Designing A Dynamically Scalable MongoDB Cluster

Tee Kah Hui

Kalkidan Fekadu

Abstract

Distributed Database Systems are becoming increasingly important in various fields. We have designed a dynamically scalable MongoDB cluster, that can effectively handle multiple queries, and supports expansion and deletion of server instances, database migration, monitoring and caching. To achieve this goal, we use tools such as, PyMongo, Studio 3T, GridFS and Jupyter notebook. Additionally, each collection in the database has been sharded according to the instructions given in the project description. The link to our code can be found at code-link .

Table of Contents

1. Introduction

Distributed database management systems (DDBMS) has become the preferred choice for large organizations such as Google, Facebook & Amazon to store their data. Distributed databases provide the advantage of distributed computing for organizations to process large volumes of data while maintaining the adequate service level commitment. Some other advantages of DDBMS are (i) Management of data with different levels of transparency, (ii) Increased reliability and availability, (iii) Easier expansion, and (iv) Improved performance.

In this project, we developed a user-friendly solution to set up a DDBMS with local hosting. We used MongoDB as our database program. The main reason for choosing MongoDB is because MongoDB is a cross-platform document-oriented database program. It is a NoSQL database program. Compared to traditional relational database management systems (RDBMS), MongoDB uses JSON-like documents to store the data. Our baseline setup consists of one config server cluster, one mongo router, and three shard cluster that represents three database management systems (DBMS).

This paper is structured in the following manner. In section 2, we discuss the problem background and motivation. Then we showed some existing solutions in section 3. Section 4 is our proposed solution and evaluation of our solution is discussed in Section 5. Lastly, we suggested some future works in Section 6 and conclude the paper in Section 7.

2. Motivation

The advancement in technology has connected a lot of people via the Internet. Below are some of the numbers published in the 7th report of Data Never Sleeps.

- It is estimated that more than 4.39 billion people all across the globe are connected to the internet so far.

- Google, the popular search engine, processes about 4.5 million searches every second. The figure would be equivalent to 6.4 billion searches per day.
- 55,140 photos are shared on Instagram each minute, which has obtained a growth rate of 12% compared to 2018.

With the ever-increasing amount of data generated daily, a centralized database management is insufficient to provide enough storage and adequate efficiency required for operation. Hence, distributed database management systems are introduced to address the issues.

In this project, the objective is to build a DDBMS that is capable of processing large amounts of data. The data provided are user, article and reading information. Our proposed system fragments the data based on region and category into two DBMS. The DBMS are located in Hong Kong and Beijing respectively. Hence, an appropriate horizontal fragmentation is required on the data to enhance operation speed and efficiency.

3. Existing Solutions

According to Md. Shohel Rana et. al [1], distributed concurrency control, replication control and resource management are some of the problem areas of DDBMS. The ideal solution is every application can work transparently across different DBMS. To design a DDBMS that provides high performance, reliability, functionality and cost-saving, general design steps below must be followed.

- Group of logically-related communal data.
- Data split into fragments.
- Fragments may be replicated.
- Fragments/replicas assigned to sites.

- Sites connected by a communications network.
- Data at each site is under control of a DBMS.
- DBMSs grip local applications autonomously.

For our project, since the DDBMS does not exist, hence we will adopt the top-down approach in which we will start from requirement analysis, then conceptual design and view design. Next, we proceed to distributed design and finally physical design.

Structured query language (SQL) has been the de-facto language for RDBMS for almost four decades. However, traditional SQL databases such as Oracle, PostgreSQL and MySQL are monolithic from an architectural standpoint. They are unable to distribute data and queries across multiple instances automatically. However, since the introduction of Docker and Kubernetes, there is a new class of databases called "Distributed SQL" that came into the mainstream. Some of the well-known distributed SQL DBMS are Amazon Aurora and Google Cloud Spanner.

On the other hand, NoSQL is being introduced. It is a non-relational database that stores data in a format other than relational tables. NoSQL databases emerged in the late 2000s as the cost of storage dramatically decreased. NoSQL can be categorized into four types namely document databases, key-value databases, wide-column stores and graph databases. A few examples of distributed NoSQL databases are Cassandra, MongoDB, Hbase, Redis, Amazon Dynamo.

In this project, we will use a NoSQL DBMS with a document database. Hence, MongoDB is our choice.

4. Proposed Solution

Our solution is built using MongoDB, Studio 3T and Jupyter Notebook. In the following subsections, we discuss the architecture, data aggregation & loading, monitoring, tools and some extra features of our solution.

4.1. Architecture

Our solution is developed using MongoDB. We used the sharding method provided by MongoDB to simulate the distributed environment. Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

In our baseline model, three shard clusters will be created upon setup. Each shard will have their respective replica set to serve as a backup. As illustrated in Figure 1, shard-1 and shard-2 has one primary shard and one secondary shard. For shard-3, we created two secondary shards. Shard cluster 1 represents Hong Kong region (DBMS1) and shard cluster 2 represents Beijing region (DBMS2). Shard cluster 3 serves as a backup server that can store both the data from DBMS1 and DBMS2, using the migration functionality.

Besides shard clusters, we also created a config server cluster. The main function of config servers is to store the metadata for a sharded cluster. The metadata reflects state and organization for all data and components within the sharded cluster. The metadata includes the list of chunks on every shard and the ranges that define the chunks. These metadata is then used by the Mongos router to direct the read and write operations to the correct shard.

For unstructured data, we used the file system provided by MongoDB called GridFS. GridFS is a specification for storing and retrieving files that exceed the BSON-document size limit of 16 MB. It divides the file into parts (or chunks) and stores each chunk as a separate document. The default chunk size is 255 kB.

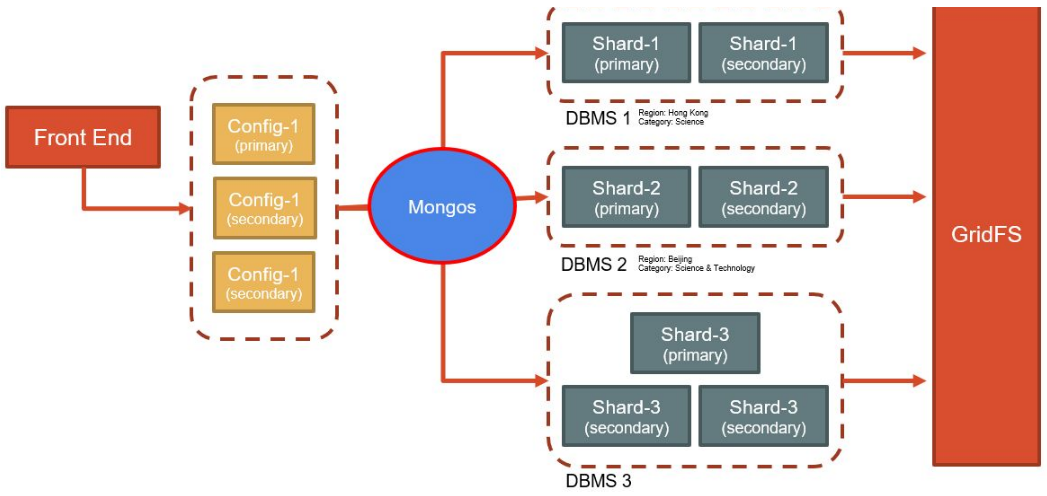


Figure 1: Architecture of proposed solution.

