process when the database is very fragmented.

To solve this problem, we can use the next command compacts(), which will create a single version of all of our segments with no repeated data, keeping it clean and ready to use for a fast usage.

Compacts()

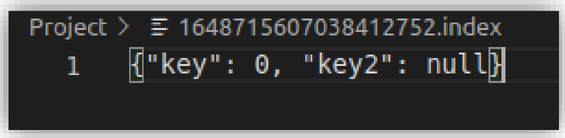
- Compacts the previously created segments

For the final part, the compact method will be the one that we will use to keep our database clean. Instead of having lots of segmented parts, we can just compact them with this method and no duplicated information will be in the compacted version of the database.

For example:

This is the **first** index and log we created:

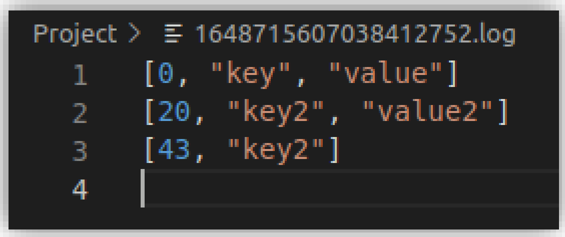
Index:



Project > 1648715607038412752.index

```
{"key": 0, "key2": null}
```

Log:

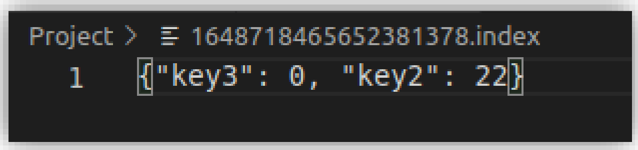


Project > 1648715607038412752.log

```
[0, "key", "value"]
[20, "key2", "value2"]
[43, "key2"]

```

And this is the **new one** on the new segment:

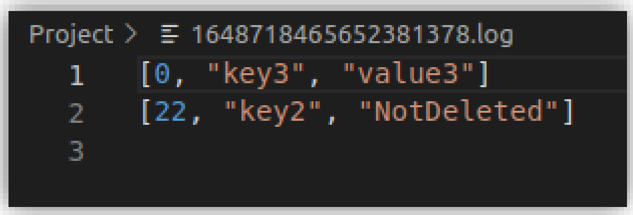
Index:



Project > 1648718465652381378.index

```
{"key3": 0, "key2": 22}
```

Log:



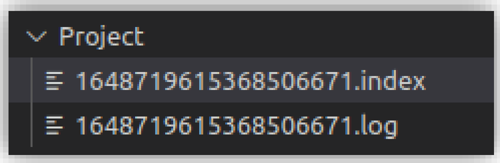
Project > 1648718465652381378.log

```
[0, "key3", "value3"]
[22, "key2", "NotDeleted"]

```

I introduced two new keys and values, one new key, “key3”, and updated the second key, “key2”.

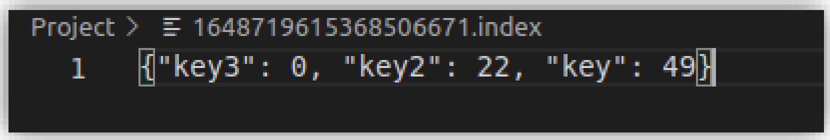
And this is what happens when we compact them:

Only one index and log generated:



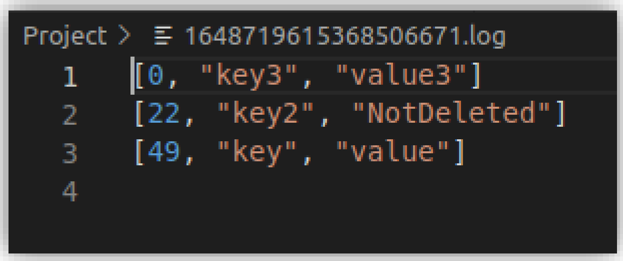
```
Project
  1648719615368506671.index
  1648719615368506671.log
```

New index with no repeated data:



```
Project > 1648719615368506671.index
{"key3": 0, "key2": 22, "key": 49}
```

New log with no repeated data:



```
Project > 1648719615368506671.log
[0, "key3", "value3"]
[22, "key2", "NotDeleted"]
[49, "key", "value"]

```

3.2. Document store

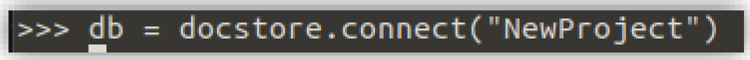
This document store uses the key-value store named above to function, using its own API, but based on that of the key-value store.

The list of commands or functionalities presented by this store is very similar to that of the key-value store since it is based on its functionalities and is as follows:

Connect(Path, opts={})

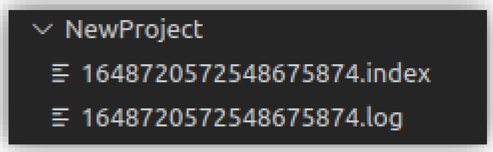
- Produces the connection against a database.
- With Path we indicate the database that we want to connect.
- With opts we can include additional options.

We can create a new database or connect to an existing one using this command:



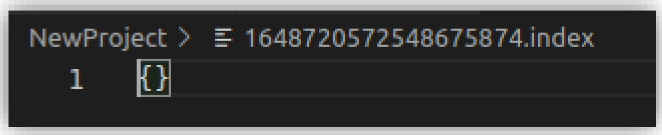
```
>>> db = docstore.connect("NewProject")
```

And it will look like this:



```
NewProject
  1648720572548675874.index
  1648720572548675874.log
```

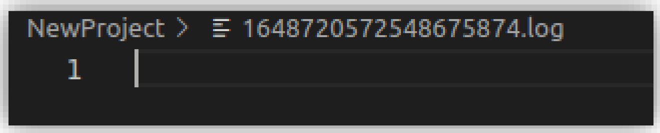
Empty index:



NewProject > 1648720572548675874.index

```
{}
```

Empty log:



NewProject > 1648720572548675874.log

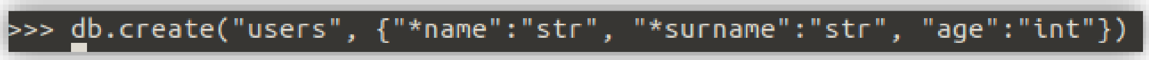
Same base as the key-value storage creation of the database.

Create (col, schema)

- Creates a collection of documents, col, with the properties specified in the schema, these properties will be chosen by the client as long as they meet established requirements.
- If any of the properties must be indexed, you must include a “ * ” in front of the name of the same.

This is where we can choose the type of document we are going to store and how we want to index it.

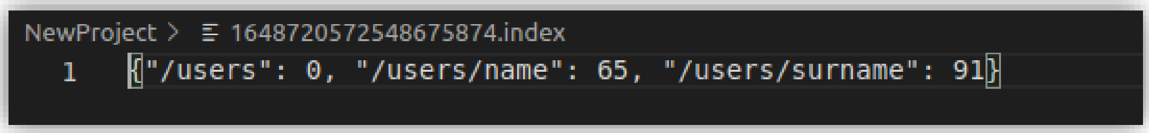
For example, if we want to add names, surnames and age of a group of people and also having their names and surnames indexed we can do this:



```
>>> db.create("users", {"*name":"str", "*surname":"str", "age":"int"})
```

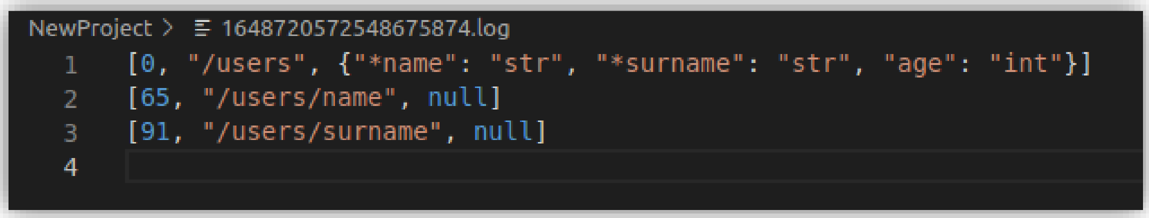
.

This will generate this index:



```
{"/users": 0, "/users/name": 65, "/users/surname": 91}
```

And this log:



```
[0, "/users", {"*name": "str", "*surname": "str", "age": "int"}]
[65, "/users/name", null]
[91, "/users/surname", null]
```

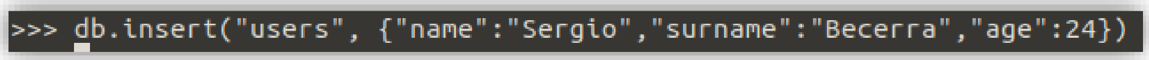
Since we can now search for the indexed data, it has to be included in the index and therefore in the log, but since there is no data for name or surname, they are set to null.

