Distributed Database Management System
Final Project

Tsinghua University

December 24, 2019

Albert Millan – 2019280366
Bastien Bedu – 2019280421

Contents

1 Problem Background & Motivation

The advent of social media, real-time messaging and forum platforms has fostered the development of a highly interconnected society. Despite having friends and family a tap away from our smartphones, the levels of psychological distress among the population have been on the rise since these were first introduced. According to the World Health Organization (WHO), 20% of global teens are suffering from psychological distress, severely affecting countries such as China and Korea where, generally, the population tend to face higher levels of stress [1]. Consequently, we aim to develop a platform that, through a set of articles, invokes the interaction of distressed people in the areas of Beijing and Hong Kong, raising the awareness on the issue and aiding them to relieve anxiety and pressure.

The requirements that the system aims to fulfill are outlined below. At the very least, the system should be capable of:

- Loading bulk data into the table.
- Execute insert, update or delete queries.
- Evaluate the running status of the servers.

The paper is structured in the following manner. An analysis of the existing solutions concerning big data management systems is provided in **section 2**. It is followed by an explanation of the architecture under which the platform is supported, as well as a definition of its primary characteristics/features in **section 3**. Sample query operations that can be performed from the client are presented in **section 4**. **Section 5** explains the monitoring tools employed to evaluate the status of the system. An evaluation of the aforementioned requirements is shown in **section 6**. Lastly, concluding remarks are highlighted in **section 7**.

2 Existing Solutions

Traditionally, developers have had generally two options concerning data storage, namely relational databases and non-relational databases. Relational databases employ expressive query language

such as SQL and secondary indexes to define dependencies across structured models/schemas of the data. Contrarily, non-relational databases provide highly flexible and convenient manner to store data, emphasizing scalability and performance [2].

Corporate entities have further contributed to the development of characteristics from both relational and non-relational models. For instance, Google's Cloud Spanner efficiently combines a relational and non-relational features within its architecture, providing highly scalable data storage while also guaranteeing strong performance and consistency driven by MySQL. Notwithstanding, the the costs associated to setting-up and maintenance are unbearable to our budget. Amazon Neptune has also been considered. While it also provides convenient features that would enhance the development time, the costs associated are also very high. Using private database management systems is therefore not an option.

Achieving scale and distributive properties are an imperative need to reach a critical volume of consumers. Containerized applications in the cloud have proven effective in attaining such goal. These characterize for the decomposition into independently monitored components, each with a specific function within the system. Maintenance and fault tolerance are clearly enhanced, however, sophisticated technical knowledge is required to implement such systems within reasonable time restrictions. On the other hand, Hadoop also offers a robust solution with distributed features. It employs clusters of nodes to store the documents of data replicating and allocating data within the nodes conveniently. Performance-wise, it process queries in parallel across the nodes storing the information, thus being an efficient alternative. Furthermore, it provides integration with modern back-end development languages such as Nodejs and Django.

3 Proposed Solution

This section describes the underlying architecture of the proposed solution. In particular, it outlines the resources employed as well as the fragmentation and replication strategies derived to store the data in efficient manner. We strive to design a system architecture that assimilates to that in a production environment, despite working on a local environment. It consists of the following components:

- **Database Cluster**: MongoDB
- **Back-end Application Server**: Nodejs
- **API endpoint**: Postman

3.1 Database Cluster

We devise an architecture build on MongoDB, employing sharding and replication in order to optimize the overall performance of the system. The architecture employed is shown in figure 1, and is detailed in subsections 3.1.1, 3.1.2 and 3.1.3.

3.1.1 Sharding & Replication Strategy

The proposed sharding strategy involves segregating according to the volume and origin of the queries. That is, ensuring there is an even distribution of the data across the shards and an alignment with the location where they are performed. For instance, data that concerns the 'Beijing' area are stored in one node, referred as ***shard-1***, while data entries regarding 'Hong Kong' are stored in a different node, referred as ***shard-2***. Data entries involving both nodes are stored in a separate node, known as ***shard-3***. This is outlined in table 1.

Node	Desciption	Documents Stored
shard-1	Beijing Data	user:Beijing
shard-2	Hong Kong Data	user:Hong Kong
shard-3	Shared Data	article, reads

Table 1: Data distribution across shards.