4.2. Cluster Setup

The design of MongoDB cluster setup allows it to be flexible and scalable for adding or removing servers and manipulating other database instances. Each server node inside a shard replica set is created using a config file. However, to avoid manual writing and editing of the config files, every config file is generated by the setup script. The setup script generates three types of config files: the ‘shardsvr’ config file, the ‘configsvr’ config file and mongos config file. As their names suggest, these config files are used to initialize the shard servers, config server and the mongos server respectively. During the initialization step, three config replica sets, three shard servers: the first two containing two and last containing three replica sets, and a mongos server are created.

4.2.1. Config Server

At this stage the script file gets the config files for each server instance, runs them using ‘*mongod -f config server_configfile_name*’ command and adds each of the config server instances into a replica set and initializes the replica set.

4.2.2. Mongos Server

The Mongos server needs information about all the config servers running in the system in order to be initialized. After getting the config files for the mongos server, the script runs ‘*mongos -f mongos_config_file_name*’ to initialize the mongos server.

4.2.3. Shard Replica

Similar to the config server and MongoDB server, after getting the config files for the shard replicas, the script runs '*mongod -f shardserver_config_file*' to initialize all the shard replica sets. After initializing the shard servers, each need to be added to a replica set, and the replica set that includes all the shard servers is also initialized. However, to enable sharding in the replica set, we have to connect with the mongos server and add the host address of the replica set and enable sharding. Now after the sharding is enabled, the servers are no longer a replica set but shard servers. The free monitoring tool provided by MongoDB is also enabled at this stage.

4.2.4. Final Step

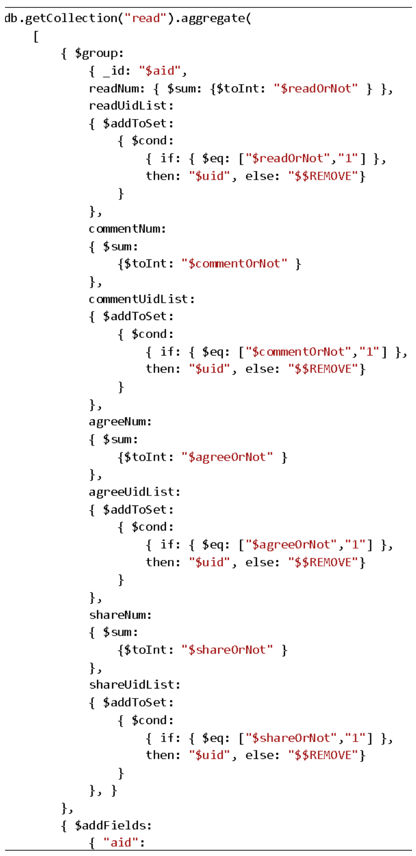
At the last step, shard keys are added to each collection in the database. And, chunks are moved to different servers based on the shard key.

4.3. Aggregation

Two main collections are generated using the aggregation scripts, Be-read and popular rank.

4.3.1. Be-read

Be-read collection is generated based on read collection and article collection. We use the MongoDB aggregate function to perform joining of two collections. Figure 2 below shows the sample code for aggregation.



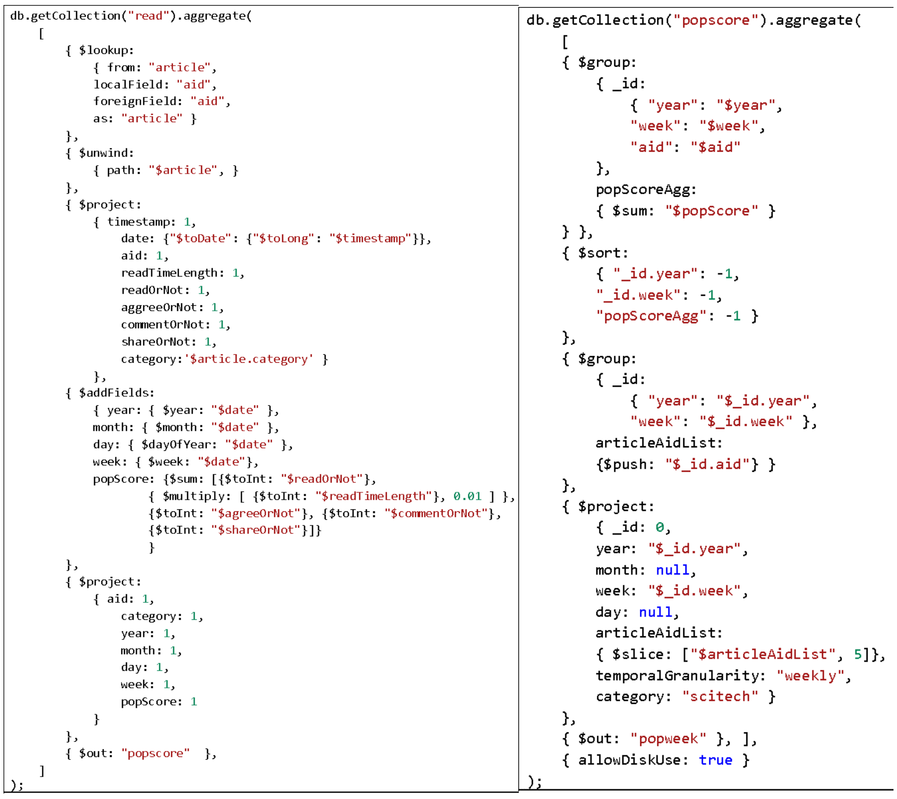
```
db.getCollection("read").aggregate(
    [
        { $group:
            { _id: "$aid",
            readNum: { $sum: {$toInt: "$readOrNot" } },
            readUidList:
            { $addToSet:
                { $cond:
                    { if: { $eq: ["$readOrNot","1"] },
                    then: "$uid", else: "$$REMOVE"}
                }
            },
            commentNum:
            { $sum:
                {$toInt: "$commentOrNot" }
            },
            commentUidList:
            { $addToSet:
                { $cond:
                    { if: { $eq: ["$commentOrNot","1"] },
                    then: "$uid", else: "$$REMOVE"}
                }
            },
            agreeNum:
            { $sum:
                {$toInt: "$agreeOrNot" }
            },
            agreeUidList:
            { $addToSet:
                { $cond:
                    { if: { $eq: ["$agreeOrNot","1"] },
                    then: "$uid", else: "$$REMOVE"}
                }
            },
            shareNum:
            { $sum:
                {$toInt: "$shareOrNot" }
            },
            shareUidList:
            { $addToSet:
                { $cond:
                    { if: { $eq: ["$shareOrNot","1"] },
                    then: "$uid", else: "$$REMOVE"}
                }
            }, }
        },
        { $addFields:
            { "aid":
```

Figure 2: Aggregation script for be-read collection.

