Design and Implementation of a Distributed Database Management System Using Modern Technologies and Tools

Kembabazi Barbara Gamukama Yihong 2020280607

Yoke Kai Wen 2020280598 (Leader)

December 29, 2020

Abstract

This report summarizes the relevant knowledge covered in our DDBS module and presents the proposed design and implementation of a distributed database system built using MongoDB deployed on Docker containers. Through this project, we understand the reasons that pushed the evolution of NoSQL database platforms from SQL database platforms that had hitherto dominated the market. We specifically focus on some of the existing distributed database solutions, particularly highlighting their main features on enabling the integration of data among database management systems distributed at different sites. After reviewing existing solutions, we elucidate the problem definition which involves implementing a simplified distributed database system for a Douban-like social networking site. By applying key learning points from existing solutions to the given problem, we then decided to use MongoDB deployed on Docker containers to simulate a distributed database environment. We describe and explain in detail our command-line based implementation of our proposed solution, and finally conclude with future work that could be done. More details regarding the code implementation can be found in our github repostiory https://github.com/kwyoke/DDBS-2020, and also in the Appendix.

Keywords: distributed database systems, database management systems, integration, centralized

Contents

1 PROBLEM BACKGROUND & MOTIVATION

The increasing usage of web and mobile applications today has led to the overwhelming need for secure and stable database systems. Database Management Systems (DBMSs) are an ubiquitous and critical component of modern computing, and the result of decades of research and development in both academia and industry.[2] Database systems are a common setup for all types of applications. Today, database system tasks have evolved from simple data processing and data maintenance in an application into centralized administration systems of data in multiple applications. Centralization of database management systems has enabled data independence with their main motivation as the desire to integrate the operational data and provide controlled access to that data within the enterprise. However, these systems, being either the mainframe or server of an enterprise's application, must satisfy all requests received by the system, otherwise they would become a bottleneck in case of any malfunction or failures. It is important to note that the success of centralized database systems lies in the ability to integrate the data and not the centralization.

1.1 Distributed database

Integration can be achieved through the implementation of both computer network technology and database system management systems during the development of the enterprise's system i.e. a distributed database system. A distributed database is a collection of multiple,logically interrelated databases distributed over a computer network; A distributed database management system is a software system that permits the management of the distributed database and makes the distribution transparent to the users[4]. Distributed database systems are more reliable and responsive as compared to the traditional database system owing to the fundamental reason of the distributed processing which copes efficiently with large-scale data management problems using the divide-and-conquer rule.

1.2 Centralized vs distributed database

The main difference between the centralized database and distributed database is the fact that a centralized database stores data at a single location only whereas a distributed database system consists of multiple databases which are connected with each other and are spread across different physical locations. Therefore, when system failure occurs at centralized system, the entire data will be destroyed, while it will not be the case for distributed database systems. Thus, our drive to build a system that achieves integration of data without the need for centralization, thereby eliminating the bottleneck faced by the traditional database system.

2 EXISTING SOLUTIONS

The serious problem with centralized databases / data centers i.e. system failures happening due to power outages, cooling failures, network failures, or natural disasters, results in the need for distributing and integrating data while at the same time maintaining its high performance, durability and /transparency among users. The study of distributed databases has been going on for some years now and has led to the rise of new database related technologies such as use of non-relational databases and NoSQL, unlike traditional databases that used only SQL and existed as relational databases.

2.1 Apache Cassandra

Development of Apache Cassandra was born from these studies. It was one of the existing solutions to distributed databases that we studied and used as basis for comparison during the development of our model. Apache Cassandra is a database platform that resembles a normal database and shares many design and implementation strategies. However, it does not support a full relational data model, and instead provides clients with a simple data model that supports dynamic control over data layout and format. In other words, it brings out the integration of data without the need to centralize the databases. Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure.[3] Cassandra was developed at Facebook for the initial purpose of solving the problem of inbox search failure, thus it was constructed in a manner to treat failures as the norm rather than the exception. The core distributed systems techniques used in Cassandra are partitioning, replication, membership, failure handling and scaling of the data and it runs on three main commands *"insert", "get", "delete"*. [3] It partitions its data over a set of nodes and makes use of replicas, placing each into *N* hosts of each data item whereby *N* is a replication factor configured as the per-instance. There is a coordinator node in charge of the replication of the data items that fall within its range and replicates them into *N-1* nodes in its ring. Since all nodes are aware of every other node in the ring and know their ranges, and amongst them one is selected as a leader using the Zookeeper system, therefore when one crashes, it knows what range it was responsible for immediately when it comes back up. This scheme of replicating across multiple data centers allows Cassandra to handle entire data center failures without any outage.

2.2 Google Spanner

A study by Google on distributed database systems was the first system to distribute data at global scale and support externally-consistent distributed transactions[1]. This system is called Spanner, a globally-distributed database deployed at Google. Spanner also makes use of the replication factor as seen in the other studies. The replication configurations for data are said to be dynamically controlled at a fine grain by applications, and applications are able to set constraints on which data centers contain which data and the distance between users and data. The two unique features that Spanner executes are the ability to provide externally-consistent reads and writes, while at the same time provide globally consistent reads across all databases at a given timestamp. The ability to control concurrency is an important factor when dealing with distributed databases. The use of TrueTime API enables the above features to function since Spanner is able to keep globally-meaningful commit timestamps to transactions, when the transactions may be distributed; it is important to note that the TrueTime API is implemented by a set of time master machines per datacenter and a timeslave daemon per machine. The entire Spanner architecture mimics and implements the master-slave technique through its zones. A single deployment of a Spanner is called a universe which is made up of a set of zones. Each zone represents a unit of physical isolation; each having its own zone master and one hundred to thousand spanservers which are assigned data from the master and serve to the client. It deploys a universe master as the primary console that displays status information about all the zones for interactive debugging and a placement driver that handles automated movement of data across zones on the timescale of minutes. Replication and distributed transactions of data occurs within the spanservers through the implementation of a single Paxos state machine on top of each tablet(data structure), each set of replicas has a leader that also follows the master-slave architecture to deal with concurrency control during distributed transactions. [1]

2.3 Google F1

F1 is a distributed relational database system built at Google to support the AdWords business that was built on top of Spanner.[6] Both studies employ the master-slave architecture. However, F1 has a slave pool in each of its clusters and each cluster contains a spanner which can communicate with F1 servers outside its cluster. The shared slave pool consists of F1 processes that exist only to execute parts of distributed query plans on behalf of regular F1 servers. The entire system is based off the F1 servers / slaves and Spanner servers. The previous study on Spanner showed its utilization of replication and this is maintained as well in this study.

2.4 Key learning points form existing solutions

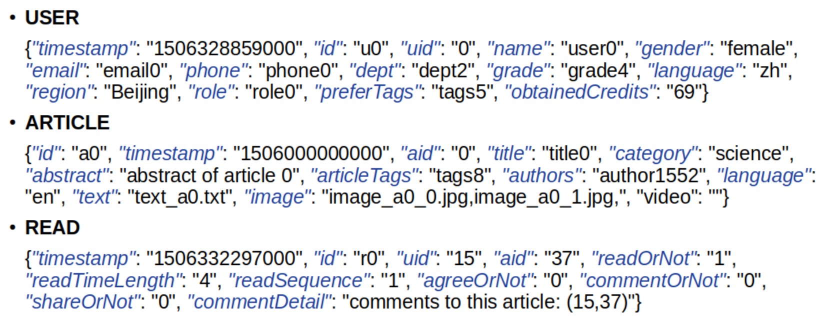
All existing solutions clarify the need for replication and partitioning of the databases / data items as a fault tolerance tactic. This is a task we aim to accomplish during the design and development of our system: to avoid system failure through the use of replicas.