Later on, when we insert in the model “users” we will need to use the correct syntax that we decided before, so that we can easily search for it whenever we want.

Insert(col, doc)

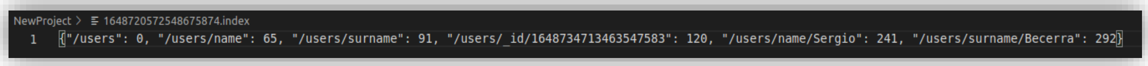
- The document, doc, is inserted in the collection, col,
- This document will have a unique ID with which it can be identified.

When we have already created a collection, we can start filling it with data like this:



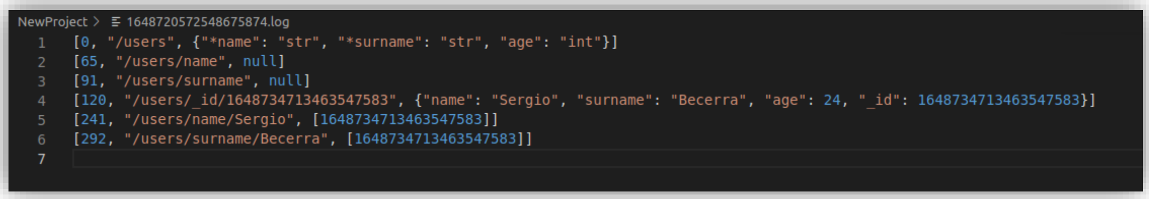
```
>>> db.insert("users", {"name":"Sergio","surname":"Becerra","age":24})
```

The index will create 3 new entries, since we have 2 indexed variables and the main entry itself:



```
{"/users": 0, "/users/name": 65, "/users/surname": 91, "/users/_id/1648734713463547583": 120, "/users/name/Sergio": 241, "/users/surname/Becerra": 292}
```

And it will be reflected in the log like this:



```
[0, "/users", {"*name": "str", "*surname": "str", "age": "int"}]
[65, "/users/name", null]
[91, "/users/surname", null]
[120, "/users/_id/1648734713463547583", {"name": "Sergio", "surname": "Becerra", "age": 24, "_id": 1648734713463547583}]
[241, "/users/name/Sergio", [1648734713463547583]]
[292, "/users/surname/Becerra", [1648734713463547583]]
```

As you can see this is where the data starts to stack really easily and that is why we need the segmentation and compact methods to make a little bit easier for the system to process.

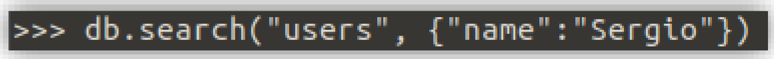
The indexed data like “name” and “surname” in this case will store the users ID since it is the only secure way to know that it is not duplicated.

Search(col, query)

- Searches the collection col for the documents that match the **query**.
- The queries will be dictionaries with equality conditions.
- A list with the documents that meet the query will be returned.

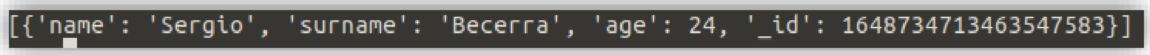
Searching can be done multiple ways, either by ID, or by using any of the indexed fields we determined before.

This is an example of how we can search by name in the collection “users” since we indexed it when we created it:



```
>>> db.search("users", {"name":"Sergio"})
```

Every single entry that contains the name Sergio in it will be shown, since we only have one entry that contains it, this is what is shows:

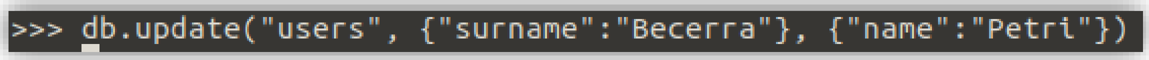


```
[{'name': 'Sergio', 'surname': 'Becerra', 'age': 24, '_id': 1648734713463547583}]
```

Update(col, query, data)

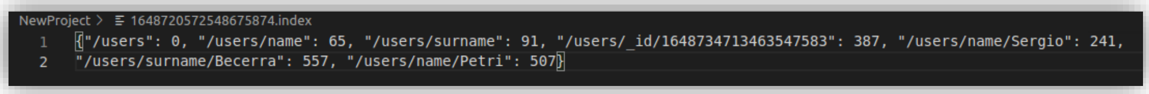
- Modifies existing documents in the col collection that verify the **query**
- The data previously saved in these documents will be replaced by those specified in **data**.

We can also replace any of the data fields we want by matching the query, in this case I'm going to change every single username which surname is Becerra to Petri:



```
>>> db.update("users", {"surname":"Becerra"}, {"name":"Petri"})
```

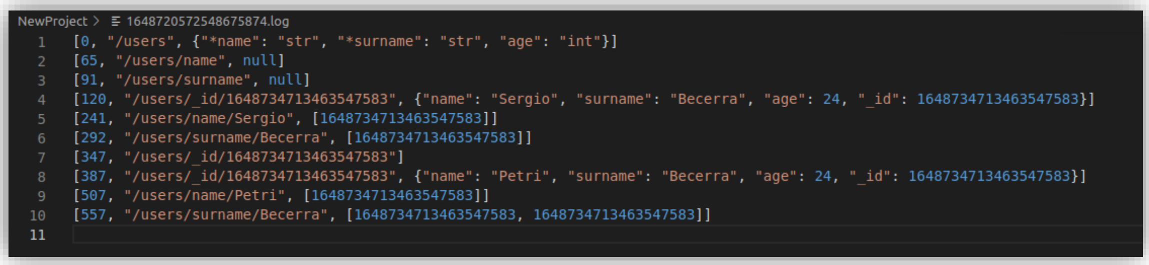
Only two new entries in the index since we did not create a new one, just modified an existing one:



NewProject > 1648720572548675874.index

```
{"/users": 0, "/users/name": 65, "/users/surname": 91, "/users/_id/1648734713463547583": 387, "/users/name/Sergio": 241,
"/users/surname/Becerra": 557, "/users/name/Petri": 507}
```

The log deleted all the data about the name "Sergio" in the collection users since we no longer have any named like that and created the new indexed data "Petri" in the name folder.



NewProject > 1648720572548675874.log

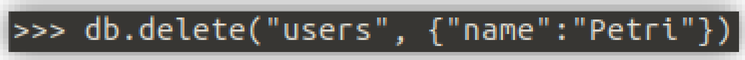
```
[0, "/users", {"*name": "str", "*surname": "str", "age": "int"}]
[65, "/users/name", null]
[91, "/users/surname", null]
[120, "/users/_id/1648734713463547583", {"name": "Sergio", "surname": "Becerra", "age": 24, "_id": 1648734713463547583}]
[241, "/users/name/Sergio", [1648734713463547583]]
[292, "/users/surname/Becerra", [1648734713463547583]]
[347, "/users/_id/1648734713463547583"]
[387, "/users/_id/1648734713463547583", {"name": "Petri", "surname": "Becerra", "age": 24, "_id": 1648734713463547583}]
[507, "/users/name/Petri", [1648734713463547583]]
[557, "/users/surname/Becerra", [1648734713463547583, 1648734713463547583]]

```

As you can also see there is two entries for the surname Becerra, but since it's the same ID that will only count as 1 as it will disappear when we segment/compact the data.