4.3.2. Popular Rank

Popular rank collection is derived from read collection and article collection. First, we calculate the popular score using this Equation 1. The reason we include readTimeLength is to differentiate the ranking of articles that share the same number of read, share, comment and agree. In our opinion, an article that is being read for a long time indicates it is more popular. With the computed popular score, we used aggregate methods to create popular rank collections for different temporal granularity. Figure 3 shows the aggregate script for popular-score collection and aggregate script to generate monthly popular rank. Finally, popular rank collections with different temporal granularity are merged to form the final popular rank table.

$$\begin{aligned} \textit{Popular score} = \; & \textit{readOrNot} + \textit{shareOrNot} + \textit{agreeOrNot} + \\ & \textit{commentOrNot} + 0.01 * \textit{readTimeLength} \end{aligned} \qquad (1)$$



```
db.getCollection("read").aggregate(
    [
        { $lookup:
            { from: "article",
            localField: "aid",
            foreignField: "aid",
            as: "article" }
        },
        { $unwind:
            { path: "$article", }
        },
        { $project:
            { timestamp: 1,
                date: {"$toDate": {"$toLong": "$timestamp"}},
                aid: 1,
                readTimeLength: 1,
                readOrNot: 1,
                aggreeOrNot: 1,
                commentOrNot: 1,
                shareOrNot: 1,
                category:'$article.category' }
            },
        { $addFields:
            { year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfYear: "$date" },
            week: { $week: "$date"},
            popScore: {$sum: [{$toInt: "$readOrNot"},
                    { $multiply: [ {$toInt: "$readTimeLength"}, 0.01 ] },
                    {$toInt: "$agreeOrNot"}, {$toInt: "$commentOrNot"},
                    {$toInt: "$shareOrNot"}]}
                    }
        },
        { $project:
            { aid: 1,
                category: 1,
                year: 1,
                month: 1,
                day: 1,
                week: 1,
                popScore: 1
            }
        },
        { $out: "popscore"  },
    ]
);
```

```
db.getCollection("popscore").aggregate(
    [
    { $group:
        { _id:
            { "year": "$year",
            "week": "$week",
            "aid": "$aid"
        },
        popScoreAgg:
        { $sum: "$popScore" }
    } },
    { $sort:
        { "_id.year": -1,
        "_id.week": -1,
        "popScoreAgg": -1 }
    },
    { $group:
        { _id:
            { "year": "$_id.year",
            "week": "$_id.week" },
        articleAidList:
        {$push: "$_id.aid"} }
    },
    { $project:
        { _id: 0,
        year: "$_id.year",
        month: null,
        week: "$_id.week",
        day: null,
        articleAidList:
        { $slice: ["$articleAidList", 5]},
        temporalGranularity: "weekly",
        category: "scitech" }
    },
    { $out: "popweek" }, ],
    { allowDiskUse: true }
);
```

Figure 3: Aggregation script for popscore and poprank collections.

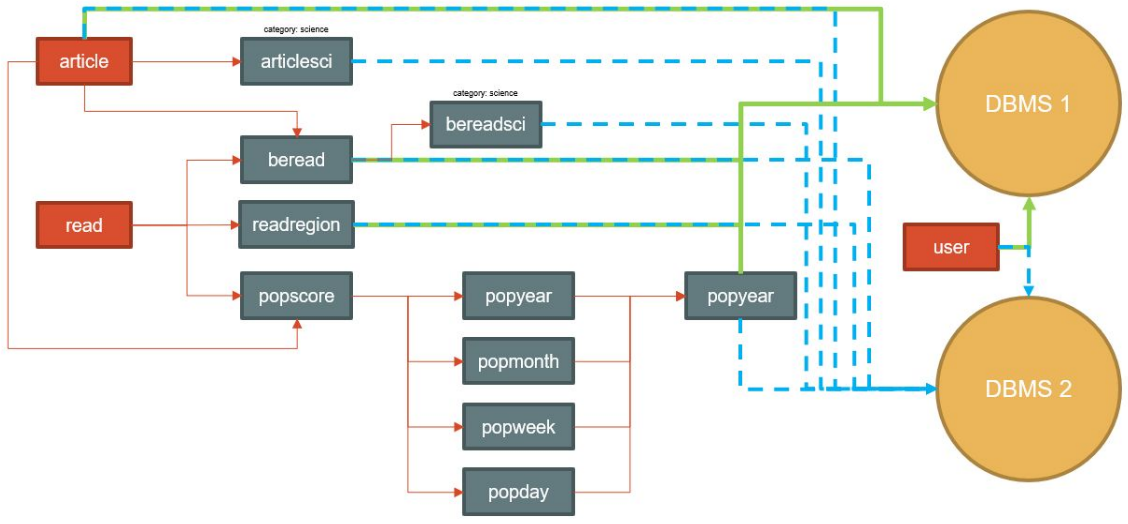


Figure 4: Aggregation and sharding map.

4.4. Data Loading

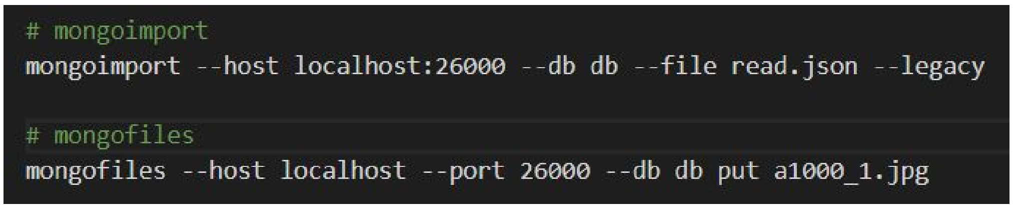
The two main tools we used to import the data are mongoimport and mongofiles. Both these commands are provided by MongoDB for importing contents. Below are the detailed description of both tools.