3 PROBLEM DEFINITION

We are given 10GB of structured and unstructured data to be fragmented and allocated to various sites, which can then be efficiently modified and queried by users, with the entire distributed system being monitored by administrators. Essentially, the problem simulates the database component of a social networking application focused on article reading like Goodreads and Douban.

3.1 Data description

The structured data includes `user.dat, article.dat, read.dat` that contain tables recording structured information (see Figure 1), and the unstructured data comprises text, image and video data associated with each article.



- **USER**

  {"timestamp": "1506328859000", "id": "u0", "uid": "0", "name": "user0", "gender": "female", "email": "email0", "phone": "phone0", "dept": "dept2", "grade": "grade4", "language": "zh", "region": "Beijing", "role": "role0", "preferTags": "tags5", "obtainedCredits": "69"}
- **ARTICLE**

  {"id": "a0", "timestamp": "1506000000000", "aid": "0", "title": "title0", "category": "science", "abstract": "abstract of article 0", "articleTags": "tags8", "authors": "author1552", "language": "en", "text": "text_a0.txt", "image": "image_a0_0.jpg,image_a0_1.jpg,", "video": ""}
- **READ**

  {"timestamp": "1506332297000", "id": "r0", "uid": "15", "aid": "37", "readOrNot": "1", "readTimeLength": "4", "readSequence": "1", "agreeOrNot": "0", "commentOrNot": "0", "shareOrNot": "0", "commentDetail": "comments to this article: (15,37)"}

FIGURE 1: Example records in user, article, and read tables in JSON format

3.2 Populating Be-Read and Popular-Rank tables

Given the three tables (User, Article, Read) to be bulk-loaded into the data centre, we are to perform query, insert and join operations on them to populate two more tables: Be-Read and Popular-Rank.

1. **Be-Read** - Displays user interactions for each article. Attributes are *id, timestamp, aid, readNum, readUidList, commentNum, commentUidList,agreeNum, agreeUidList, shareNum, shareUidList*.
2. **Popular-Rank** - Ranks the top five most popular articles for daily, weekly and monthly time intervals. Attributes are *id, timestamp, temporalGranularity, articleAidList*.

The relationships between all five tables are demonstrated in Figure 2.

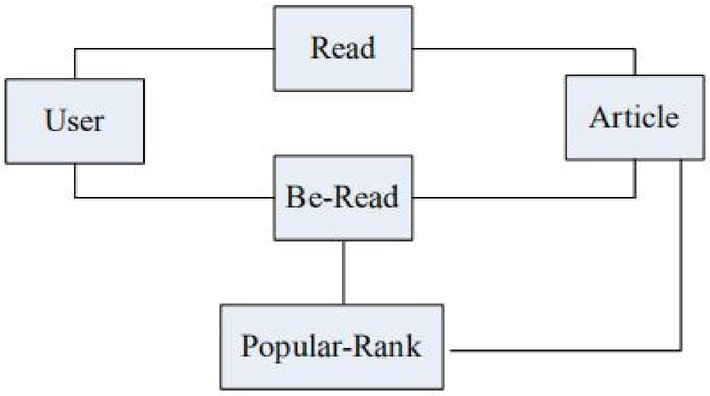


FIGURE 2: Relationships between five tables

3.3 Fragmentation and allocation requirements

We are to demonstrate fragmentation and allocation of structured data in a distributed environment using two DBMS sites, DBMS1 and DBMS2, according to the horizontal fragmentation and allocation scheme shown in Table 1. Unstructured data (text files, images, videos) are to be stored in Hadoop HDFS.

	Attribute	*DBMS1*	*DBMS2*
User	region	Beijing	Hong Kong
Article	category	science	science technology
Read	region	Beijing	Hong Kong
Be-Read	category	science	science technology
Popular-Rank	category	science	science technology

TABLE 1: Table showing fragmentation and allocation scheme of five tables

3.4 All requirements summarised

1. Bulk data loading with data partitioning and replica consideration.
2. Efficient execution of data insert, update and queries, including retrieval of unstructured multimedia data.
3. Monitoring running status of DBMS servers (including amount and location of data, workload etc).

4 PROPOSED SOLUTION

To fulfil the requirements set out in the previous section, we used Docker containers to simulate the different DBMS sites, and MongoDB, a NoSQL document database as our database system. To query and modify the database, the Mongo shell (command line) is used. All scripts and instructions required for implementing our solution can be found in our Github repository https://github.com/kwyoke/DDBS-2020.

4.1 Tools used in this project

In this section, we describe and justify our choice of software tools for this project.

4.1.1 Docker

Containerisation has become a major trend in software development, with Docker being the most representative provider of container technology. The Docker platform allows different applications to run in separated isolated environments called 'containers' on the same host machine. Furthermore, the containers are lightweight (compared to virtual machines) since they run directly on the host machine's kernel. This makes Docker containers a perfect choice for deploying our distributed database environment to simulate different servers, given our limited computing resources.

4.1.2 MongoDB

MongoDB is a non-relational NoSQL document database that provides support for JSON-like storage, and is amenable to both structured and unstructured data which are stored in unstructured collections instead of relational tables, and hence suitable for our use case. Furthermore, MongoDB comes with a whole suite of functionalities such as sharding (automatic distribution of data across different servers), replica sets (allowing hot/cold standby DBMSs), PyMongo (python library interface), GridFS (specification for storing and retrieving large files) and monitoring capabilities that greatly enhances the efficiency of our project implementation. Below, we describe in greater detail the relevant functionalities of MongoDB.

- **Sharding** - MongoDB allows automatic distribution of data to different servers (shard clusters) based on how the shard clusters are configured, in a process called 'sharding'. This automatic routing is carried out by the mongos router server that tracks what data is on which shard by caching the metadata from the config servers, and then using the metadata to route operations from applications and clients to the mongod instances (servers configured as shards).[5]

- **Replica sets** - MongoDB allows convenient set up of replica sets that are groups of mongod processes (servers) that maintain the same dataset. Replica sets provide redundancy and high availability. In the event of system failure in the primary server, the secondary server in the same replica set will become the primary server, hence making the database system tolerant to fault. [5]
- **PyMongo** - The PyMongo library allows us to work with MongoDB from Python, as we could not do everything within the Mongo shell and needed to write Python scripts to execute some of the required functions.
- **GridFS** - We were expected to use Hadoop HDFS to store the unstructured multimedia data, but we found MongoDB's GridFS to be a better substitute for Hadoop HDFS as it was much easier to use and interface with our MongoDB database. GridFS is a MongoDB specification for storing and retrieving large files such as images, audio files, video files within MongoDB collections, and hence suitable for storing the large image and video files generated from the 10GB data.

4.2 Distributed system architecture

We set up 8 docker containers in total (see Figure 3): 3 config servers in a replica set; 1 mongos router server; 2 DBMS servers shard, each being a single server replica set, for distributing structured data; 2 GridFS server shards, each being a single server replica set, for distributing multimedia data. This can be achieved by running our docker-compose files in our Github repository https://github.com/kwyoke/DDBS-2020, which sets up each Docker container as a mongod or mongos instance based on their respective purpose and configure them as shards. The output on the command line is shown in Figure 4. For greater availability and fault tolerance, we should have replica sets of at least three servers for our DBMS and GridFS shards, but due to lack of computer space, we decided to stick with only single server shards, and demonstrate the replica set concept via the config servers.