Delete(col, query)

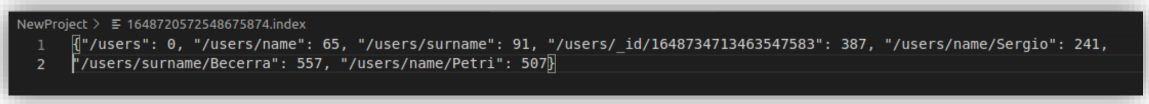
- Removes the documents from the col collection that match the **query**

We can delete any user by matching the query to an associated index, in this case we are going to delete every user called “Petri”



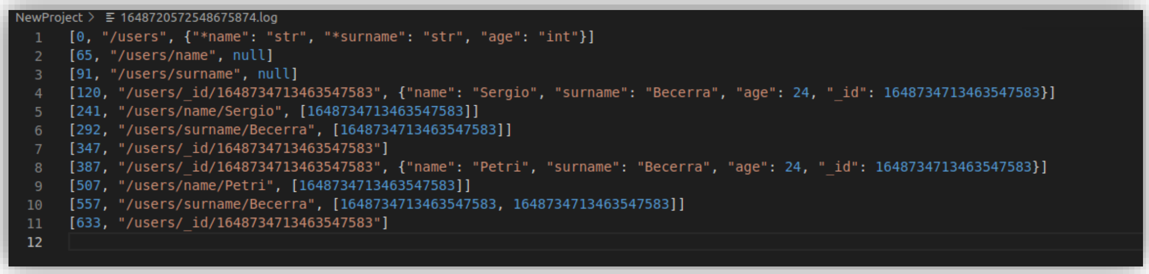
```
>>> db.delete("users", {"name":"Petri"})
```

The index will remain the same:



```
{"/users": 0, "/users/name": 65, "/users/surname": 91, "/users/_id/1648734713463547583": 387, "/users/name/Sergio": 241,
"/users/surname/Becerra": 557, "/users/name/Petri": 507}
```

But the log will delete the data for every user that was in the list of name/Petri, so whenever we look for the name “Petri” it will lead to an empty array, meaning it doesn´t exist.



```
[0, "/users", {"*name": "str", "*surname": "str", "age": "int"}]
[65, "/users/name", null]
[91, "/users/surname", null]
[120, "/users/_id/1648734713463547583", {"name": "Sergio", "surname": "Becerra", "age": 24, "_id": 1648734713463547583}]
[241, "/users/name/Sergio", [1648734713463547583]]
[292, "/users/surname/Becerra", [1648734713463547583]]
[347, "/users/_id/1648734713463547583"]
[387, "/users/_id/1648734713463547583", {"name": "Petri", "surname": "Becerra", "age": 24, "_id": 1648734713463547583}]
[507, "/users/name/Petri", [1648734713463547583]]
[557, "/users/surname/Becerra", [1648734713463547583, 1648734713463547583]]
[633, "/users/_id/1648734713463547583"]

```

Also, when the log gets compacted, the empty data will disappear making it faster for the database.

3.3. Server

The server is one layer above the document store, so it will use the document store's methods to serve multiple clients simultaneously, all of which will be able to access the same database.

To make the server work I have used the Flask framework that uses the RestFul protocol, which are REST services to work on the web, these specify certain restrictions such as a uniform interface and induce appropriate properties such as good scalability, performance, etc. (Fernandes, Lopes, Rodrigues & Ullah, 2013)

Flask logo

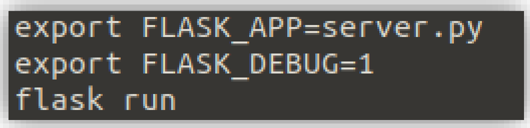
Rest logo

On the other hand, speaking of Flask, I have used Flask Python, which is a Python module that allows me to easily develop web, in this case, the main utility that I have used is the hosting of a server. (Grinberg, 2018)
Although it also offers many other options such as templates, URL routing...

Flask was created by a small group of programmers called "Pocoo" who mainly work on a few Python projects like the "Pygments" syntax highlighter and other projects.

Every day Flask becomes more popular becoming in early 2022 the 7th framework with the most stars on GitHub with a total of 57,584 stars. (Tao, 2022)

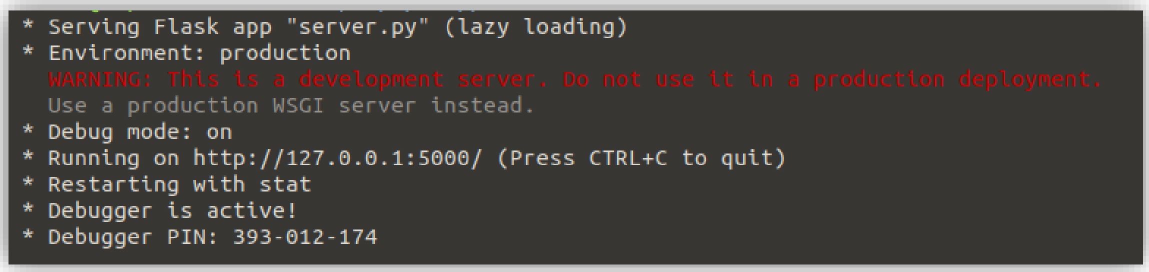
The commands we will use to start the server are:



```
export FLASK_APP=server.py
export FLASK_DEBUG=1
flask run
```

FLASK_APP will tell the server the script it has to use to run the server.
FLASK_DEBUG allows me to see everything that is happening to the server in order to be able to detect the connections.
And last but not least the server starts with the command “flask run”.

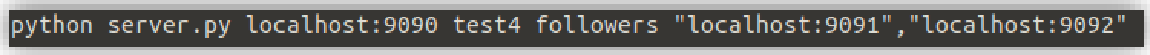
This will initialize the server, which will wait for a petition



```
 * Serving Flask app "server.py" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 393-012-174
```

That would be the normal use for a flask server, but since we will be using passive replication (explained in the point 3.6) we have to specify who will be the leader and who will be the followers for this connection.

In order to create the “**leader**” we have to use this syntax, specifying the “**followers**” of the “**leader**”, which will be receiving the data after the leader does.

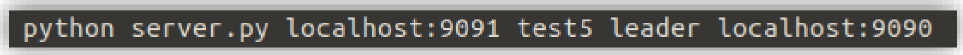


```
python server.py localhost:9090 test4 followers "localhost:9091","localhost:9092"
```

This “**leader**” will store the data in the database called “**test4**” and will send the data to the specified followers: “**localhost:9091**” and “**localhost:9092**”.

Also, “**server.py**” is the script that we will use to run the server.

On the other hand, if we want to create a “**follower**”, we will have to use this syntax:



```
python server.py localhost:9091 test5 leader localhost:9090
```

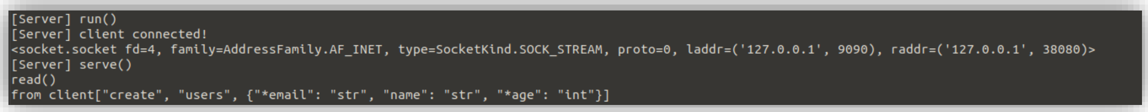
Specifying who the leader is, “**localhost:9090**”, and once again creating a new database called **test5** which it will be using for the passive replication.

The “followers” will work the same way as the “leader”, they will wait for a petition, but instead of receiving it from a client, they will receive it from the “leader”.

This way, we can a have a set of followers attached to a leader, enabling the passive replication to work.

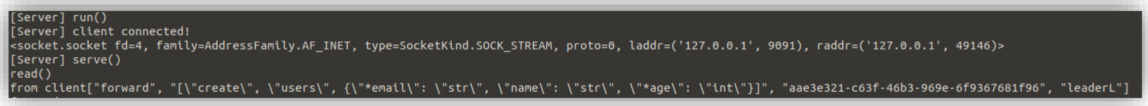
Once that is done the server will be running and waiting for petitions from the host.

Example of a “create” petition from the host to the “leader” server



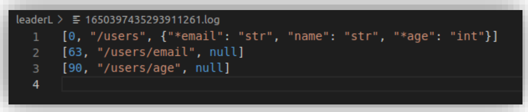
```
[Server] run()
[Server] client connected!
<socket.socket fd=4, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 9090), raddr=('127.0.0.1', 38080)>
[Server] serve()
read()
from client["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
```

And the petition when it reaches the “followers”

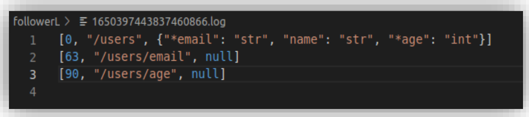


```
[Server] run()
[Server] client connected!
<socket.socket fd=4, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 9091), raddr=('127.0.0.1', 49146)>
[Server] serve()
read()
from client["forward", "[\"create\", \"users\", {\"*email\": \"str\", \"name\": \"str\", \"*age\": \"int\"}]", "aae3e321-c63f-46b3-969e-6f9367681f96", "leaderL"]
```

The log of the leader and the followers will look the same for now:

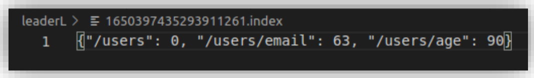


```
leaderL > 1650397435293911261.log
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
```

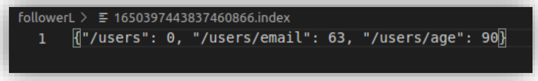


```
followerL > 1650397443837460866.log
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
```

And same thing for the index:



```
leaderL > 1650397435293911261.index
{"/users": 0, "/users/email": 63, "/users/age": 90}
```



```
followerL > 1650397443837460866.index
{"/users": 0, "/users/email": 63, "/users/age": 90}
```

Since it is replicated the data remains similar, but due to the Write-Ahead Log (WAL) system, there will be some redundant data inside of the followers since it has to check that every operation was made successfully, this however won’t be a problem because redundant will be eliminated later thanks to the **segments()** and **compacts()** methods.