4.4.1. Mongoimport

The mongoimport tool imports content from an Extended JSON, CSV, or TSV export created by mongoexport, or potentially, another third-party export tool. An example of mongoimport is provided in Figure 5**.**

4.4.2. Mongofiles

The mongofiles utility makes it possible to manipulate files stored in your MongoDB instance in GridFS objects from the command line. It is particularly useful as it provides an interface between objects stored in your file system and GridFS. Example of mongofiles usage is shown in Figure 5.



```
# mongoimport
mongoimport --host localhost:26000 --db db --file read.json --legacy

# mongofiles
mongofiles --host localhost --port 26000 --db db put a1000_1.jpg
```

Figure 5: Example of mongoimport and mongofiles.

4.5. Tools

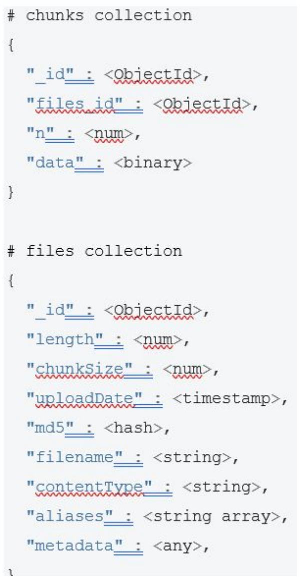
We use various tools for designing our database cluster, loading data, monitoring and making queries to the database.

4.5.1. GridFS

We are provided three types of unstructured data for the project which includes images, videos and text files. To store these unstructured data, we used the file system from MongoDB namely GridFS.

GridFS is a specification for storing and retrieving files that exceed the BSON-document size limit of 16 MB. Instead of storing a file in a single document, GridFS divides the file into parts, or chunks, and stores each chunk as a separate document. By default, GridFS uses a default chunk size of 255 kB with the exception of the last chunk. The last chunk is only as large as necessary.

Similarly, files that are no larger than the chunk size only have a final chunk, using only as much space as needed plus some additional metadata. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata. The section GridFS Collections describes each collection in detail. When querying GridFS for a file, the driver will reassemble the chunks as needed. Range queries on files stored can be performed through GridFS. Information from arbitrary sections of files, such as to "skip" to the middle of a video or audio file also can be performed in GridFS. When a file is stored in GridFS, two collections are created by MongoDB to keep the information namely fs.chunks and fs.files. Figure 6 shows the fields for chunks collection and files collection.



```
# chunks collection
{
  "_id" : <ObjectId>,
  "files_id" : <ObjectId>,
  "n" : <num>,
  "data" : <binary>
}

# files collection
{
  "_id" : <ObjectId>,
  "length" : <num>,
  "chunkSize" : <num>,
  "uploadDate" : <timestamp>,
  "md5" : <hash>,
  "filename" : <string>,
  "contentType" : <string>,
  "aliases" : <string array>,
  "metadata" : <any>,
}
```

Figure 6: Structure of chunks and files collection.

4.5.2. PyMongo

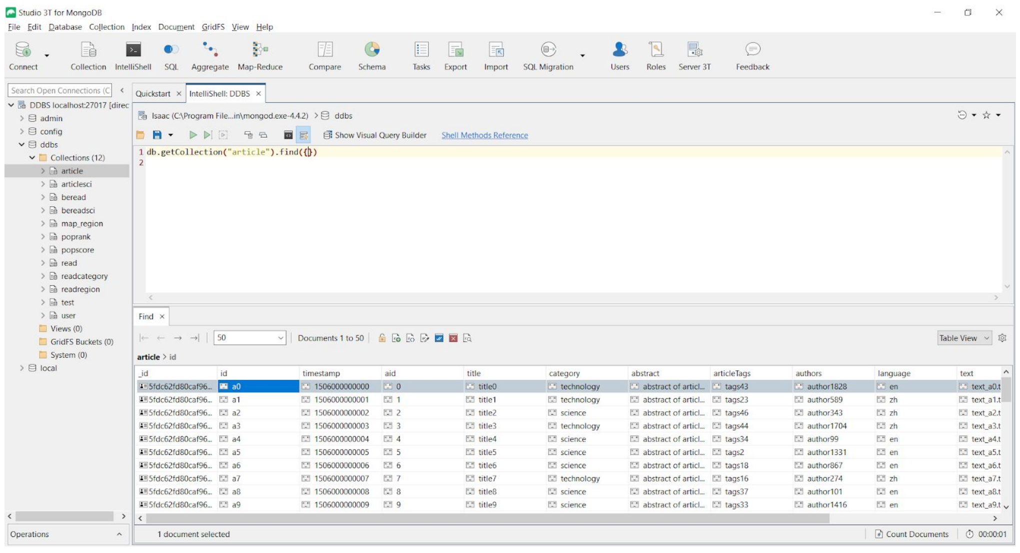
We used PyMongo API to perform queries on our database. PyMongo is a Python distribution containing tools for working with MongoDB. To connect to our database, PyMongo creates a MongoClient to the running mongod instance. It will connect to the dedicated host and port. An example connection request is shown in Figure 7.

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["DDBS"]
```

Figure 7: PyMongo connection to database.

4.5.3. Studio 3T

Studio 3T is an IDE & GUI software that provides some functionality to manage MongoDB databases. It does all the heavy lifting for developers in managing MongoDB. Some of the features we used in this project are Import/Export Wizard, Visual Query Builder, IntelliShell, Aggregation Editor. Figure 10 shows the user interface of Studio 3T.