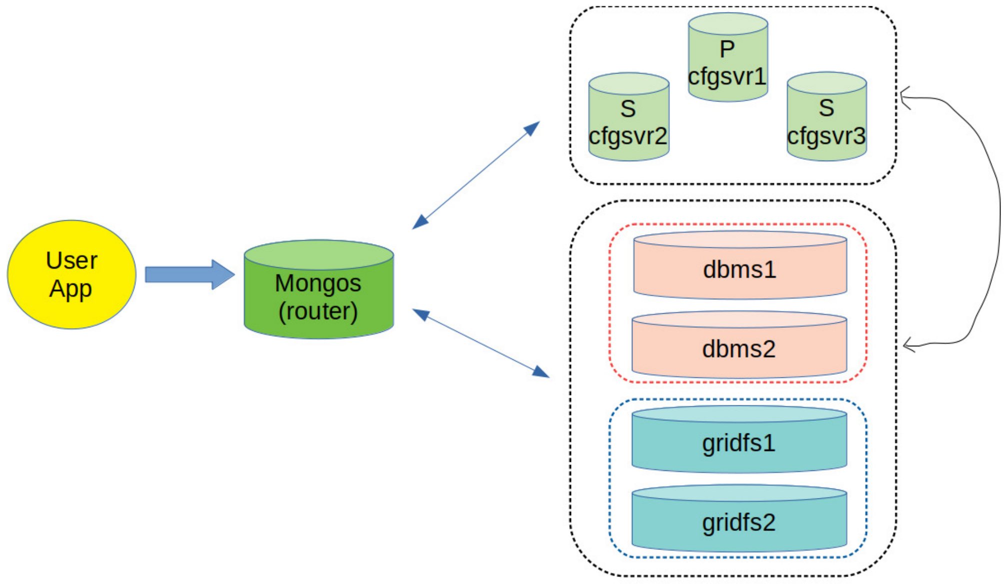
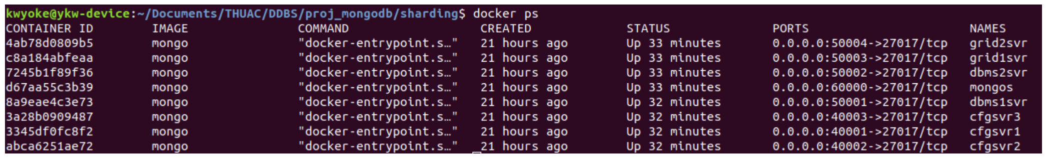


FIGURE 3: Architecture of proposed solution: distributed database system with MongoDB deployed on Docker containers



```
kwyoke@ykw-device:~/Documents/THUAC/DDBS/proj_mongodb/sharding$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                        NAMES
4ab78d0809b5        mongo               "docker-entrypoint.s…"   21 hours ago        Up 33 minutes       0.0.0.0:50004->27017/tcp     grid2svr
c8a184abfeaa        mongo               "docker-entrypoint.s…"   21 hours ago        Up 33 minutes       0.0.0.0:50003->27017/tcp     grid1svr
7245b1f89f36        mongo               "docker-entrypoint.s…"   21 hours ago        Up 33 minutes       0.0.0.0:50002->27017/tcp     dbms2svr
d67aa55c3b39        mongo               "docker-entrypoint.s…"   21 hours ago        Up 33 minutes       0.0.0.0:60000->27017/tcp     mongos
8a9eae4c3e73        mongo               "docker-entrypoint.s…"   21 hours ago        Up 32 minutes       0.0.0.0:50001->27017/tcp     dbms1svr
3a28b0909487        mongo               "docker-entrypoint.s…"   21 hours ago        Up 32 minutes       0.0.0.0:40003->27017/tcp     cfgsvr3
3345df0fc8f2        mongo               "docker-entrypoint.s…"   21 hours ago        Up 32 minutes       0.0.0.0:40001->27017/tcp     cfgsvr1
abca6251ae72        mongo               "docker-entrypoint.s…"   21 hours ago        Up 32 minutes       0.0.0.0:40002->27017/tcp     cfgsvr2
```

FIGURE 4: Output on command line after setting up docker containers

Figure 3 also shows the relationships between each server. The config servers store metadata on the volume and location of data in the DBMS and GridFS shards, which is accessed by the mongos router for routing operations from applications to the DBMS and GridFS shards.

4.3 Bulk loading

The `user.dat`, `article.dat`, `read.dat` data files are copied into the mongos docker container, and then imported into the mongos server's `ddbs` database using the `mongoimport` command.

4.4 Sharding

Sharding of collections is quite simple in MongoDB. There are two types of sharding: ranged and hashed. Ranged sharding allows us to assign documents of specific shard key values to specific target shards, while hashed sharding assigns documents more evenly (at random) based on hashed index. For this project, we mainly used ranged sharding because we have to direct documents based on a specific fragmentation rule (see Table 1). For the unstructured multimedia data, we employed hashed sharding as we are not concerned over which GridFS shard (grid1, grid2) the multimedia data chunks go to. Once we set the shard keys of each collection and configure the shard zones of each shard, the balancer running in the background will automatically distribute the database documents to the respective shard. These processes are completely transparent to the user who can simply interact with the mongos server as if it was a centralised database.

4.4.1 Summary of main steps for enabling and configuring sharding

1. Enable sharding for the target database.
2. Create index for collection to be sharded, and this index will be used as shard key. Shard key can be made up of multiple fields, and is critical for even distribution of data across shards.
3. Enable sharding for the collection, and set the collection index as shard key.
4. Configure the sharding zones (zone tags and zone range) for each shard, specific to the collection. This determines where each document in the collection will be assigned to.
5. Ensure that the balancer is running for that collection.

4.4.2 Limitations of sharding in MongoDB

While sharding in MongoDB is very easy to implement, it does not fulfil some of our requirements.

1. **Assigning same data to both shards** - According to the allocation scheme in Table 1, we are supposed to assign science articles to both DBMS1 and DBMS2. However, in MongoDB, sharding does not allow two shards to

contain the same data, as all replica considerations are meant to be accounted for by the replica sets. To work around this, we copied `article` to another collection `articlesci` for just science articles, and assigned it to DBMS2. The same problem arose for `beread` and `popRankSci`, which were tackled in the same manner. See Table 2 for more details.

2. **Collection must contain fields from shard key** - Another issue is that sharding of a collection requires the collection to contain fields from the shard key. For instance, since we are required to shard `read` according to the "region" field in the `user` collection, we must also include the "region" field in the `read` table so that it can be sharded according to "region". Unfortunately, this adds extra overhead as we now have to store one more extra field. The same problem also arose for `beread, bereadsci` which requires us to include an extra "category" field for these two collections.

Taking into consideration sharding and its limitations in MongoDB, while trying to stick to the problem specification as closely as possible, we finally fragmented and allocated our collections in the scheme shown in Table 2.

Collection	*Shard key*	*DBMS1 zone tag*	*DBMS2 zone tag*
user	{region, uid}	BJ	HK
article	{category, aid}	SCI	TECH
articlesci	{category, aid}	-	SCI2
read	{region, id}	BJ	HK
beread	{category, aid}	SCI	TECH
bereadsci	{category, aid}	-	SCI2
popRank	{_id}	-	POPALL
popRankSci	{_id}	POPSCI	-
popRankSci2	{_id}		POPSCI2
popRankTech	{_id}	-	TECH

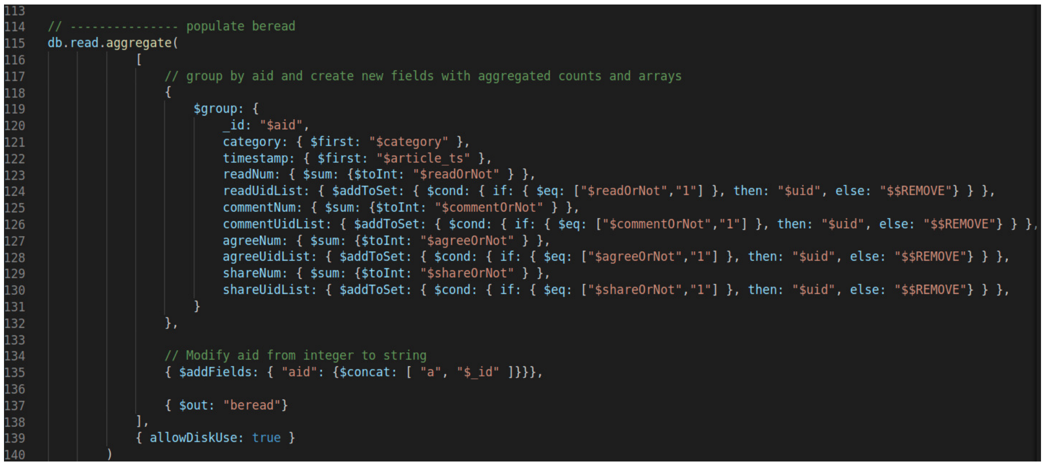
TABLE 2: Table showing sharding configuration for each collection

4.5 MongoDB aggregation pipeline

Aggregation processes multiple data documents and return computed records. MongoDB provides an aggregation pipeline that we use to populate the `beread` and `popRank` collections.