The inner methods that the server will use are the following:

Run()

- The server boots up and starts listening for connection requests.

Serve(clientSocket)

- Upon receiving a connection request, this method is responsible for analyzing the request and returning a response to establish the connection.

This process will be carried out through writing and reading requests using JSON format, which is a format for exchanging data that is intuitive for people and simple to interpret for machines, this format is based on JavaScript and was created in December from 1999. (Smith, 2015)

JSON logo

The main structure of JSON is based on:

- A collection of key/value pairs.
- A set of related values in the form of an array or list.

This way, the communication between client and server that I have used in this part of the project is very simple, in which the client sends access requests to the server, be it reading, writing, searching, etc.
And these responds confirming if the request has been successful or notifies the error in case one has occurred.

3.4. Client

The client is the part that corresponds to the end user, which has functionalities similar to those of the server and is at the same level as the server.

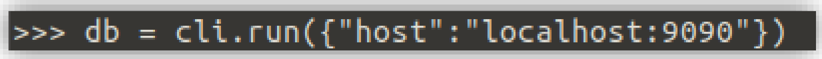
A private connection will be established between each client and the server where you can access the specified database and use the available features.

The commands that the client will have available will be:

Run(opts={})

- Connects the **client** with the "**leader**"

Using this command will allow the **client** to connect to the database using the "**leader**" replica.



```
>>> db = cli.run({"host":"localhost:9090"})
```

Connect(Path, opts={})

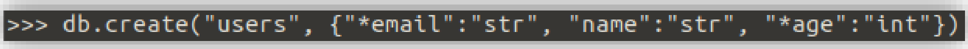
- Produces the connection against a database.
- With **Path** we indicate to the database that we want to connect.
- With **opts** we can include additional options.

This method will be used by every other method to create a connection every time it is needed.

Create (col, schema)

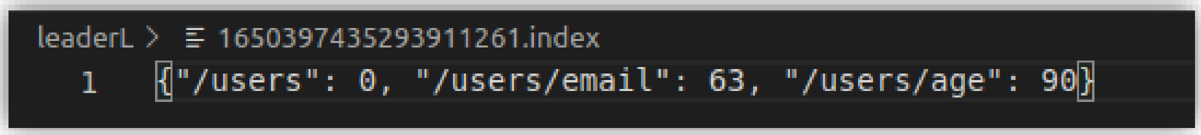
- Creates a collection of documents, **col**, with the properties specified in the **schema**, these properties will be chosen by the client as long as they meet established requirements.
- If any of the properties must be indexed, you must include an " * " in front of the property name.

The usage of this command is similar to the document store, but it will now generate a collection in the "**leader**" and all the "**followers**"



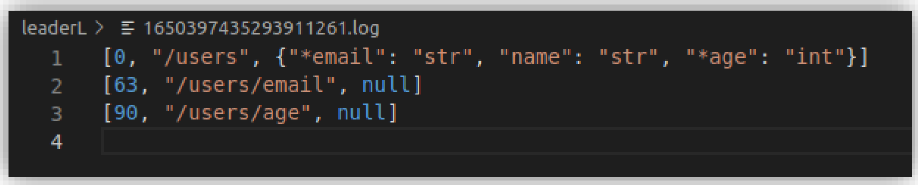
```
>>> db.create("users", {"*email":"str", "name":"str", "*age":"int"})
```

Leader index:



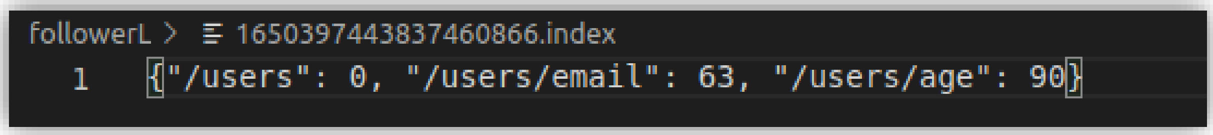
```
leaderL > 16503974352939l1261.index
{"/users": 0, "/users/email": 63, "/users/age": 90}
```

Leader log:



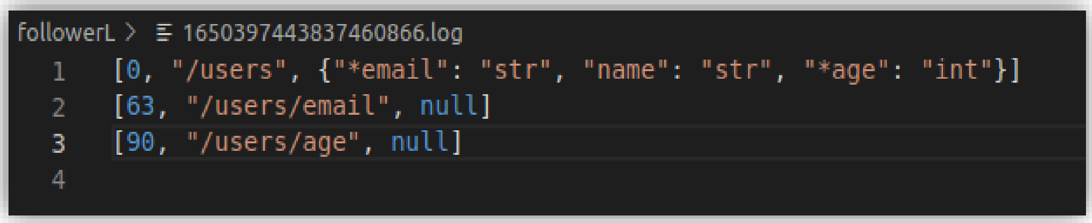
```
leaderL > 16503974352939l1261.log
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
```

Follower index:



```
followerL > 16503974438374608666.index
{"/users": 0, "/users/email": 63, "/users/age": 90}
```

Follower log:

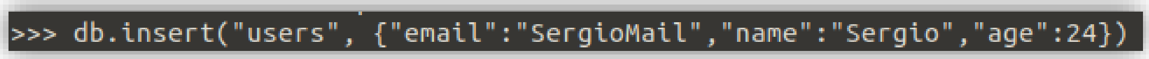


```
followerL > 16503974438374608666.log
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
```

Insert(col, doc)

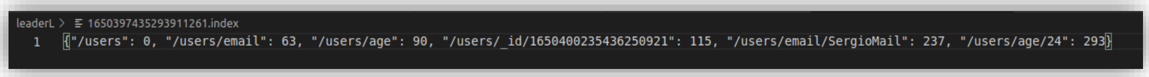
- The document, **doc**, is inserted in the collection, **col**,
- This document will have a unique ID with which it can be identified.

In this example I have introduced some data into the collection “users”.



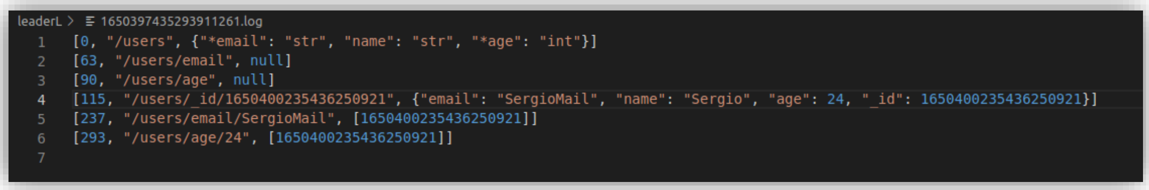
```
>>> db.insert("users", {"email":"SergioMail","name":"Sergio","age":24})
```

Same operation as seen in the document store.
Leader index:



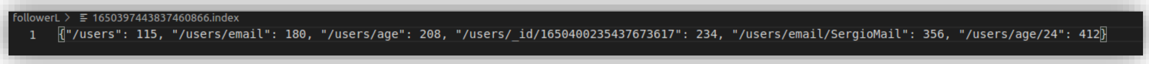
```
leaderL > 1650397435293911261.index
{"/users": 0, "/users/email": 63, "/users/age": 90, "/users/_id/1650400235436250921": 115, "/users/email/SergioMail": 237, "/users/age/24": 293}
```

Leader log:



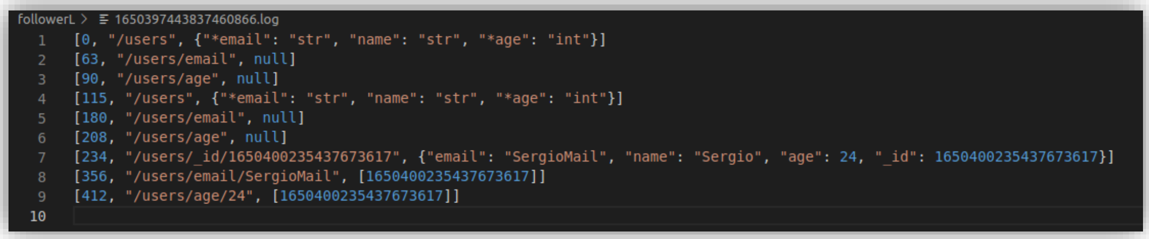
```
leaderL > 1650397435293911261.log
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
[115, "/users/_id/1650400235436250921", {"email": "SergioMail", "name": "Sergio", "age": 24, "_id": 1650400235436250921}]
[237, "/users/email/SergioMail", [1650400235436250921]]
[293, "/users/age/24", [1650400235436250921]]
```