```
db.getCollection("article").find({})
```

_id	id	timestamp	aid	title	category	abstract	articleTags	authors	language	text
5fdc62fd80caf96...	a0	1506000000000	0	title0	technology	abstract of articl...	tags43	author1828	en	text_a0.t
5fdc62fd80caf96...	a1	1506000000001	1	title1	technology	abstract of articl...	tags23	author589	zh	text_a1.t
5fdc62fd80caf96...	a2	1506000000002	2	title2	science	abstract of articl...	tags46	author343	zh	text_a2.t
5fdc62fd80caf96...	a3	1506000000003	3	title3	technology	abstract of articl...	tags44	author1704	zh	text_a3.t
5fdc62fd80caf96...	a4	1506000000004	4	title4	science	abstract of articl...	tags34	author99	en	text_a4.t
5fdc62fd80caf96...	a5	1506000000005	5	title5	science	abstract of articl...	tags2	author1331	en	text_a5.t
5fdc62fd80caf96...	a6	1506000000006	6	title6	science	abstract of articl...	tags18	author867	en	text_a6.t
5fdc62fd80caf96...	a7	1506000000007	7	title7	technology	abstract of articl...	tags16	author274	zh	text_a7.t
5fdc62fd80caf96...	a8	1506000000008	8	title8	science	abstract of articl...	tags37	author101	en	text_a8.t
5fdc62fd80caf96...	a9	1506000000009	9	title9	science	abstract of articl...	tags33	author1416	en	text_a9.t

Figure 8: Studio 3T interface

4.5.4. Jupyter Notebook

We used Jupyter notebook to visualize the efficient execution of data insert and update queries.

4.6. Monitoring

4.6.1. MongoDB 4.0 Free Monitoring

MongoDB provides free monitoring for version 4.0 and above. Some of the statistics available in free monitoring are operation execution time, memory usage, CPU usage and operation counts. The monitoring data will be made available for 24 hours. The free monitoring can be enabled and disabled during runtime using *db.enableFreeMonitoring()* and *db.disableFreeMonitoring()*. Figure 9 is the screenshot of MongoDB Free Monitoring.

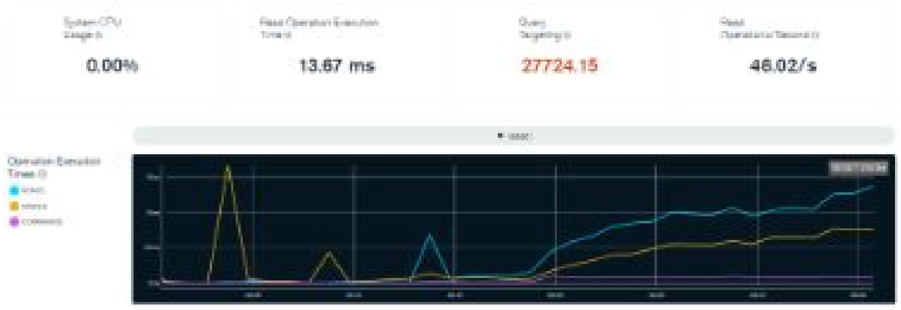
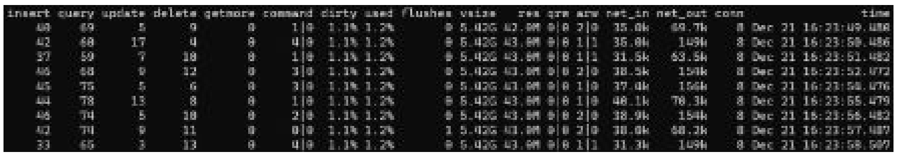


Figure 9: MongoDB free monitoring.

4.6.2. Mongostat

Mongostat is a utility that provides a quick overview of the status of a currently running mongod or mongos instance. mongostat is functionally similar to the UNIX/Linux file system utility vmstat, but provides data regarding mongod and mongos instances. Figure 10 is an example of mongostat**.**