4.5.1 Populating `beread`

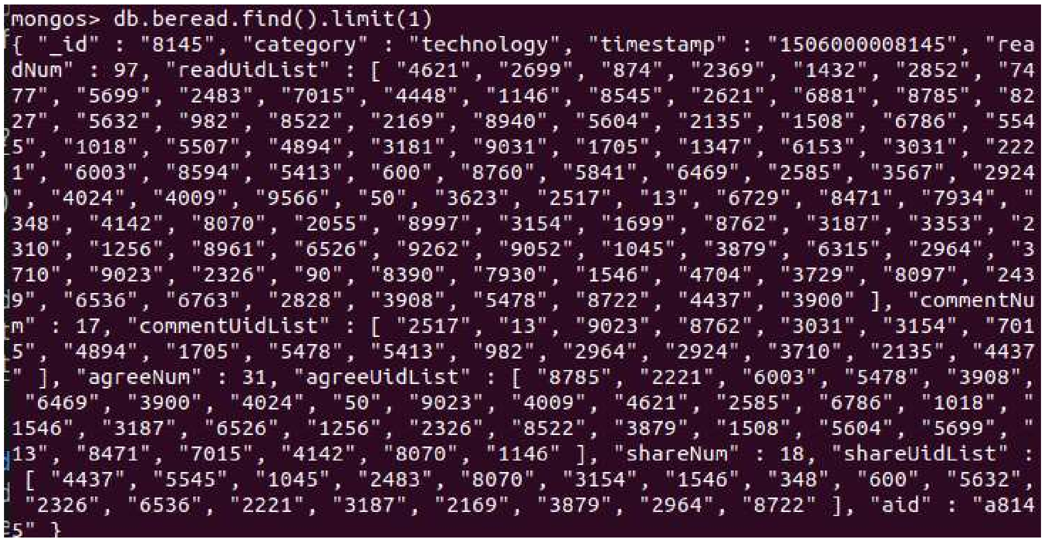
`beread` is formed from aggregating `read` - the `read` documents are grouped by the "aid" field, and then the number of reads, agrees, shares and comments are counted for each "aid" group, together with the relevant list of "uid"s involved in these interactions, and then finally output as the `beread` collection. The exact mongo shell command for the aggregation pipeline is shown in Figure 5. Note that before the aggregation, we already added the article category (and article timestamp) field to `read` to facilitate subsequent sharding for `beread`.



```

// -------------- populate beread
db.read.aggregate(
            [
                // group by aid and create new fields with aggregated counts and arrays
                {
                    $group: {
                        _id: "$aid",
                        category: { $first: "$category" },
                        timestamp: { $first: "$article_ts" },
                        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"} } },
                    }
                },

                // Modify aid from integer to string
                { $addFields: { "aid": {$concat: [ "a", "$_id" ]}}},

                { $out: "beread"}
            ],
            { allowDiskUse: true }
        )
```

FIGURE 5: Aggregation pipeline to populate `beread` collection

Figure 6 shows a sample document from the populated `beread` collection.

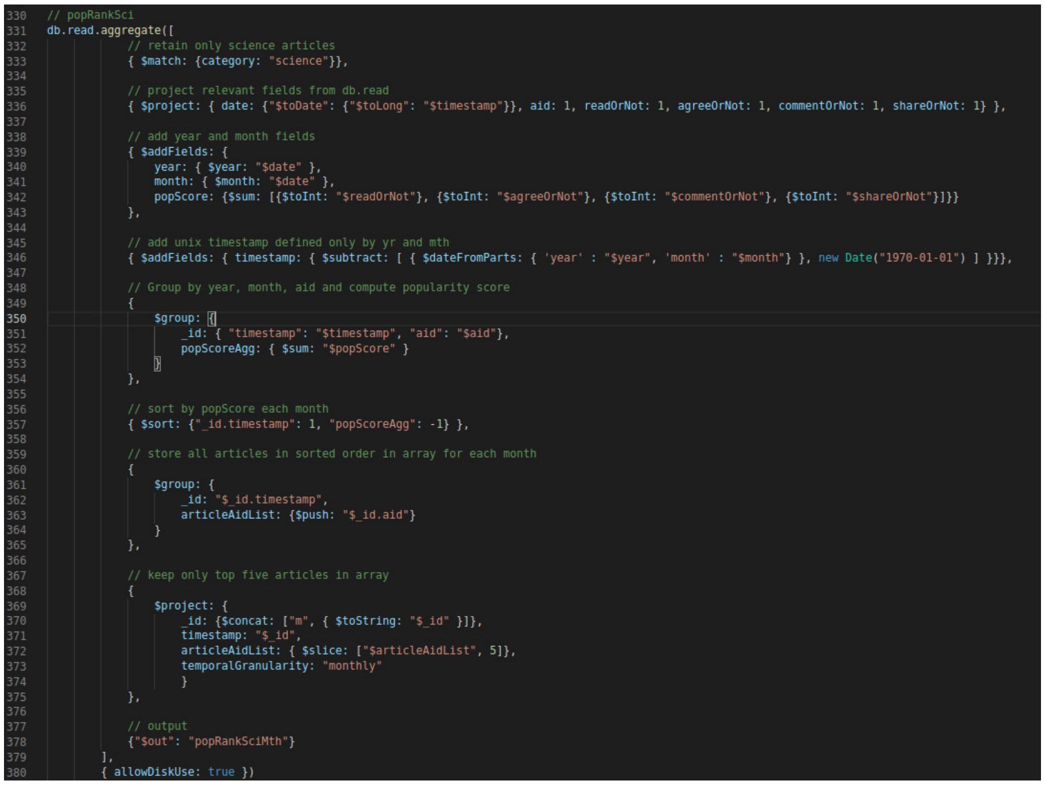


```
mongos> db.beread.find().limit(1)
{ "_id" : "8145", "category" : "technology", "timestamp" : "1506000008145", "rea
dNum" : 97, "readUidList" : [ "4621", "2699", "874", "2369", "1432", "2852", "74
77", "5699", "2483", "7015", "4448", "1146", "8545", "2621", "6881", "8785", "82
27", "5632", "982", "8522", "2169", "8940", "5604", "2135", "1508", "6786", "554
5", "1018", "5507", "4894", "3181", "9031", "1705", "1347", "6153", "3031", "222
1", "6003", "8594", "5413", "600", "8760", "5841", "6469", "2585", "3567", "2924
", "4024", "4009", "9566", "50", "3623", "2517", "13", "6729", "8471", "7934", "
348", "4142", "8070", "2055", "8997", "3154", "1699", "8762", "3187", "3353", "2
310", "1256", "8961", "6526", "9262", "9052", "1045", "3879", "6315", "2964", "3
710", "9023", "2326", "90", "8390", "7930", "1546", "4704", "3729", "8097", "243
9", "6536", "6763", "2828", "3908", "5478", "8722", "4437", "3900" ], "commentNu
m" : 17, "commentUidList" : [ "2517", "13", "9023", "8762", "3031", "3154", "701
5", "4894", "1705", "5478", "5413", "982", "2964", "2924", "3710", "2135", "4437
" ], "agreeNum" : 31, "agreeUidList" : [ "8785", "2221", "6003", "5478", "3908",
 "6469", "3900", "4024", "50", "9023", "4009", "4621", "2585", "6786", "1018", "
1546", "3187", "6526", "1256", "2326", "8522", "3879", "1508", "5604", "5699", "
13", "8471", "7015", "4142", "8070", "1146" ], "shareNum" : 18, "shareUidList" :
 [ "4437", "5545", "1045", "2483", "8070", "3154", "1546", "348", "600", "5632",
 "2326", "6536", "2221", "3187", "2169", "3879", "2964", "8722" ], "aid" : "a814
5" }
```