To ensure the robustness and failure tolerance, we deploy a replica for each of the shards ***shard-1*** ***shard-2***, known as ***shard-1-repl*** and ***shard-2-repl***. Considering that **shard-3** stores a larger volume of entries and is accessed by both users from Hong Kong and Beijing, we employ two replicas, known as ***shard-3-repl-1*** and ***shard-3-repl-2***. The read preference configuration is set to target the primary nodes (***shard-1, shard-2 and shard-3***) first, and only target the replica nodes when the primary nodes are down. Such approach enables the execution of read operations even when, given unknown circumstances, one of the node servers shuts down, enhancing the resiliency of the

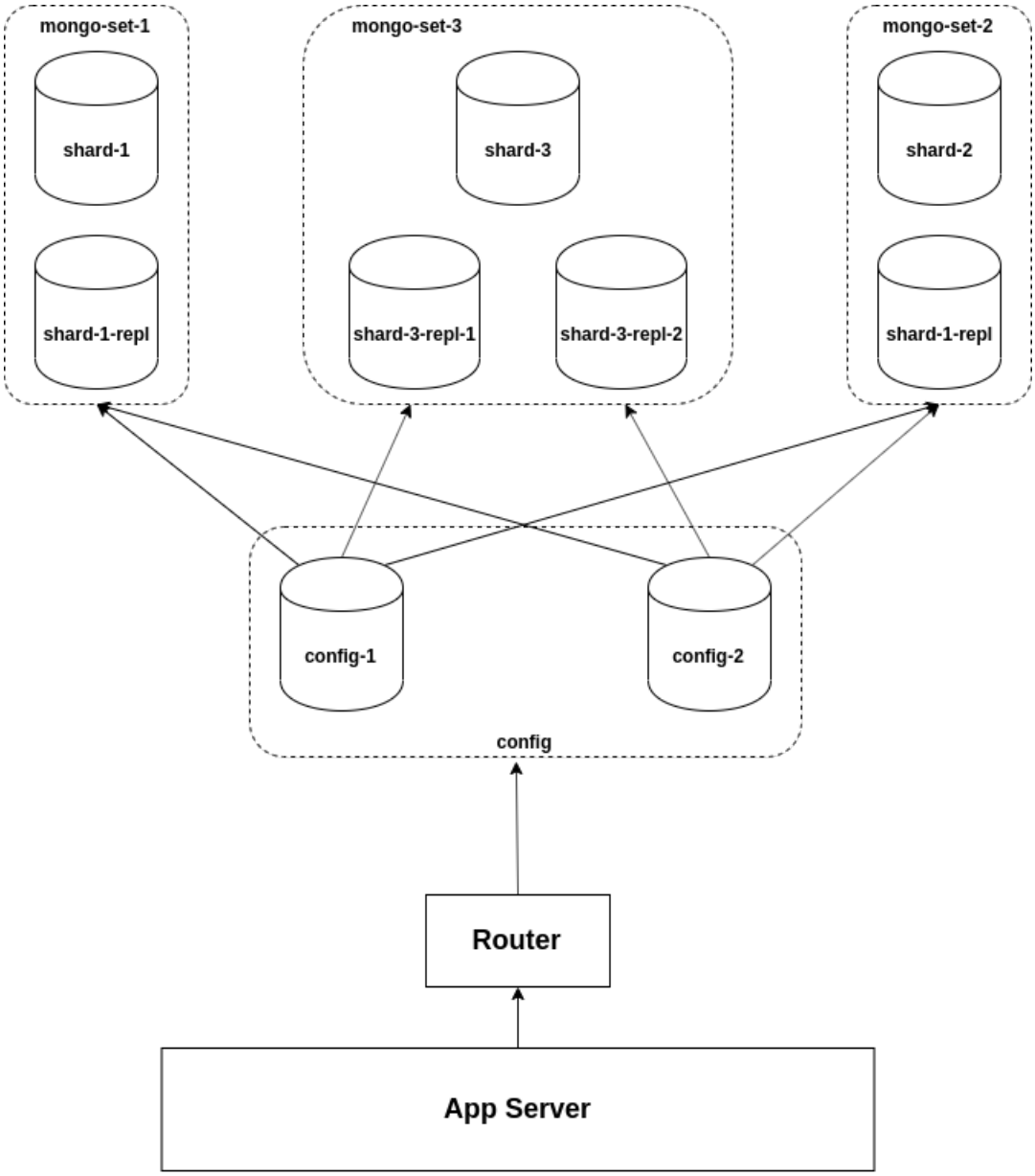

Figure 1: Architecture of the proposed soultion.

system. Therefore, as long as there is either one of the primary or secondary nodes actives, users can query the required information. A summary of the data stored in the replicas is provided in table 2.