```
insert query update delete getmore command dirty used flushes vsize   res qrw arw net_in net_out conn                time
    48    69      5      9       0     1|0  1.1% 1.2%       0 5.42G 43.0M 0|0 2|0  35.0k   69.7k    8 Dec 21 16:23:49.488
    42    68     17      4       0     4|0  1.1% 1.2%       0 5.42G 43.0M 0|0 1|1  35.0k    149k    8 Dec 21 16:23:50.486
    37    59      7     10       0     1|0  1.1% 1.2%       0 5.42G 43.0M 0|0 1|1  31.5k   63.5k    8 Dec 21 16:23:51.482
    46    68      9     12       0     3|0  1.1% 1.2%       0 5.42G 43.0M 0|0 2|0  38.5k    158k    8 Dec 21 16:23:52.472
    45    75      5      6       0     3|0  1.1% 1.2%       0 5.42G 43.0M 0|0 1|0  37.0k    156k    8 Dec 21 16:23:53.476
    44    78     13      8       0     1|0  1.1% 1.2%       0 5.42G 43.0M 0|0 1|0  40.1k   78.3k    8 Dec 21 16:23:55.479
    46    74      5     10       0     2|0  1.1% 1.2%       0 5.42G 43.0M 0|0 2|0  38.9k    154k    8 Dec 21 16:23:56.482
    42    74      9     11       0     0|0  1.1% 1.2%       1 5.42G 43.0M 0|0 2|0  38.0k   68.3k    8 Dec 21 16:23:57.487
    33    65      3     13       0     4|0  1.1% 1.2%       0 5.42G 43.0M 0|0 1|1  31.3k    149k    8 Dec 21 16:23:58.507
```

Figure 10: mongostat.

4.6.3. Studio 3T server status chart

Server status chart is a utility provided by Studio 3T for real time monitoring of MongoDB instances. Figure 11 shows the screenshot of the server status chart.

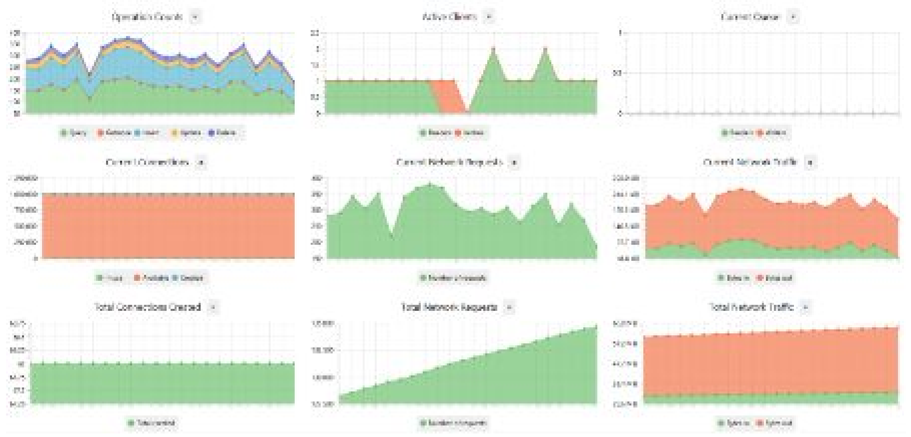


Figure 11: Studio 3T server status chart.

4.7. Additional Features

Our DB cluster also includes additional features that are essential for managing a distributed database system. The additional features are Expanding the DBMS servers, Monitoring, dropping a DBMS Server, and migration. Each of the features will be discussed in-depth below.

4.7.1. Expanding DBMS Server

Dynamically adding a DBMS server to the system can be very useful for making the system fault tolerant. The user can add a DBMS server by choosing the feature from the list provided by the script that manages the server instances. The user also needs to enter the number of replicas set for creating a shard server. After creation the new DBMS server will be set up and the user can check its status using the monitoring functionality.

4.7.2. Monitoring

The monitoring includes two important information about the DBMS status. First it shows the shard status for each server. Second it provides the URL to the free monitoring tool provided by MongoDB. This monitoring tool includes information on operation execution time, memory usage, CPU usage and operation count. Since this monitoring is provided by MongoDB, the information will expire after 24 hours.

4.7.3. Dropping DBMS Server

This functionality can be used for dropping a single server from a shard replica set. However, users need to make sure that this operation won't affect the running of the system. If a shard server has less than 3 replicas, it's not advisable to remove the servers from the replica. After the server's removal the system won't be able to elect a primary, and it will only have secondary servers. Unless the read preference is changed to enable reading from the secondary, the system can no longer make queries to the shard repl.

4.7.4. Migration

This feature enables users to copy data from one shard server to another. To achieve this goal we use *mongodump* and *mongorestore* commands.

4.7.5. Remove All Server Instances

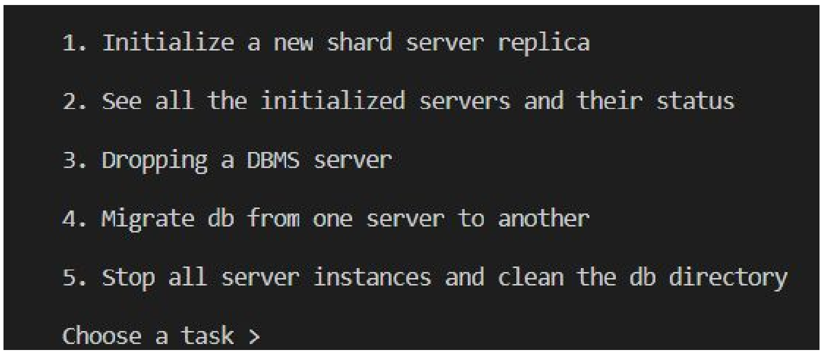
The last features provided by our system that, the removal of all server instances. Running this operation drops all servers, removes all the config, db, and log files and stops the script. This feature is very essential when we want to restart the servers again after failure. The processes of the servers won't collide the second time we run the script. Hence, the user needs to make sure all the server instances are cleared before running the server script file for the second time. To see if there remains an initialize instance of a mongo server running in the system, we can run 'ps -ef | grep mongo' on linux of MacOS. Please search for the equivalent command in Windows.

4.7.6. Redis Cache Server

After the setup and bulk import stage a Redis server is started to be used as a cache server. After a user makes a query to the server, if it's a very frequent query, the user can store the query and the returned value as a key value pair in the redis cache server. This technique is very useful for speeding up data access and avoiding directing queries to the Mongo cluster, which is costly.

4.8. User Manual

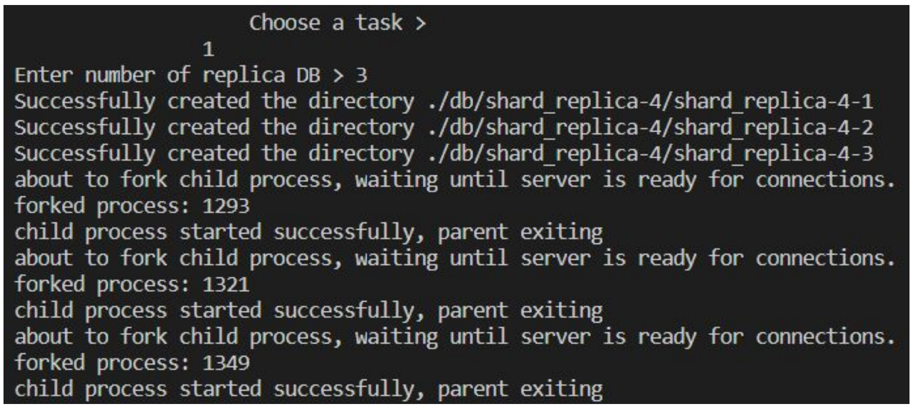
After setup, the user will see this text on the terminal window. The text notifies the user that the setup step has finished, and now the user can choose a task from the numbered list.



```
1. Initialize a new shard server replica

2. See all the initialized servers and their status

3. Dropping a DBMS server

4. Migrate db from one server to another

5. Stop all server instances and clean the db directory

Choose a task >
```

Figure 12: Menu option.

4.8.1. Initialize a new shard server replica