FIGURE 6: Sample document from `beread` collection

4.5.2 Populating popRank

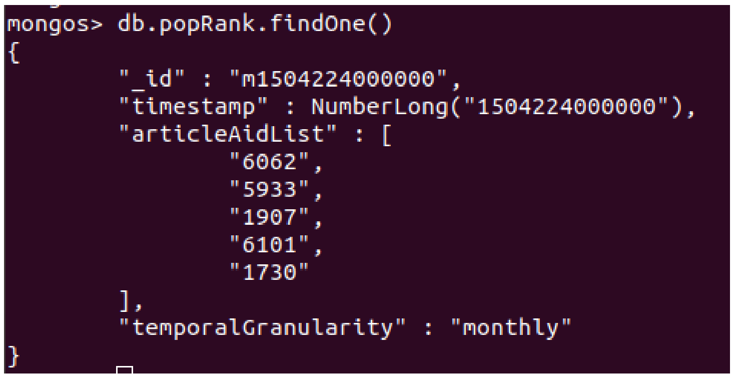
`popRank, popRankSci, popRankTech` collections are formed from multiple aggregations and concatenations from the `read` collection. We have three collections for Popular-Rank to represent the most popular articles for both categories, for science category only and for technology articles only. For each `popRank` collection, we perform three aggregations, one for monthly, one for weekly and one for daily intervals. Each aggregation outputs a temporary collection, e.g. `popRankMth, popRankWk, popRankDay` which are then concatenated together to form `popRank`. In each aggregation pipeline, we extract the year, month, week, day information from the timestamp, group the documents by the respective interval (month, week or day), and then sum up all the interactions (read, agree, share, comment) for each article in each group to form a popularity score. Within each group, we then sort and rank the articles by their popularity score, and select the top five articles and push them into an array of field "articleAidList", finally outputting a temporary collection for that particular time interval (e.g. monthly). See Figure 7 for the full mongo shell command to aggregate and form `popRankSciMth`.



```
// popRankSci
db.read.aggregate([
            // retain only science articles
            { $match: {category: "science"}},

            // project relevant fields from db.read
            { $project: { date: {"$toDate": {"$toLong": "$timestamp"}}, aid: 1, readOrNot: 1, agreeOrNot: 1, commentOrNot: 1, shareOrNot: 1} },

            // add year and month fields
            { $addFields: {
                year: { $year: "$date" },
                month: { $month: "$date" },
                popScore: {$sum: [{$toInt: "$readOrNot"}, {$toInt: "$agreeOrNot"}, {$toInt: "$commentOrNot"}, {$toInt: "$shareOrNot"}]}}
            },

            // add unix timestamp defined only by yr and mth
            { $addFields: { timestamp: { $subtract: [ { $dateFromParts: { 'year' : "$year", 'month' : "$month"} }, new Date("1970-01-01") ] }}},

            // Group by year, month, aid and compute popularity score
            {
                $group: {
                    _id: { "timestamp": "$timestamp", "aid": "$aid"},
                    popScoreAgg: { $sum: "$popScore" }
                }
            },

            // sort by popScore each month
            { $sort: {"_id.timestamp": 1, "popScoreAgg": -1} },

            // store all articles in sorted order in array for each month
            {
                $group: {
                    _id: "$_id.timestamp",
                    articleAidList: {$push: "$_id.aid"}
                }
            },

            // keep only top five articles in array
            {
                $project: {
                    _id: {$concat: ["m", { $toString: "$_id" }]},
                    timestamp: "$_id",
                    articleAidList: { $slice: ["$articleAidList", 5]},
                    temporalGranularity: "monthly"
                    }
            },

            // output
            {"$out": "popRankSciMth"}
        ],
        { allowDiskUse: true })
```

FIGURE 7: Aggregation pipeline to populate `popRankSciMth` (temporary collection)

Figure 8 shows a sample document from the populated collection `popRank`.



```
mongos> db.popRank.findOne()
{
        "_id" : "m1504224000000",
        "timestamp" : NumberLong("1504224000000"),
        "articleAidList" : [
                "6062",
                "5933",
                "1907",
                "6101",
                "1730"
        ],
        "temporalGranularity" : "monthly"
}
```

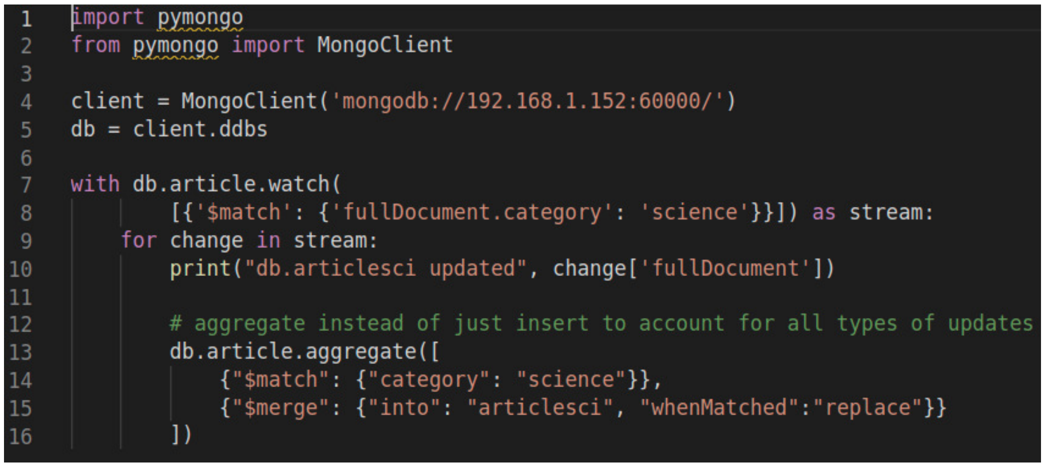
FIGURE 8: Sample document from popRank

4.6 MongoDB CRUD operations

CRUD operations like data insert, update and queries can be performed efficiently on our distributed database system, just like a centralised database due to MongoDB's sharding functionality. However, the issue arises when we want to reflect any new changes on for instance `read` to `beread` and `popRank` which are derived from the `read` collection. This is achieved using MongoDB's `db.collection.watch()` function which allows us to watch for changes in the `read` collection and subsequently update the `beread` and `popRank` collections accordingly. We implement this using a python script running in the background.

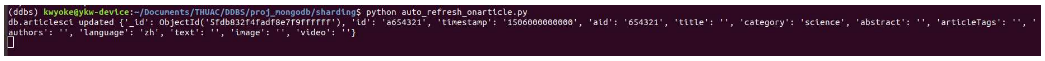
4.6.1 Python script to automatically refresh collections upon CRUD changes

In our Github Repository https://github.com/kwyoke/DDBS-2020, we include python scripts to watch and automatically refresh collections `auto_refresh_-onread.py, auto_refresh_onarticle.py`. The python scripts leverage the PyMongo library to interface with MongoDB, watch out for CRUD changes in the target collections, and update derived collections accordingly. A code snippet demonstrating PyMongo and `db.collection.watch()` is shown in Figure 9 and the terminal output when the python script is running is shown in Figure 10. The required collections to watch and refresh are summarised in Figure 11.