Node	Desciption	Documents Stored
shard-1-repl	Beijing Data Copy	user:Beijing
shard-2-repl	Hong Kong Data Copy	user:Hong Kong
shard-3-repl-1	Shared Data Copy	article, reads
shard-3-repl-2	Shared Data Copy	article, reads

Table 2: Data distribution across replicas.

3.1.2 Cluster Configuration

Mongo stores the shard configuration metadata and settings for the cluster in separate nodes to those that have the data entry points. The metadata reflects the state and organization for the data stored in the sharded cluster, including a list of chunks on each shard and the ranges that define the chunks. It is recommended that each cluster has its own configuration file. Considering the locality of the data, we opt for two configuration servers composing a replica set in order to enhance resilience, referred as ***config-1*** and ***config-2***. If one of the configuration fails, the other would guarantee availability. A summary of the configuration servers is provided in table 3.

Node	Desciption	Documents Stored
config-1	Shard Data	Metadata
config-2	Shard Data	Metadata

Table 3: Data distribution across configuration servers.

3.1.3 Cluster Router: Data Access Point

The Mongo router server facilitates the distribution of queries to each shard accordingly. The router determines the list of shards that must receive the query[1] and subsequently establishes a cursor on all the the targeted shards. It then merges the data from each of the targeted shards into a single document. From developer's perspective, the key advantage is that the mongo router

[1]It identifies the shard where the data is located by caching the metadata from the config servers.

poses is the fact that the client does not necessarily need to comprehend the cluster structure to query data. Instead, it returns data as if it was all stored in a single database. Because we only use a single application server (refer to section 3.2) under a development environment, we only deploy a single mongo router, referred as ***mongos***. Notwithstanding, multiple routers are recommended as the number of application servers increases in order to support high availability and scalability. Table 4 summarizes the router configuration.

Node	**Desciption**	**IP Address**
mongos	Distribute Queries	127.0.0.1:27014

Table 4: Data distribution across router server.

3.2 Back-end Application Server

We employ a single application, Nodejs server to host the API, interfacing between the browser and database cluster. This server also hosts all the required media and text files. To access this files, the media file identifier is initially queried from the mongo database cluster, and successively used to return the file stored in the server. We consider this approach inefficient. Should a larger number of application servers be used, the media files would be replicated on each server. Write operations would have to replicated on all servers, overloading network traffic. The optimal solution would involve the use of an external server or hadoop nodes to store the data, and is further discussed in section 6.1. Due to the difficulties encountered setting-up the hadoop server and the associated time constraints, we opted for the aforementioned solution.

3.3 Client API

We use the browser to perform queries. For sample queries, refer to section 4.

4 Query Operations

This section outlines some sample queries that can be performed by the client. We include CRUD operations, including a join operation between users and the articles table to obtain the article information that the given user has read. This are shown in figures 2 and 3.

Figure 2: Sample basic queries based on user and read tables.

4.1 Be-Read Generation

Considering the data is dispersed across shards, we apply a map-reduce algorithm to generate the Be-Read table. Map-reduce operation is performed on each of the shards. The resulting objects are merged together, and successively perform the 'reduce' operation until the data in all shards has been reduced into a unique result. In other words, it is a distributed query operation running on multiple machines at the same time, reducing the required time to process large datasets. In our use case, we group by 'aid' (articles ID), and reduce by summing the number fields (commentNum, readNum, etc.), and appending ids to the corresponding fields (commentAidList, readAidList, etc.).

GET (http://localhost:3000/api/users/read?uid=12)


```
{
 "12": {
  "_id": "5e91c16c1a9a4b422a4b6287",
  "timestamp": "1506328059012",
  "id": "u12",
  "uid": "12",
  "name": "user12",
  "gender": "female",
  "email": "email12",
  "phone": "phone12",
  "dept": "dept2",
  "grade": "grade3",
  "language": "zh",
  "region": "Beijing",
  "role": "role2",
  "preferTags": "tags41",
  "obtainedCredits": "75",
  "read": [
   {
    "article": {
     "_id": "5e01c16e1a9a4b422a4bafb7",
     "id": "a9750",
     "timestamp": "1506000009750",
     "aid": "9750",
     "title": "title9750",
     "category": "science",
     "abstract": "abstract of article 9750",
     "articleTags": "tags4",
     "authors": "author643",
     "language": "zh",
     "text": "text_a9750.txt",
     "image": "image_a9750_0.jpg,image_a9750_1.jpg,image_a9750_2.jpg,image_a9750_3.jpg,",
     "video": ""
    },
    "read": {
     "_id": "5e01c1721a9a4b422a4ce157",
     "timestamp": "1507111347080",
     "id": "r77995",
     "uid": "12",
     "aid": "9750",
     "readOrNot": "1",
     "readTimeLength": "12",
     "readSequence": "2",
     "agreeOrNot": "0",
     "commentOrNot": "0",
     "shareOrNot": "0",
     "commentDetail": "comments to this article: (12,9750)"
    }
   },
```