Follower index:



```
followerL > 1650397443837460866.index
{"/users": 115, "/users/email": 180, "/users/age": 208, "/users/_id/1650400235437673617": 234, "/users/email/SergioMail": 356, "/users/age/24": 412}
```

Follower log:



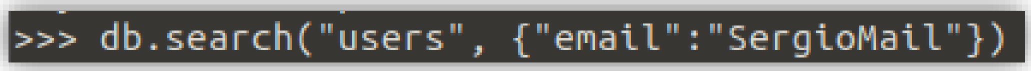
```
followerL > 1650397443837460866.log
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
[115, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[180, "/users/email", null]
[208, "/users/age", null]
[234, "/users/_id/1650400235437673617", {"email": "SergioMail", "name": "Sergio", "age": 24, "_id": 1650400235437673617}]
[356, "/users/email/SergioMail", [1650400235437673617]]
[412, "/users/age/24", [1650400235437673617]]
```

There is a duplicated operation in the follower log, due to the WAL system, that operation will later be deleted for better storaging,

Search(col, query)

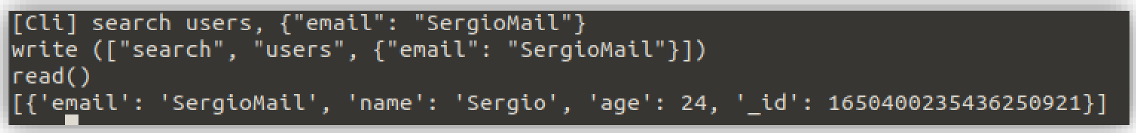
- Searches the collection col for the documents that match the **query**.
- The queries will be dictionaries with equality conditions.
- A list with the documents that meet the query will be returned.

To search for an specific data field we can use:



```
>>> db.search("users", {"email":"SergioMail"})
```

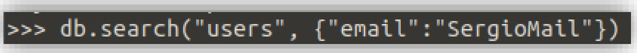
This will return all the users which email is “SergioMail”



```
[Cli] search users, {"email": "SergioMail"}
write (["search", "users", {"email": "SergioMail"}])
read()
[{'email': 'SergioMail', 'name': 'Sergio', 'age': 24, '_id': 1650400235436250921}]
```

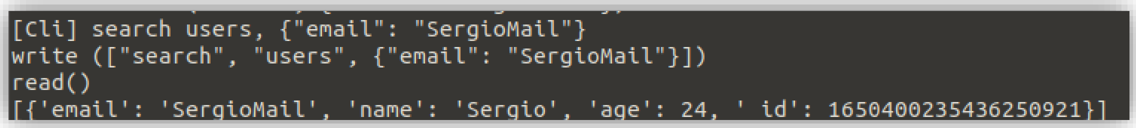
It is important to highlight that the search operation is the only that doesn´t require the “followers” to send the command to the client, since it is not a write operation, the “followers” can directly communicate with the client and receive the data directly.

Since the “leader” and “followers” databases are similar they will return the same results when we try to do the same search process from a “follower”:



```
>>> db.search("users", {"email":"SergioMail"})
```

Result:

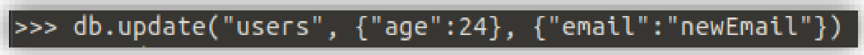


```
[Cli] search users, {"email": "SergioMail"}
write (["search", "users", {"email": "SergioMail"}])
read()
[{'email': 'SergioMail', 'name': 'Sergio', 'age': 24, '_id': 1650400235436250921}]
```

Update(col, query, data)

- Modifies existing documents in the col collection that verify the query.
- The data previously saved in these documents will be replaced by those specified in data.

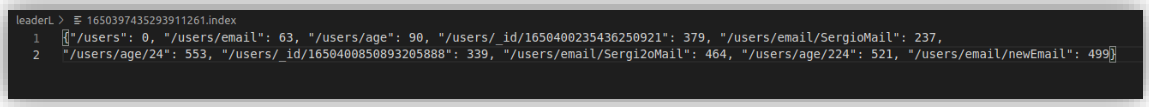
In order to update some already introduced data, we can use this:



```
>>> db.update("users", {"age":24}, {"email":"newEmail"})
```

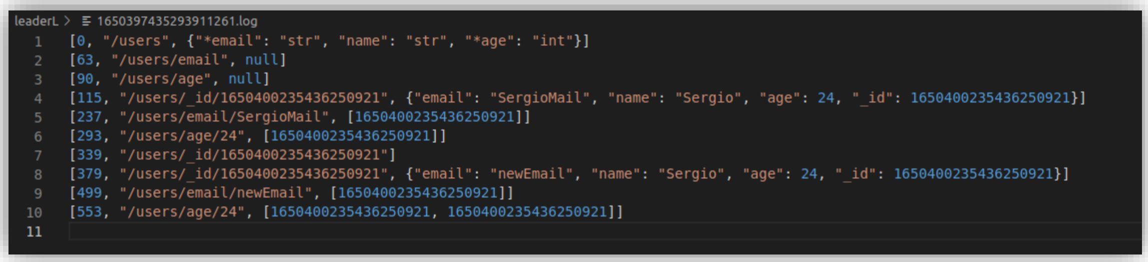
This will update every email data with age 24 to “newEmail”

This is how the new leader index looks like:



```
{"/users": 0, "/users/email": 63, "/users/age": 90, "/users/_id/1650400235436250921": 379, "/users/email/SergioMail": 237,
"/users/age/24": 553, "/users/_id/1650400850893205888": 339, "/users/email/Sergi2oMail": 464, "/users/age/224": 521, "/users/email/newEmail": 499}
```

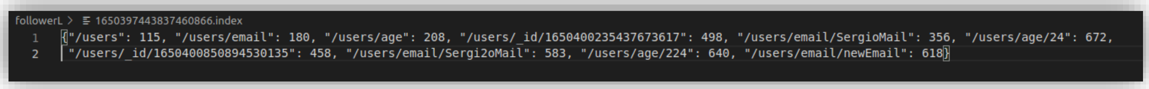
And the new leader log:



```
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
[115, "/users/_id/1650400235436250921", {"email": "SergioMail", "name": "Sergio", "age": 24, "_id": 1650400235436250921}]
[237, "/users/email/SergioMail", [1650400235436250921]]
[293, "/users/age/24", [1650400235436250921]]
[339, "/users/_id/1650400235436250921"]
[379, "/users/_id/1650400235436250921", {"email": "newEmail", "name": "Sergio", "age": 24, "_id": 1650400235436250921}]
[499, "/users/email/newEmail", [1650400235436250921]]
[553, "/users/age/24", [1650400235436250921, 1650400235436250921]]

```

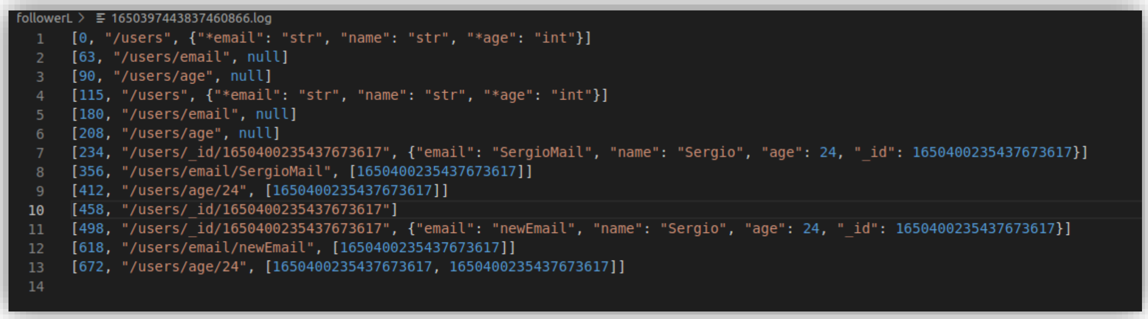
And the same process for the followers:

Follower index:



```
{"/users": 115, "/users/email": 180, "/users/age": 208, "/users/_id/1650400235437673617": 498, "/users/email/SergioMail": 356, "/users/age/24": 672,
 "/users/_id/1650400850894530135": 458, "/users/email/Sergi2oMail": 583, "/users/age/224": 640, "/users/email/newEmail": 618}
```

Follower log:

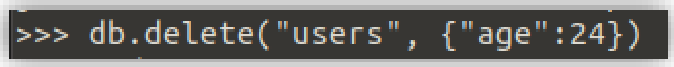


```
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
[115, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[180, "/users/email", null]
[208, "/users/age", null]
[234, "/users/_id/1650400235437673617", {"email": "SergioMail", "name": "Sergio", "age": 24, "_id": 1650400235437673617}]
[356, "/users/email/SergioMail", [1650400235437673617]]
[412, "/users/age/24", [1650400235437673617]]
[458, "/users/_id/1650400235437673617"]
[498, "/users/_id/1650400235437673617", {"email": "newEmail", "name": "Sergio", "age": 24, "_id": 1650400235437673617}]
[618, "/users/email/newEmail", [1650400235437673617]]
[672, "/users/age/24", [1650400235437673617, 1650400235437673617]]

```

Delete(col, query)

- Removes the documents from the col collection that match the query, query.