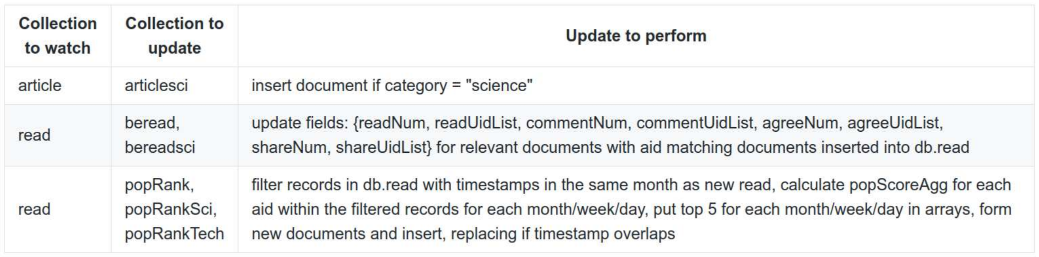
```python
import pymongo
from pymongo import MongoClient

client = MongoClient('mongodb://192.168.1.152:60000/')
db = client.ddbs

with db.article.watch(
        [{'$match': {'fullDocument.category': 'science'}}]) as stream:
    for change in stream:
        print("db.articlesci updated", change['fullDocument'])

        # aggregate instead of just insert to account for all types of updates
        db.article.aggregate([
            {"$match": {"category": "science"}},
            {"$merge": {"into": "articlesci", "whenMatched":"replace"}}
        ])
```

FIGURE 9: Code fragment: leveraging PyMongo and db.collection.watch() to update collections



```
(ddbs) kwyoke@ykw-device:~/Documents/THUAC/DDBS/proj_mongodb/sharding$ python auto_refresh_onarticle.py
db.articlesci updated {'_id': ObjectId('5fdb832f4fadf8e7f9ffffff'), 'id': 'a654321', 'timestamp': '1506000000000', 'aid': '654321', 'title': '', 'category': 'science', 'abstract': '', 'articleTags': '', '
authors': '', 'language': 'zh', 'text': '', 'image': '', 'video': ''}
```

FIGURE 10: PyMongo running in terminal and updating `articlesci` collection when changes to `article` are detected



Collection to watch	Collection to update	Update to perform
article	articlesci	insert document if category = "science"
read	beread, bereadsci	update fields: {readNum, readUidList, commentNum, commentUidList, agreeNum, agreeUidList, shareNum, shareUidList} for relevant documents with aid matching documents inserted into db.read
read	popRank, popRankSci, popRankTech	filter records in db.read with timestamps in the same month as new read, calculate popScoreAgg for each aid within the filtered records for each month/week/day, put top 5 for each month/week/day in arrays, form new documents and insert, replacing if timestamp overlaps

FIGURE 11: List of refreshes to perform upon change in article, read collections

4.7 Storing of unstructured data in GridFS

The articles come with multimedia data that are not stored directly on our MongoDB DBMS servers due to their large size. Instead, our MongDB DBMS servers only store the filenames of the multimedia data, which are then used to retrieve the image data from other storage servers (grid1, grid2) dedicated for multimedia data, which use the GridFS specification. We found GridFS to be a better substitute than Hadoop HDFS as they perfor the same function but it is much easier to interface GridFS with our MongoDB database system.

4.7.1 GridFS and sharding

We have two shard servers, grid1 and grid2 dedicated to storing multimedia data of articles. Before loading the multimedia to the data, we first set up sharding so that the (10GB) worth of data will be distributed to the two shards as it is uploaded to GridFS. GridFS is a specification for storing and retrieving large files of more than 16MB which is perfect for our use case in storing image, video and text files data. We apply hashed sharding here to allow more even distribution of multimedia data across the two shard servers.

GridFS automatically stores the data files in db.fs.chunks collection, and the filenames in db.fs.files collection. We only shard the db.fs.chunks collection since it is very large, while db.fs.files is not be sharded and remain on the mongos server.

4.7.2 Loading and retrieving files from GridFS

To load and retrieve files to/from GridFS, we use the `mongofiles` utility which allows us to manipulate GridFS objects from the commnad line.

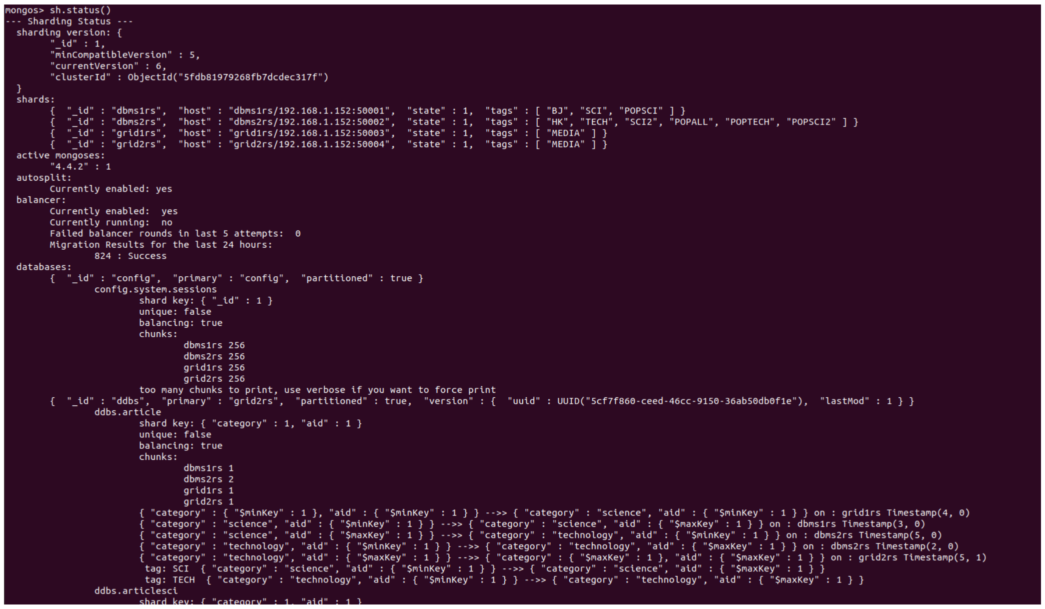
```
# load image file into GridFS collection stored in ddbs
mongofiles --host=192.168.1.152:60000 --local=image_a0_0.jpg -d=
    ddbs put image_a0_0.jpg

#retrieve image file from GridFS into local computer
mongofiles --host=192.168.1.152:60000 -d=ddbs get image_a0_0.jpg
xdg-open image_a0_0.jpg
```

4.8 Monitoring

MongoDB provides monitoring tools that allows administrators to easily monitor the running status of the DBMS servers, including its managed data (amount and location) as well as workload.