Figure 3: Articles information and read information with user id:=12

4.2 Popular-Ranking

The Popular Rank is generated based on the Be-Read table. The previously computed statistics on the articles are used to simplify the trending[2] generation, as these values avoid redundant computations. We group the values based on the dates (year with month and day, or year with week number, or year with month). Then we sort each embedded element based on the number of agree (or like/up-votes).

5 Monitoring

The mechanisms utilized to monitor the status of the database are described in this section. It also describes the tool employed. The parameters considered include the state of the replicas and locks, memory usage, number of connections as well as disk utilization.

[2]Articles that have the largest amount of up-votes.

Figure 4: Rank based on user and read tables.

5.1 Mongostat

Mongostat is a built-in tool inside the mongo package. It provides real-time, detailed information concerning parameters such query operations, memory and disk storage, replica-set, node preference and status, etc. In addition, it can be exported into a JSON file, hence could be used for automatic resync or restart in case of failure.

host	insert	query	update	delete	getmore	command	dirty	used	flushes	vsize	res	qrw	arw	net_in	net_out	conn	set	repl	time
127.0.0.1:27018	*0	*0	*0	*0	0	2|0	0.1%	8.3%	0	1.81G	331M	0|0	1|0	**160b**	**48.0k**	24	mongo-set-1	PRI	**Dec 23 20:32:57.625**
127.0.0.1:27019	*0	*0	*0	*0	0	**3|0**	0.0%	8.3%	0	1.74G	343M	0|0	1|0	**326b**	**48.5k**	13	mongo-set-1	SEC	**Dec 23 20:32:57.628**
127.0.0.1:27020	*0	*0	*0	*0	0	**3|0**	0.0%	4.5%	**0**	1.65G	202M	0|0	1|0	**332b**	**48.3k**	23	mongo-set-2	PRI	**Dec 23 20:32:57.628**
127.0.0.1:27021	*0	*0	*0	*0	0	**2|0**	**0.0%**	4.5%	0	1.61G	206M	0|0	1|0	**160b**	**47.9k**	13	mongo-set-2	SEC	**Dec 23 20:32:57.628**
127.0.0.1:27022	*0	*0	*0	*0	0	2|0	**0.3%**	11.2%	**1**	1.90G	449M	0|0	1|0	**160b**	**47.8k**	26	mongo-set-3	PRI	**Dec 23 20:32:57.626**
127.0.0.1:27023	*0	*0	*0	*0	0	3|0	4.8%	11.4%	0	1.89G	463M	0|0	1|0	**326b**	**48.5k**	17	mongo-set-3	SEC	**Dec 23 20:32:57.626**
127.0.0.1:27024	*0	*0	*0	*0	0	4|0	3.2%	11.3%	0	1.84G	454M	0|0	1|0	**499b**	**49.0k**	14	mongo-set-3	SEC	**Dec 23 20:32:57.626**

Figure 5: Mongostat monitoring tool summary.

5.2 Free Monitoring

Mongo also provides a command to track the status of the cluster/router in the browser. It generates a token that is then used to identify the cluster and plot graphs about the status of the cluster. This tool can only be used with Mongo 3.6 or later versions. Unfortunately, there are no readily available docker containers containing such version of Mongo in the Chinese servers, hence we were

unable to use it. Notwithstanding, we illustrate a sample output in figure 6.