```
                    Choose a task >
                1
Enter number of replica DB > 3
Successfully created the directory ./db/shard_replica-4/shard_replica-4-1
Successfully created the directory ./db/shard_replica-4/shard_replica-4-2
Successfully created the directory ./db/shard_replica-4/shard_replica-4-3
about to fork child process, waiting until server is ready for connections.
forked process: 1293
child process started successfully, parent exiting
about to fork child process, waiting until server is ready for connections.
forked process: 1321
child process started successfully, parent exiting
about to fork child process, waiting until server is ready for connections.
forked process: 1349
child process started successfully, parent exiting
```

Figure 13: Add a new shard.

As we can see on the screen shot above a *shard_replica_4* has been successfully initialized.

4.8.2. See all initialized server and their status

The monitoring information mainly provides three pieces of information. It prints out information from ‘sh.status’. If a shard replica has a shard server, it prints out the address of the primary server in that shard replica. Lastly it provides the URL the cloud mongodb monitoring tool for each shard replica set. Sample monitoring graphs after bulk loading are shown in figure 14. As we can see the write and command operations have been executed at high frequency.

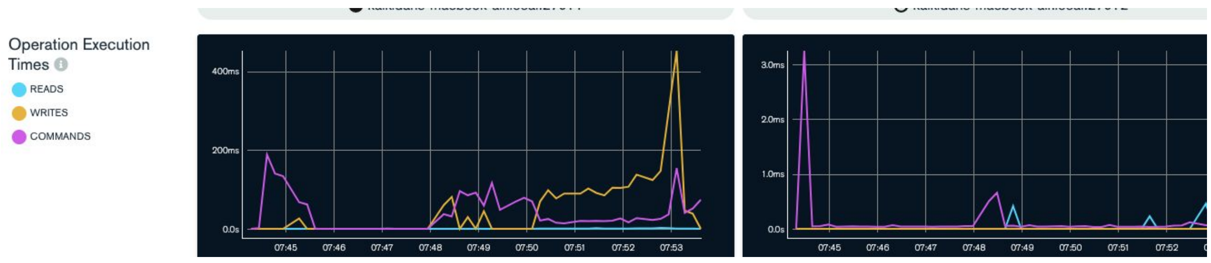
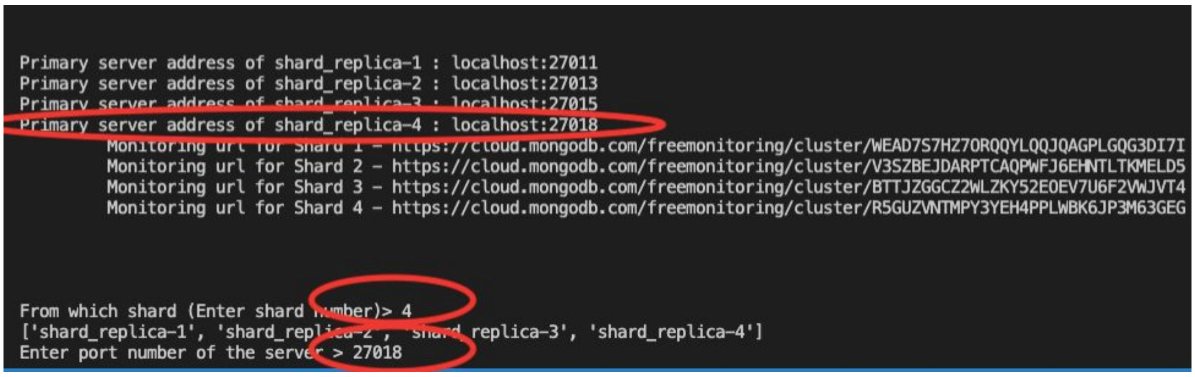


Figure 14: Sample monitoring graph.

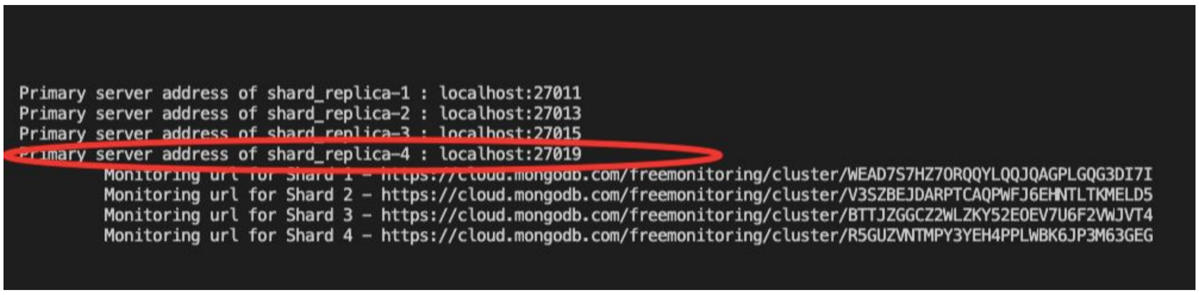
4.8.3. Dropping a DBMS server

Before dropping the server the address of the primary server in the shard is localhost:27018. After dropping the primary server, since the server with port number 27018 is down, a new primary server with port number 27019 has been elected to be the primary server.



```
Primary server address of shard_replica-1 : localhost:27011
Primary server address of shard_replica-2 : localhost:27013
Primary server address of shard_replica-3 : localhost:27015
Primary server address of shard_replica-4 : localhost:27018
        Monitoring url for Shard 1 - https://cloud.mongodb.com/freemonitoring/cluster/WEAD7S7HZ7ORQQYLQQJQAGPLGQG3DI7I
        Monitoring url for Shard 2 - https://cloud.mongodb.com/freemonitoring/cluster/V3SZBEJDARPTCAQPWFJ6EHNTLTKMELD5
        Monitoring url for Shard 3 - https://cloud.mongodb.com/freemonitoring/cluster/BTTJZGGCZ2WLZKY52EOEV7U6F2VWJVT4
        Monitoring url for Shard 4 - https://cloud.mongodb.com/freemonitoring/cluster/R5GUZVNTMPY3YEH4PPLWBK6JP3M63GEG


From which shard (Enter shard number)> 4
['shard_replica-1', 'shard_replica-2', 'shard_replica-3', 'shard_replica-4']
Enter port number of the server > 27018
```

Figure 15: Before dropping the DBMS port 27018

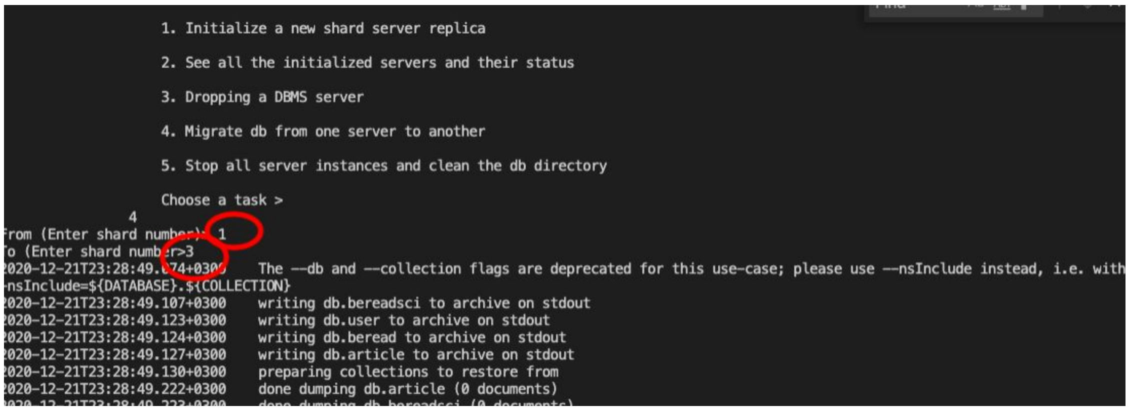