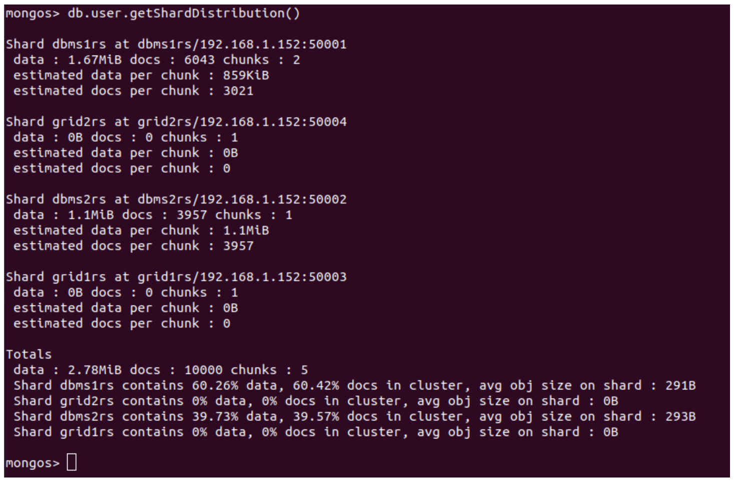
4.8.1 Monitoring managed data

Simply type `sh.status()` and `db.collection.getShardDistribution()` in the mongo shell of the mongos router container to observe the status of the various connected shard clusters (see Figure 12) and the distribution of documents in that collection (see Figure 13) respectively.



```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5fdb81979268fb7dcdec317f")
  }
  shards:
        {  "_id" : "dbms1rs",  "host" : "dbms1rs/192.168.1.152:50001",  "state" : 1,  "tags" : [ "BJ", "SCI", "POPSCI" ] }
        {  "_id" : "dbms2rs",  "host" : "dbms2rs/192.168.1.152:50002",  "state" : 1,  "tags" : [ "HK", "TECH", "SCI2", "POPALL", "POPTECH", "POPSCI2" ] }
        {  "_id" : "grid1rs",  "host" : "grid1rs/192.168.1.152:50003",  "state" : 1,  "tags" : [ "MEDIA" ] }
        {  "_id" : "grid2rs",  "host" : "grid2rs/192.168.1.152:50004",  "state" : 1,  "tags" : [ "MEDIA" ] }
  active mongoses:
        "4.4.2" : 1
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                824 : Success
  databases:
        {  "_id" : "config",  "primary" : "config",  "partitioned" : true }
                config.system.sessions
                        shard key: { "_id" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                dbms1rs 256
                                dbms2rs 256
                                grid1rs 256
                                grid2rs 256
                        too many chunks to print, use verbose if you want to force print
        {  "_id" : "ddbs",  "primary" : "grid2rs",  "partitioned" : true,  "version" : {  "uuid" : UUID("5cf7f860-ceed-46cc-9150-36ab50db0f1e"),  "lastMod" : 1 } }
                ddbs.article
                        shard key: { "category" : 1, "aid" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                dbms1rs 1
                                dbms2rs 2
                                grid1rs 1
                                grid2rs 1
                        { "category" : { "$minKey" : 1 }, "aid" : { "$minKey" : 1 } } -->> { "category" : "science", "aid" : { "$minKey" : 1 } } on : grid1rs Timestamp(4, 0)
                        { "category" : "science", "aid" : { "$minKey" : 1 } } -->> { "category" : "science", "aid" : { "$maxKey" : 1 } } on : dbms1rs Timestamp(3, 0)
                        { "category" : "science", "aid" : { "$maxKey" : 1 } } -->> { "category" : "technology", "aid" : { "$minKey" : 1 } } on : dbms2rs Timestamp(5, 0)
                        { "category" : "technology", "aid" : { "$minKey" : 1 } } -->> { "category" : "technology", "aid" : { "$maxKey" : 1 } } on : dbms2rs Timestamp(2, 0)
                        { "category" : "technology", "aid" : { "$maxKey" : 1 } } -->> { "category" : { "$maxKey" : 1 }, "aid" : { "$maxKey" : 1 } } on : grid2rs Timestamp(5, 1)
                         tag: SCI  { "category" : "science", "aid" : { "$minKey" : 1 } } -->> { "category" : "science", "aid" : { "$maxKey" : 1 } }
                         tag: TECH  { "category" : "technology", "aid" : { "$minKey" : 1 } } -->> { "category" : "technology", "aid" : { "$maxKey" : 1 } }
                ddbs.articlesci
                        shard key: { "category" : 1, "aid" : 1 }
```

FIGURE 12: Output of `sh.status()`

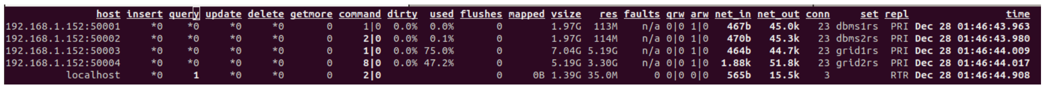


```
mongos> db.user.getShardDistribution()

Shard dbms1rs at dbms1rs/192.168.1.152:50001
 data : 1.67MiB docs : 6043 chunks : 2
 estimated data per chunk : 859KiB
 estimated docs per chunk : 3021

Shard grid2rs at grid2rs/192.168.1.152:50004
 data : 0B docs : 0 chunks : 1
 estimated data per chunk : 0B
 estimated docs per chunk : 0

Shard dbms2rs at dbms2rs/192.168.1.152:50002
 data : 1.1MiB docs : 3957 chunks : 1
 estimated data per chunk : 1.1MiB
 estimated docs per chunk : 3957

Shard grid1rs at grid1rs/192.168.1.152:50003
 data : 0B docs : 0 chunks : 1
 estimated data per chunk : 0B
 estimated docs per chunk : 0

Totals
 data : 2.78MiB docs : 10000 chunks : 5
 Shard dbms1rs contains 60.26% data, 60.42% docs in cluster, avg obj size on shard : 291B
 Shard grid2rs contains 0% data, 0% docs in cluster, avg obj size on shard : 0B
 Shard dbms2rs contains 39.73% data, 39.57% docs in cluster, avg obj size on shard : 293B
 Shard grid1rs contains 0% data, 0% docs in cluster, avg obj size on shard : 0B

mongos> 
```

FIGURE 13: Output of `db.user.getShardDistribution()`

4.8.2 Monitoring workload

MongoDB also facilitates the monitoring of operation execution times, memory usage, CPU usage and operation counts. By using the `mongostat` utility in the command line, we can observe in real-time the number of operations of each server (see Figure 14).



```
               host insert query update delete getmore command dirty  used flushes mapped vsize   res faults qrw arw net_in net_out conn     set repl                time
192.168.1.152:50001     *0    *0     *0     *0       0     1|0  0.0%  0.0%       0        1.97G  113M    n/a 0|0 1|0   467b   45.0k   23 dbms1rs  PRI Dec 28 01:46:43.963
192.168.1.152:50002     *0    *0     *0     *0       0     2|0  0.0%  0.1%       0        1.97G  114M    n/a 0|0 1|0   470b   45.3k   23 dbms2rs  PRI Dec 28 01:46:43.980
192.168.1.152:50003     *0    *0     *0     *0       0     1|0  0.0% 75.0%       0        7.04G 5.19G    n/a 0|0 1|0   464b   44.7k   23 grid1rs  PRI Dec 28 01:46:44.009
192.168.1.152:50004     *0    *0     *0     *0       0     8|0  0.0% 47.2%       0        5.19G 3.30G    n/a 0|0 1|0  1.88k   51.8k   23 grid2rs  PRI Dec 28 01:46:44.017
          localhost     *0     1     *0     *0       0     2|0                   0     0B 1.39G 35.0M      0 0|0 0|0   565b   15.5k    3          RTR Dec 28 01:46:44.908
```

FIGURE 14: Output of `mongostat`

Unfortunately, the Docker version of MongoDB does not allow free monitoring at the moment, but could be provided in the near future. In any case, we demonstrate that MongoDB does allow for monitoring of memory usage, CPU utilisation etc using a browser interface as shown in Figure 15.