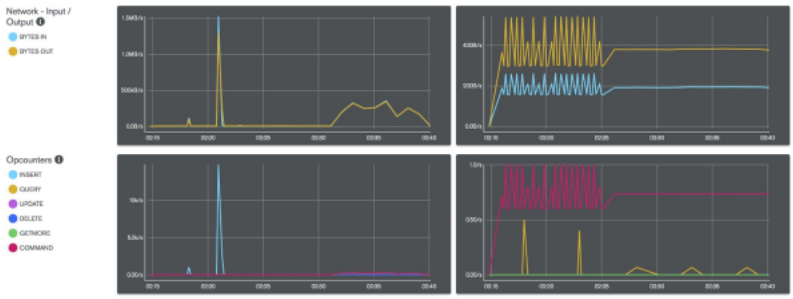

Figure 6: Mongo free monitoring tool summary.

6 Evaluation

In this section we asses the proposed solution relative to the project requirements. We also expand on the limitations of the current solution and potentiall areas of improvement.

1. **Bulk data loading with data partitioning and replica consideration.** The script outlined in appendix A sets up the docker containers, shards the database cluster, bulk loads the data and allocates it accordingly.

2. **Efficient execution of data read, insert and update operations**. Upon successfully sharding the cluster, queries can be executed from the from the Nodejs server to the mongo router.

3. **Monitoring the running status of the cluster.** Mongostat and Free Monitoring provide real-time relevant metrics concerning the status of the cluster.

4. **Advanced Features**:

   (a) **Dropping DBMS at will**. We set preferences within the replica set to prioritize primary nodes over secondary nodes. Notwithstanding, read operations will be performed on the secondary replica nodes should the primary node fail.

(b) **Data Migration from one data center to another**. Upon adding a new datacenter, data can be moved through the router by calling the moveChunk method (see Appendix A for more details).

6.1 Limitations

Storing the media files on each of the application servers adds a considerable burden to the network communication. Images have to be transferred upon new write operations. A significant improvement would involve storing the media and text data on a hadoop cluster, and progressively expand the number of nodes as the volume increases.

7 Conclusion

This project presented a fantastic learning experience. It challenged our capabilities and encouraged the practical application of the concepts explained in the lectures. Although our solution presented in this paper predominantly relies on Mongo as the storage system, we did try to implement hadoop, yet it had a steep learning curve and ran out of time. We resorted to Mongo and managed to fulfill all of the project requirements, including advanced ones. We acknowledged the limitations and explained potential improvement for future work.

References

[1] WHO, "Child and adolescent mental health," 2019.

[2] C. Győrödi, R. Győrödi, G. Pecherle, and A. Olah, "A comparative study: Mongodb vs. mysql," in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, pp. 1–6, June 2015.

[3] "Mongo db," 2019.

Appendix A Manual

In this section we present step-by-step guid to set-up the cluster locally.

A.1 Prerequisites

We assume the following applications are installed and available: docker and Node (npm).

- Docker is used for starting, sharding, replicating and bulkloading the data.
- Node is used to set-up the server back-end API.

A.2 Cluster Setup

1. Download the required docker images and create a docker network by running:

```
$ docker pull daocloud.io/mongo
$ docker network create xnet

```

2. Execute python dataset generation script:

```
$ python genTable_mongoDB10G.py

```

3. Move to the path containing the '.dat' generated document files and execute the following commands from the script 'Makefile' in this location. This should start-up all the mongo containers (shards, replicas, router and config servers). It should also perform sharding and replication operations on the router, as well as loading the data to the cluster.

```
$ make setup
$ make bulkload

```

A.3 Monitoring Setup

1. Ensure mongo is installed locally. Perform the following command to start the real-time cluster server status tracking:

```
$ mongostat -h 127.0.0.1:27018, 127.0.0.1:27019,
127.0.0.1:27020, 127.0.0.1:27021, 127.0.0.1:27022, 127.0.0.1:27023,
127.0.0.1:27024 --discover --interactive

```

2. Free monitoring is not available in the daocloud.io/mongo version, hence cannot be used. Should you upgrade to a later version, further information can be found in [3]

A.4 Application Server Setup

1. Go to the path where 'DDBS' folder is located, or otherwise, clone the following repository:

```
$ git clone https://github.com/Bastien-Bedu/DDBS.git

```

2. Install dependencies in the client and application server:

```
$ cd server
$ npm install

```

3. Run the server:

```
$ npm run dev

```

Server is ready and queries can now be performed!

A.5 Be-read & Popular-Rank Generation

1. Go back to the path where the github files were downloaded and execute the following command from the 'Makefile' to generate be-read and Popular rank collections:

```
$ cd ..
$ make beread
$ make popularrank

```

2. Queries can now be performed. Open up the browser and type the routes specified in 4.