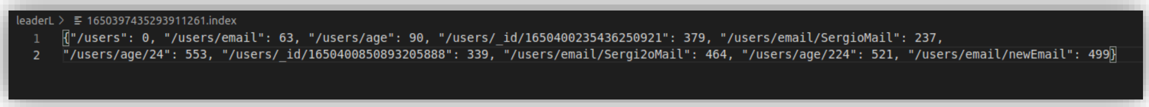
If we want to delete any of the data inside the database, we can use this:



```
>>> db.delete("users", {"age":24})
```

This will delete every user with age 24.

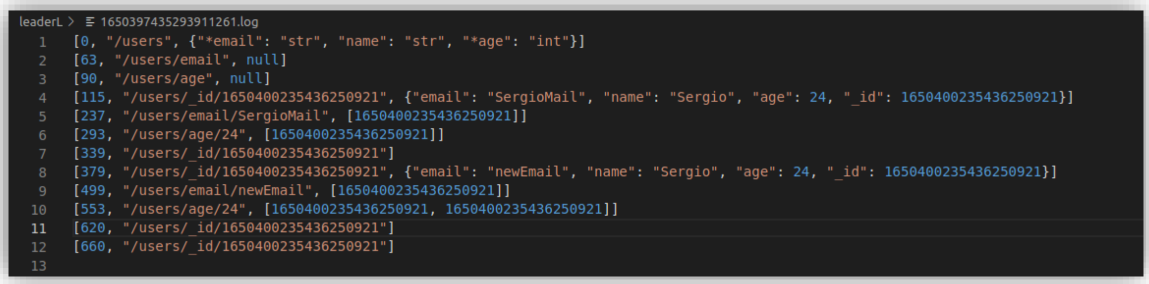
Leader index:



leaderL > 1650397435293911261.index

```
{"/users": 0, "/users/email": 63, "/users/age": 90, "/users/_id/1650400235436250921": 379, "/users/email/SergioMail": 237,
"/users/age/24": 553, "/users/_id/1650400850893205888": 339, "/users/email/Sergi2oMail": 464, "/users/age/224": 521, "/users/email/newEmail": 499}
```

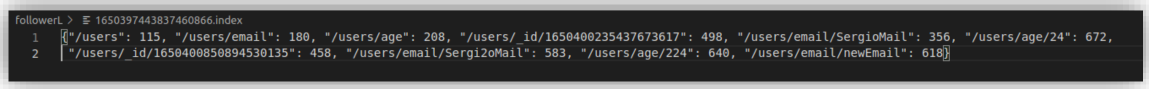
Leader log:



leaderL > 1650397435293911261.log

```
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
[115, "/users/_id/1650400235436250921", {"email": "SergioMail", "name": "Sergio", "age": 24, "_id": 1650400235436250921}]
[237, "/users/email/SergioMail", [1650400235436250921]]
[293, "/users/age/24", [1650400235436250921]]
[339, "/users/_id/1650400235436250921"]
[379, "/users/_id/1650400235436250921", {"email": "newEmail", "name": "Sergio", "age": 24, "_id": 1650400235436250921}]
[499, "/users/email/newEmail", [1650400235436250921]]
[553, "/users/age/24", [1650400235436250921, 1650400235436250921]]
[620, "/users/_id/1650400235436250921"]
[660, "/users/_id/1650400235436250921"]
```

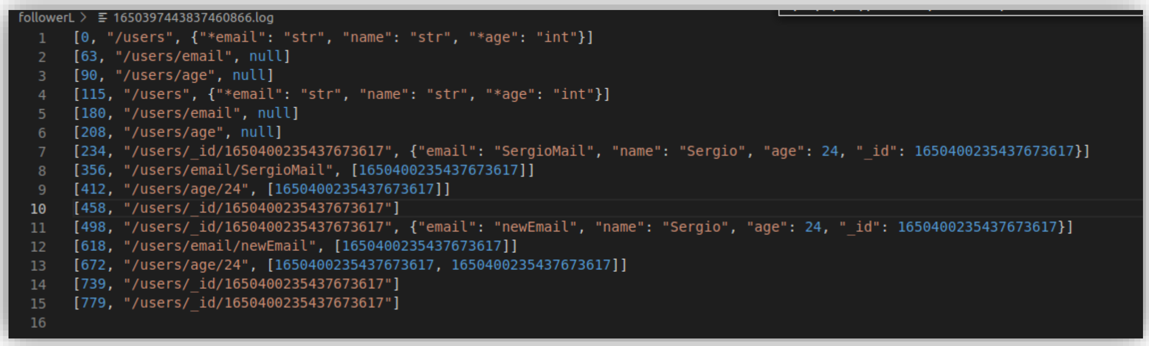
Follower index:



followerL > 1650397443837460866.index

```
{"/users": 115, "/users/email": 180, "/users/age": 208, "/users/_id/1650400235437673617": 498, "/users/email/SergioMail": 356, "/users/age/24": 672,
 "/users/_id/1650400850894530135": 458, "/users/email/Sergi2oMail": 583, "/users/age/224": 640, "/users/email/newEmail": 618}
```

Follower log:



followerL > 1650397443837460866.log

```
[0, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[63, "/users/email", null]
[90, "/users/age", null]
[115, "/users", {"*email": "str", "name": "str", "*age": "int"}]
[180, "/users/email", null]
[208, "/users/age", null]
[234, "/users/_id/1650400235437673617", {"email": "SergioMail", "name": "Sergio", "age": 24, "_id": 1650400235437673617}]
[356, "/users/email/SergioMail", [1650400235437673617]]
[412, "/users/age/24", [1650400235437673617]]
[458, "/users/_id/1650400235437673617"]
[498, "/users/_id/1650400235437673617", {"email": "newEmail", "name": "Sergio", "age": 24, "_id": 1650400235437673617}]
[618, "/users/email/newEmail", [1650400235437673617]]
[672, "/users/age/24", [1650400235437673617, 1650400235437673617]]
[739, "/users/_id/1650400235437673617"]
[779, "/users/_id/1650400235437673617"]
```

The indexes remained the same because there is no need to change them, since they are now pointing to an empty data, which is the same as being deleted.

At the end, for every operation, a connection will be produced between the **client** and the **server**, in which the client will send a request and wait for a response from the **server**, and when it gets a response, it will ensure that the data is consistent using the WAL system.

3.5. Write-Ahead Log (WAL)

To ensure the integrity of a database, a data recovery system is necessary in case of failure, for this project I have chosen to create a Write-Ahead Log (WAL).

This method is one of the fastest methods I have seen so far, and the computational cost is really low too that´s why I have chosen it. (Jhingran & Khedkar, 1992)

This recovery process is of vital importance for a database since a loss of information due to a power outage can cause the loss of millions of euros in a large company.

Its structure consists mainly of:

1. Create a log before writing to the disk in which the beginning of the operation will be written, and after writing to the disk, the end of the operation will be written in the log.

2. Every time an action occurs within the server, be it writing, reading, etc. It will check that the last operation started is finished and if not, it will be repeated so as not to lose any action

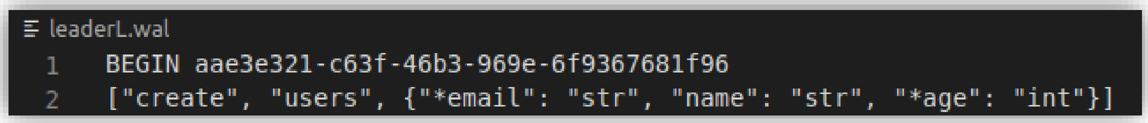
Since the operation itself will always be written in the WAL log as soon as it is received, it is possible to reproduce the request the necessary number of times until it is implemented correctly.

This process will ensure that the database will remain consistent and that it is a safe database.