```
Primary server address of shard_replica-1 : localhost:27011
Primary server address of shard_replica-2 : localhost:27013
Primary server address of shard_replica-3 : localhost:27015
Primary server address of shard_replica-4 : localhost:27019
        Monitoring url for Shard 1 - https://cloud.mongodb.com/freemonitoring/cluster/WEAD7S7HZ7ORQQYLQQJQAGPLGQG3DI7I
        Monitoring url for Shard 2 - https://cloud.mongodb.com/freemonitoring/cluster/V3SZBEJDARPTCAQPWFJ6EHNTLTKMELD5
        Monitoring url for Shard 3 - https://cloud.mongodb.com/freemonitoring/cluster/BTTJZGGCZ2WLZKY52EOEV7U6F2VWJVT4
        Monitoring url for Shard 4 - https://cloud.mongodb.com/freemonitoring/cluster/R5GUZVNTMPY3YEH4PPLWBK6JP3M63GEG
```

Figure 16: After dropping the DBMS port 27018

4.8.4. Migrate db from one server to another

Migrating db from shard 1 to shard 3, and afterwards checking if the user collection has been moved successfully.

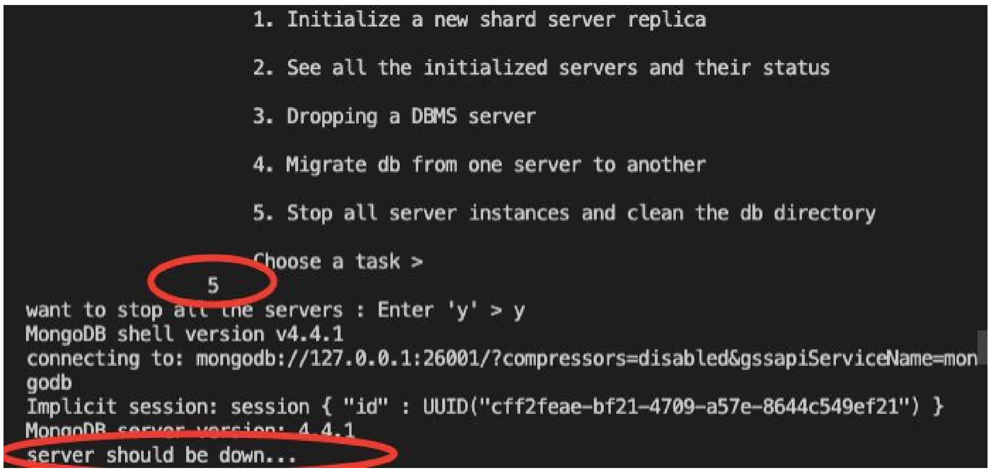


```
                    1. Initialize a new shard server replica

                    2. See all the initialized servers and their status

                    3. Dropping a DBMS server

                    4. Migrate db from one server to another

                    5. Stop all server instances and clean the db directory

                    Choose a task >
                4
rom (Enter shard number): 1
o (Enter shard number>3
020-12-21T23:28:49.074+0300    The --db and --collection flags are deprecated for this use-case; please use --nsInclude instead, i.e. with
nsInclude=${DATABASE}.${COLLECTION}
020-12-21T23:28:49.107+0300    writing db.bereadsci to archive on stdout
020-12-21T23:28:49.123+0300    writing db.user to archive on stdout
020-12-21T23:28:49.124+0300    writing db.beread to archive on stdout
020-12-21T23:28:49.127+0300    writing db.article to archive on stdout
020-12-21T23:28:49.130+0300    preparing collections to restore from
020-12-21T23:28:49.222+0300    done dumping db.article (0 documents)
```

Figure 17: Migration status report

4.8.5. Stop all server instances and clean db directory

Menu option 5 is for shutting down all server instances.



```
                1. Initialize a new shard server replica

                2. See all the initialized servers and their status

                3. Dropping a DBMS server

                4. Migrate db from one server to another

                5. Stop all server instances and clean the db directory

                Choose a task >
         5
want to stop all the servers : Enter 'y' > y
MongoDB shell version v4.4.1
connecting to: mongodb://127.0.0.1:26001/?compressors=disabled&gssapiServiceName=mon
godb
Implicit session: session { "id" : UUID("cff2feae-bf21-4709-a57e-8644c549ef21") }
MongoDB server version: 4.4.1
server should be down...
```

Figure 18: Shutting down server

5. Solution Evaluation

We evaluate our solution based on efficiency and completeness. The evaluation is as below.

5.1. Efficiency

Efficiency is determined based on the 10 GB dataset provided and Intel® Core™ i5-7200U CPU @ 2.50GHz with 4.00GB RAM. Excluding the time required to import data, users can set up the entire system in less than 2 minutes.

5.2. Completeness

No data is lost during the create, read, update and delete process.

No.	Task	Execution Time (sec)
1.	Start baseline servers	117.513
2.	End baseline servers	2.699
3.	Import user collection	0.824
4.	Import article collection	0.726
5.	Import read collection	100.899
6.	Import images, video and text files	4983.178
7.	Add a new shard with two replica server	21.592
8.	Drop a server	3.179

Table 1 : Display execution time for each task.

6. Future Works

Currently our MongoDB cluster is in the development stage, and we haven't implemented its security features. In the future we plan to add authentication protocols that can protect the cluster from undesirable activities. Additionally, all the server instances in the cluster are managed by a continuously running script. This can be a single point of failure. In the future we would like to create a fully distributed system by replicating the instances of the script and making the system fully fault tolerant. Our cluster manager(the script) currently can't detect node failures, and our future goal is to create a self-healing cluster, that can detect failures and initialize new instances using replicated data in other instances. These are few of the improvements we plan to add to our system, and the design decisions we have followed so far will be instrumental in achieving this goal.

7. Conclusion

In this project, we developed a distributed database management system that can perform data fragmentation according to predefined shard keys. We designed a solution that is easy to use. Users can set up the entire system with only Python 3.x and MongoDB 4.0 above. For additional features, Studio 3T can be installed and it provides IDE and GUI such as visual query builder for non-technical users. All in all, our solution takes into consideration the usage of non-technical users.

8. Acknowledgement

We wish to thank Prof. Feng Ling and teaching assistant Lei Cao for the lectures and advice that help us complete this project successfully. Besides, we also thank our classmates Kai Wen, Florent, Malte and Barbara who provided additional help during the development of this project.

9. References

[1] Rana, M. S., Sohel, M. K., & Arman, M. S. (2018). Distributed Database Problems, Approaches and Solutions—A Study. In International Journal of Machine Learning and Computing (IJMLC).

[2] Vinoo Das. 2015. Learning Redis. Packt Publishing.

[3] Tauro, C., Easo, A., Mathew, S., & Gnesan, N. (2013, August). Convergent replicated data structures that tolerate eventual consistency in NoSQL databases. In 2013 Third International Conference on Advances in Computing and Communications (pp. 70-75). IEEE.

[4] M. Stonebraker, Operating System Support for Database Management, University of California, Berkeley.

[5] C. S. Kumar, J. Seetha, and S. R. Vinotha, "Security implications of distributed database management system models," International Journal of Soft Computing and Software Engineering, vol. 2, no. 11, 2012.

[6] Distributed DBMS-Replication Control (December 1, 2020). [Online]. Available: https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_replication_control.htm

[7] MongoDB Tutorials (December 1, 2020). [Online]. Available: https://docs.mongodb.com/manual/tutorial/

[8] PyMongo tutorial (December 13, 2020). [Online]. Available: https://pymongo.readthedocs.io/en/stable/tutorial.html

[9] Getting started with Studio 3T (December 11, 2020). [Online]. Available: https://studio3t.com/knowledge-base/articles/getting-started/