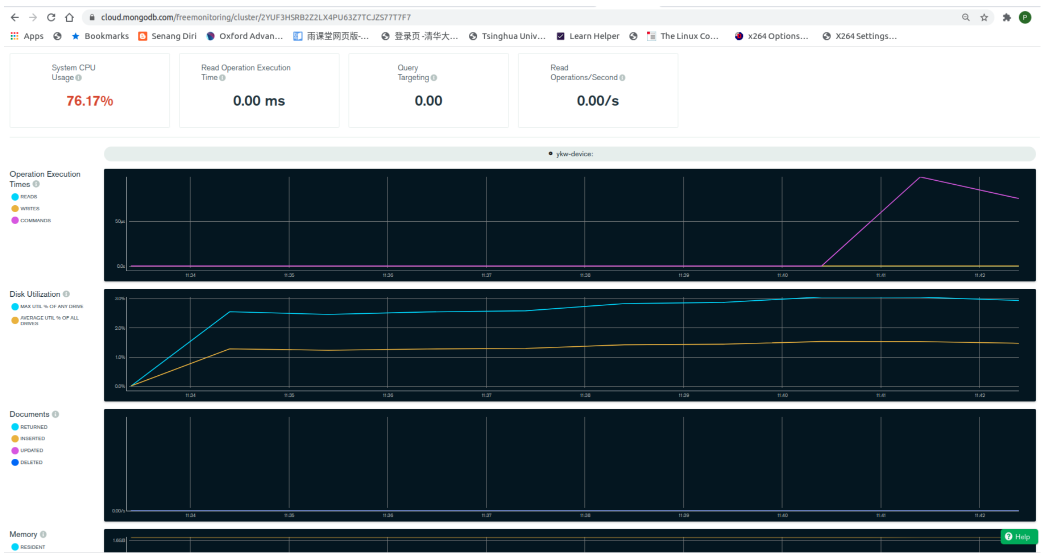


FIGURE 15: Browser demonstrating free monitoring

5 SOLUTION EVALUATION

In this section, we show that our system is able to fulfil the three main requirements set in the problem specification (section 3.4) of this project.

1. **Bulk loading with data partitioning and replica consideration.** - We discussed this in-depth in Sections 4.3 and 4.4, and this can be verified by running Makefile commands `make bulkload` and `make popNshard` as explained in the Manual (appendix).
2. **Efficient execution of CRUD operations.** - MongoDB easily facilitates CRUD operations on sharded collections as if they were all in one place. Furthermore, we also showed in section 4.6 that when one collection changes, another related collection will be updated via a python script. Moreover, in section 4.7, we show how multimedia data associated with an article can be retrieved.
3. **Monitoring running status of DBMS servers.** - We demonstrated in section 4.8 how the monitoring utilities in MongoDB can be used to monitor the sharded data and the workload on each server.

6 CONCLUSION

6.1 Future Work

Although our system successfully implements the three main objectives, there are some functions advantageous to a distributed system that can be added. These include:

1. **Hot Standby DBMS for fault tolerance.**

   MongoDB facilitates the setting up of replica sets which increase robustness against faults. At the moment, we only have a replica set for the config servers, but not for the data shards (DBMS1, DBMS2, grid1, grid2) due to lack of computer space, but this could be easily enabled by setting up extra Docker containers to simulate secondary servers to DBMS1, DBMS2. In case the primary server in one of the DBMS shard clusters fails, the system will automatically switch to the secondary server (which contains the same exact data as the primary server), thus allowing all processes to continue without any interruption.

2. **Expansion at the DBMS-level allowing a new DBMS server to join.**

   Another docker container could be configured as another mongod instance and connected to the mongos router, a typical requirement when the volume of data gets too large. In fact, this was how we added the two GridFS servers (grid1, grid2) to manage the multimedia data. However, at the moment, the steps to configure another docker container as another DBMS and adding to the distributed system environment are quite convoluted, and in the future, a GUI could be designed for easy addition of a new DBMS server.

3. **Dropping a DBMS server at will.**

   For the cases where a DBMS server is detected to have corrupt data/ missing data, it is unremunerative to maintain it in the system since the data would not be consistent among the replicas and lead to missing data in the new replicas. To maximize the space in our system, we need to enable the system to be able to detect and drop DBMS servers that fit the category.

4. **Data migration from one data center to other.**

Data center migration in the context of enterprises would refer to the physical movement of a data center from one location to another, and this tends to require proper planning, time and high costs. The goal is to minimize downtime and disruption while users experience no interruptions on their side during data migration which can be a result due to any change in our hardware systems. Therefore, there is need to ensure that whenever migration occurs, the data remains consistent between databases after a migration so that there is no case of corrupted or partially migrated data or in the worst case missing data.

6.1.1 GUI

Aside from the future work listed above, creating a user-friendly GUI to operating our database system would be ideal. At the moment our system can only be utilised through the terminal, and requires a certain level of technical knowledge to be able to use it. Furthermore, it takes quite a lot of time to key in all the required commands. Therefore there is a need to build a user friendly graphical interface where users can directly access the multimedia data from the articles at the front-end, and for administrators to easily manage the database system at the back-end.

References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 2013.

[2] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a Database System*. Now Publishers Inc., Hanover, MA, USA, 2007.

[3] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 2010.

[4] P. V. M. Tamer Özsu. Principles of Distributes Database Systems. *Springer-Verlag New York*, pages XX, 846, 2011.

[5] I. MongoDB. MongoDB Manual. *Copyright MongoDB*, 2020.

[6] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6:1068–1079, 2013.

7 Appendix A: Manual

The detailed instructions and scripts for our project can be found in our Github repository https://github.com/kwyoke/DDBS-2020, but we still present the main steps required for the set-up in this section.

7.1 Prerequisites to Install

1. Ubuntu 18.04 machine
2. Docker, docker-compose
3. MongoDB Community Edition, mongofiles
4. PyMongo

We implemented the project on a Ubuntu 18.04 machine, but other machines should work fine as well. Docker and MongoDB should be installed according to their official documentations.

7.2 Generating dummy data

Run the `executive_program/proj_data/genTable_mongoDB_toy.py` to generate about 80MB of dummy data including `user.dat, article.dat, read.dat` and multimedia data. The 10GB version of the script can be found in `mongoshell_tutorial/`.

7.3 Running `Makefile` to set up everything

Navigate to `executive_program/` which contains the `Makefile`. Simply run in the terminal:

```
make all # runs everything
```

Alternatively, to run only specific portions (see `Makefile` for more detail):

```
make setup # containers
make bulkload # copy and load user, article, tables
make popNshard # populate and shard collections
make store_multimedia # store multimedia data in GridFS servers
```

Now, all the required collections (user, article, read, beread, popRank) are populated and sharded, and queries can be performed in the mongos server's mongo shell.

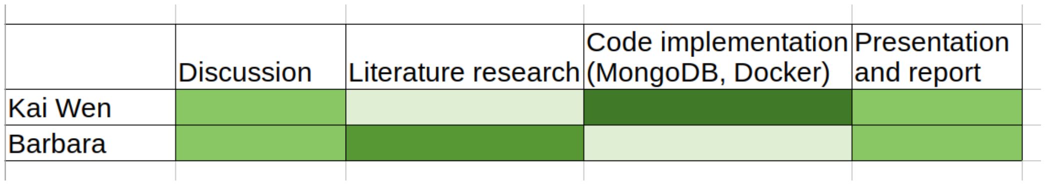
Other relevant information:

The `Makefile` sets everything up, with the help of `popNshard.js` which contains the mongo shell commands for populating and sharding the collections, and `bulk_store_multimedia.sh` for storing multimedia files into GridFS.

Also, run `auto_refresh_onarticle.py` and `auto_refresh_onread.py` in terminals so that the collections will be updated when there are any changes.

8 Appendix B: Workload allocation

The project workload is allocated as shown in Figure 16, with Kai Wen as the leader. The colour intensity indicates the amount of contribution in each area.



	Discussion	Literature research	Code implementation (MongoDB, Docker)	Presentation and report
Kai Wen	▒	░	█	▒
Barbara	▒	▓	░	▒

FIGURE 16: Work allocation