3.5.1 What will be written in the Leader WAL log

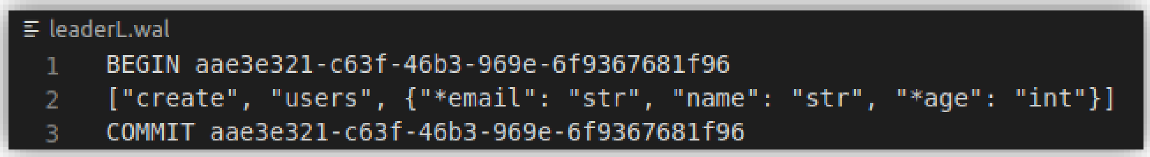
When the leader receives a request, it has to send the request to the followers and then execute the given request.

As soon as the leader receives a request, it will be reflected in the leader WAL log by the mark “BEGIN” and the ID of the request, Followed by the request itself



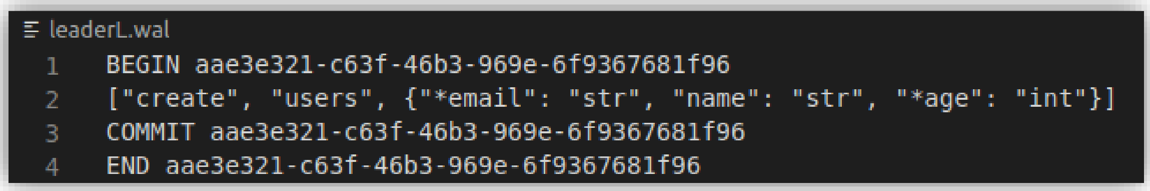
```
BEGIN aae3e321-c63f-46b3-969e-6f9367681f96
["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
```

When the leader receives the request, it will be sent to the followers, it that process is successful it will reflected in the wall with the mark “COMMIT” and then the ID of the request.



```
BEGIN aae3e321-c63f-46b3-969e-6f9367681f96
["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
COMMIT aae3e321-c63f-46b3-969e-6f9367681f96
```

Lastly, when the followers have received the request, the leader proceeds to execute it, it this process is successful, it will be marked in the log as the end of the operation by writing “END” and the ID of the request.



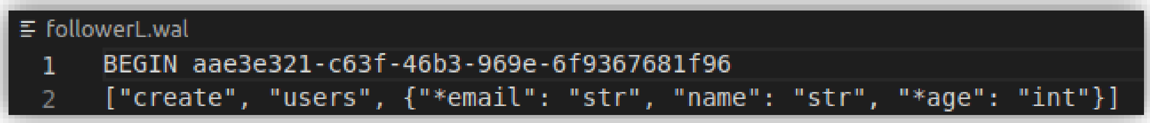
```
BEGIN aae3e321-c63f-46b3-969e-6f9367681f96
["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
COMMIT aae3e321-c63f-46b3-969e-6f9367681f96
END aae3e321-c63f-46b3-969e-6f9367681f96
```

This will mark the end of the operation, meaning that it was successful.

3.5.2 What will be written in the Followers WAL log

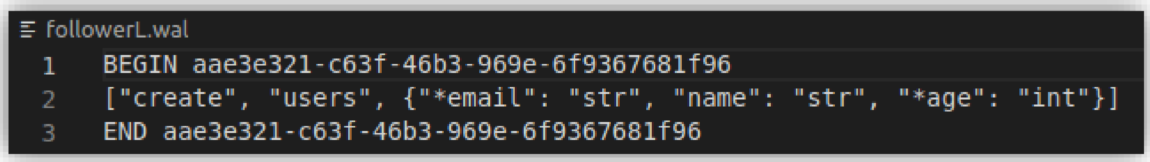
When the followers receive a request, they simply have to execute it.

As soon as the followers receives a request, it will be reflected in the follower WAL log by the mark “BEGIN” and the ID of the request, Followed by the request itself



```
BEGIN aae3e321-c63f-46b3-969e-6f9367681f96
["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
```

When the request is received it proceeds to be executed, when the execution is successful it is reflected in the follower WAL log by writing “END” and the request ID.



```
BEGIN aae3e321-c63f-46b3-969e-6f9367681f96
["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
END aae3e321-c63f-46b3-969e-6f9367681f96
```

The “END” marks that the operation was successful and there is no need to take any actions.

3.5.3 Potential WAL actions

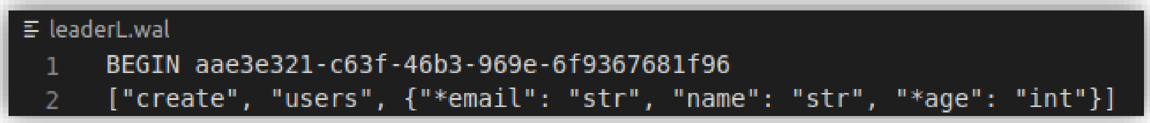
The WAL will be used when any of this action takes place:

- The request is sent but the receiver do not receive it.

In this case there is not much that the database can do, if the request doesn´t reach the server there is no way to replicate it.

- The request is received but not executed by the server or sent to the followers.

The leader WAL will look like this:

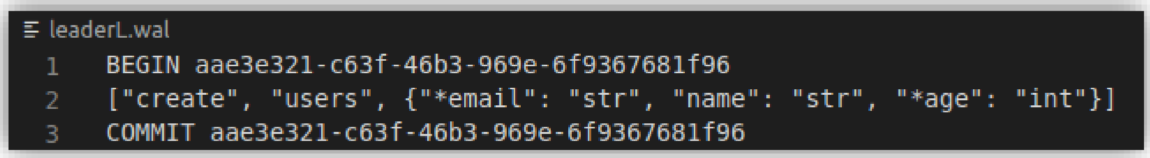


```
BEGIN aae3e321-c63f-46b3-969e-6f9367681f96
["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
```

In this case, since the server has the information about the request, even though it failed to execute it for some reason, it will be executed as soon as the next incoming request is received.

- The request is received by the leader and sent to the followers but not executed by the leader.

The leader WAL will look like this:

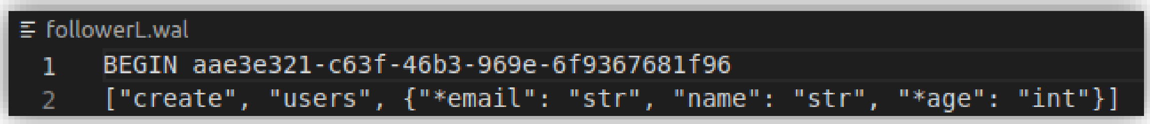


```
BEGIN aae3e321-c63f-46b3-969e-6f9367681f96
["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
COMMIT aae3e321-c63f-46b3-969e-6f9367681f96
```

If the leader manages to the sent the request to the followers but not execute it itself, it will reproduce the request when the next request is received.

- The request is received and sent to the followers, but the followers failed to execute it.

The followers WAL will look like this:



```
BEGIN aae3e321-c63f-46b3-969e-6f9367681f96
["create", "users", {"*email": "str", "name": "str", "*age": "int"}]
```

When a follower fails to execute a request, it will reproduce the request when the next request is received, the same way as the leader would operate.

3.6. Replication

To carry out the communication of multiple users with the same database, I have decided to use the passive replication strategy.

Passive replication is a form of replication that reduces the time it takes for file changes to be reflected on replicas. The origin server uses a notification system to immediately inform the client replica that it needs to be updated

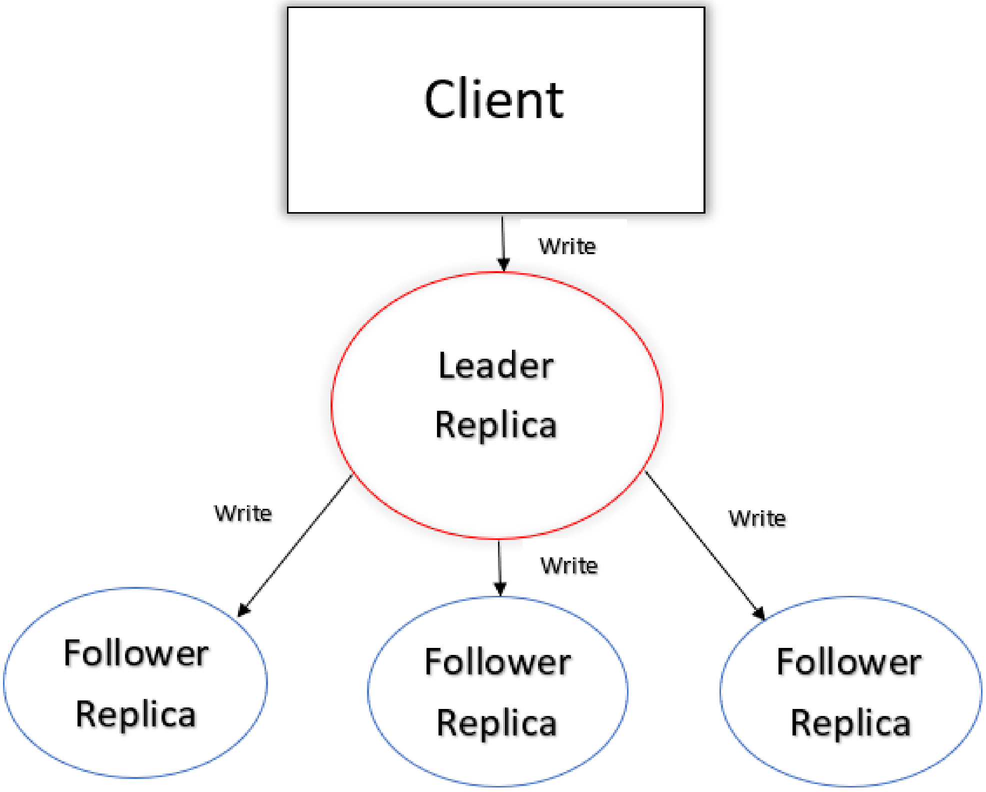
This strategy basically consists of two parts, leader and followers, the leader is chosen randomly from one of the replicas in the store and the rest will be the followers, the leader is the only one that can receive write requests, while everyone can receive reading requests, speeding up the process.

This leading replica is in charge of disseminating the information among the other replicas, either informing of a write or notifying an error produced in the leading replica, at the same time, the following replicas must also write about themselves when the leader notifies it or recover if necessary.

Carrying out this extra replication step requires a new check in the WAL to verify that no information has been lost between the leader and the followers.

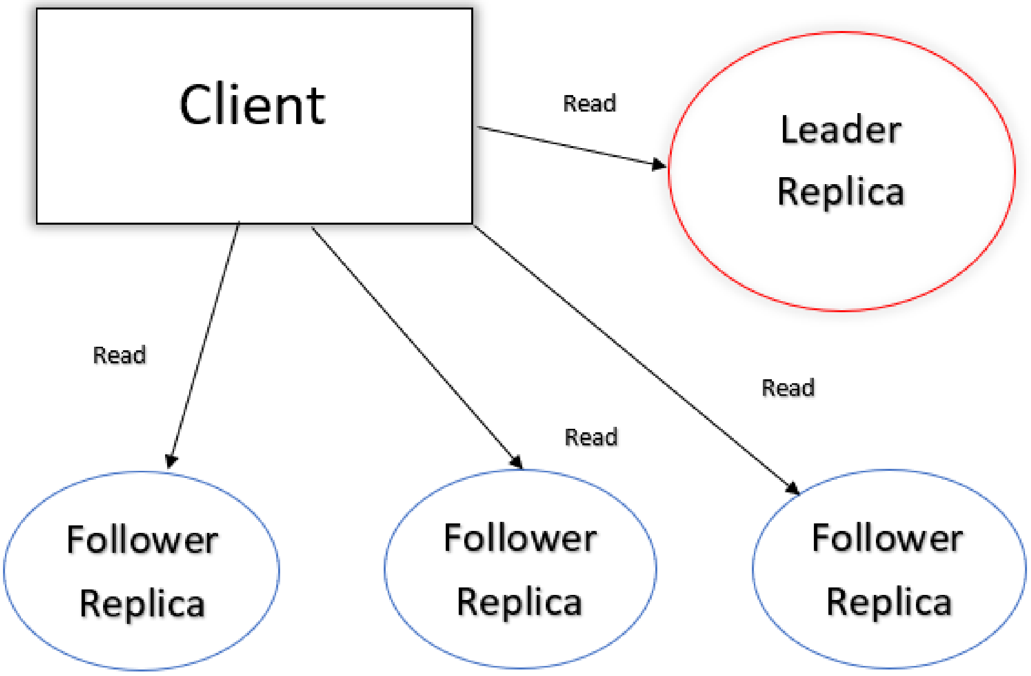
.

This is an example of how a **writing** operation would look like using this technique:



Since the follower replicas cannot directly receive a write operation from the client, it will the leader the one who will receive it, and then the leader will sent the petition to the followers, which will execute the petition to keep the information consistent

However, if instead of writing, we just want to **read,** it is possible directly communicate the client with the followers and the leader.



This way the client is able to search for information in any of the replicas directly if needed.

In some cases, this will help to verify the integrity of the database since we can manually see that the data identical in the leader replicas than in the followers.

4. Conclusions

The first time that this thesis idea came to my mind was in a Distributed storage and processing systems class, while learning about how important a good database is and how impactful they are on the world we live in.

Investigating a little bit about all the different types of databases I noticed that the type who had more potential to be improved was the non-relational type, even though they are not as good as the relational databases for the most part, I feel like in the future this can change significatively, and that´s why I started this project, so I could learn more and have a better understanding about how this type of databases works.

Of course, the database I created is not the fastest or the most reliable one, but it combines some of the ideas that I think are necessary for a database to be useful, being able to save any type of document, being able to share it in real-time with other users and being a consistent way of keeping your data safe is definitely the way to go for my work.

The key-value storaging is my most hated and loved part about this project, even though I considered it the hardest part to create, it is the part with most potential, being able to accelerate the speed that the data is introduced is definitely where I have spent most of my project time working, and it still have a lot of potential to be better.

About the document store, I want it to become more flexible, but for the moment it is a good base that can handle most of the documents that will be used.

Another part which consumed a lot of time of this project was the server-client communication, every database has to be able to host more than one user, but since I did not know much about how to create a server I started from the bases and although it was not so complicated it took me a long time to create it from start to end.

It still has some work to do, especially the security part, I want to create a login to ensure that no other user will be able to access the database unless the administrator allows it.

One of the most important parts about creating this database was the consistency, I had this in mind from the first minute, and changed the security system multiples times, but in the end, creating the Write-Ahead Log was the right choice, it is not computationally heave for the system to handle and it is capable of solving every single potential data loss inside the system.

Overall, I´m still learning about how to improve this project, and I have more ideas that I want to implement in a future, but so far, I´m happy about how this project ended up

5. Bibliography

Bartholomew, D. (2010). SQL vs. NoSQL. Linux Journal, 2010(195), 4.

Bhat, U., & Jadhav, S. (2010). Moving towards non-relational databases. International Journal of Computer Applications, 1(13), 40-47

Cassandra, A. (2014). Apache cassandra. Website. Available online at http://planetcassandra. org/what-is-apache-cassandra, 13.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2), 1-26

Denton, J. W., & Peace, A. G. (2003). Selection and use of MySQL in a database management course. Journal of Information Systems Education, 14(4), 40

Fernandes, J. L., Lopes, I. C., Rodrigues, J. J., & Ullah, S. (2013, July). Performance evaluation of RESTful web services and AMQP protocol. In 2013 Fifth international conference on ubiquitous and future networks (ICUFN) (pp. 810-815). IEEE

Grinberg, M. (2018). Flask web development: developing web applications with python. " O'Reilly Media, Inc.".

Haderle, D. J., & Jackson, R. D. (1984). IBM Database 2 overview. IBM Systems Journal, 23(2), 112-125.

Hammes, D., Medero, H., & Mitchell, H. (2014). Comparison of NoSQL and SQL Databases in the Cloud. Proceedings of the Southern Association for Information Systems (SAIS), Macon, GA, 21-22s

Jhingran, A., & Khedkar, P. (1992). Analysis of recovery in a database system using a write-ahead log protocol. Acm Sigmod Record, 21(2), 175-184

Khvoynitskaya , Sandra. (2020). Why do we need a database? Available at

https://www.itransition.com/blog/the-future-of-big-data

Lahiri, T., Chavan, S., Colgan, M., Das, D., Ganesh, A., Gleeson, M., ... & Zait, M. (2015, April). Oracle database in-memory: A dual format in-memory database. In 2015 IEEE 31st International Conference on Data Engineering (pp. 1253-1258). IEEE.

Mehul Nalin Vora, (2011), "Hadoop-HBase for large-scale data," Proceedings of 2011 International Conference on Computer Science and Network Technology, 2011, pp. 601-605, doi: 10.1109/ICCSNT.2011.6182030

Silva, Y. N., Almeida, I., & Queiroz, M. (2016, February). SQL: From traditional databases to big data. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (pp. 413-418).

Smith, B. (2015). Beginning JSON. Apress.

Smith, Chrish. (2019). Why databases are so important in our lives. Available at

https://knowtechie.com/why-databases-are-so-important-in-our-lives/

Tao Christhopher. (2022). Top 30 GitHub Python Projects at the beginning of 2022

Tyas Abi. (2022). The 63 Biggest Data Breaches (Updated for February 2022). Available at

https://www.upguard.com/blog/biggest-data-breaches