YanZhao Chen, Péter Garamvölgyi

Distributed Database Systems

Final report

[1]

Abstract

This project is aimed to build an efficient and user-friendly system to support a reading social platform that enables users to read articles that help reduce their anxiety and pressure as well as to communicate with other users with in-depth comments. The system is composed of two parts: web client interface and data center. A web client interface is implemented to provide user operations such as querying articles, querying popular rankings, and adding new articles; meanwhile, a cache buffer is provided to improve client performance. On the other hand, the data center is composed with MongoDB clusters and a single-node HDFS on Docker containers.

[1] image source: https://sungsoo.github.io

Problem background and motivation

With the development of modern society, people are paying more attention to mental health and the spiritual world. According to a survey, 20% of global teens are suffering from psychological diseases, and it is the top killer in China and Korea (WHO, 2016). Teenagers and adolescents are increasingly becoming more stressed in this fast-paced world. There needs to be more resources and platforms to help them overcome this difficulty. Under these circumstances, we aim to provide a platform to help them read articles that interest them and communicate with others with the same interests so as to help them relieve their anxiety and pressure.

This project will set up a system to provide users with a better reading experience. Our system is designed to provide smooth user experience and support various user operations. Users can query articles they want to read as well as create new articles. The system also provides popular article rankings to help users find articles that might interest them.

Existing solutions

Traditional server-side architecture has been shaken up by new technologies that better streamline infrastructure and maximize server resources (Upwork, n.d.). Containerized applications for deployment in cloud platforms have become increasingly popular. Docker has been the driving force behind this trend, with 2/3 of the companies trying it ending up adopting its usage (Novoseltseva, 2017). It ensures consistency and standardization across platforms and allows for faster configuration. This is more lightweight and easy to use compared to virtual machines. Hadoop file system is also supported with Docker images.

There are popular choices in the industry with regards to data storage. For relational databases, common existing solutions for server-side database include MySQL, Oracle, MS SQL Server, etc. These database share similar properties such as standard storage in tables, existence of primary and foreign keys to establish relationship, similar syntax to operate, and support for scalability and high-performance (Upwork, n.d.). Oracle and SQL Server are not free, while MySQL is open source. Each company has additional support for their own database, for example, .NET framework can be easier to integrate with SQL Server. On the

other hand, MongoDB is representative of NoSQL databases, where it has collections and JSON-like documents instead of tables and rows. There is more flexibility in changing the structure of the documents and having a hierarchical structure by using subdocuments. Therefore, MongoDB's properties are more suitable for this use case providing superior scalability and performance (mongoDB, n.d.).

A cache provides faster querying speed especially when dealing with a lot of users performing similar operations at the same time. Two popular options are Redis and Memcached, which are both in-memory key-value data stores (Vimal, 2017). They both support super fast caches stored in RAM, however, Redis offers more powerful features. Memcached is better used for small and static data without support for different data types. On the other hand, Redis is more of a "data structure server" and provides greater power and efficiency (Vimal, 2017). Since data from the database contain more complex structures and relations, Redis is the more preferred option.

Proposed solution

The proposed architecture consists of four parts:

1. MongoDB database cluster.
2. Hadoop/HDFS file storage for unstructured data.
3. Python web API providing indirect data access.
4. Redis cache.

The figure below illustrates the overall architecture of our system.

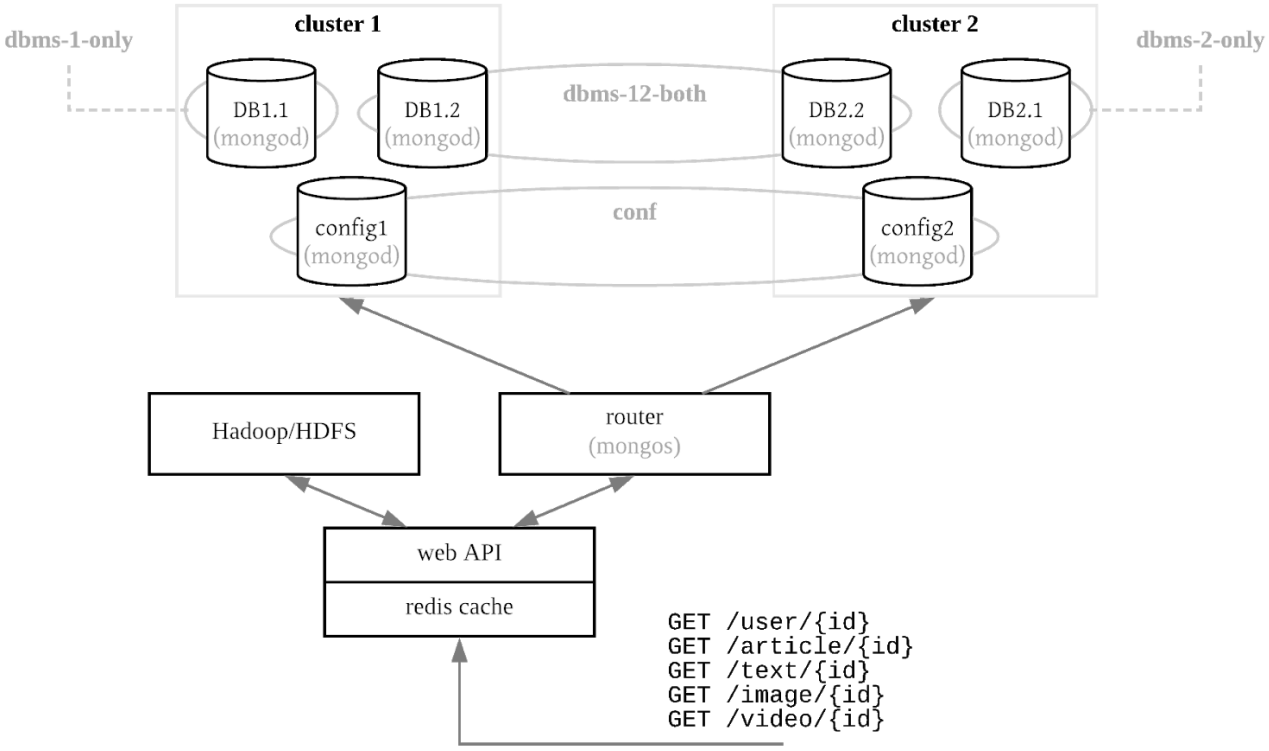
Database design

In our database design, we strive to utilize many built-in mongoDB functionalities. These include

- Replica sets for replication.
- Shards for distribution of data among different clusters.
- Automatic query execution.

To fulfill the project requirements, we decided to use two clusters of three-three mongo instances each. Together, these six nodes constitute four replica sets:

- config1 and config2 constitute the configuration server replica set. In mongoDB, shard configurations are stored in such configuration servers. While having one such server is sufficient, we chose to have one in each cluster to increase resilience.
- DB1.2 and DB2.2 constitute a cross-cluster replica set. Data that needs to be stored on both clusters should be saved to this replica set.
- DB1.1 and DB2.1 each constitute its own single-node replica sets. Data that does not require cross-cluster replication should be stored in one of these.



Using the built-in replication functionality of mongoDB provides us with great flexibility: Increasing the resilience of any of these replica sets by adding more nodes is almost trivial.

The entry point of the system is the mongos router node, responsible for sharding and query execution. This node has the following properties:

- The router stores its configuration in the configuration server. This means that losing a router instance does not lead to significant data loss. (The only potential data loss is that of ongoing queries.)
- mongoDB allows us to use an arbitrary number of such router nodes; i.e. there could be one instance hosted on cluster-1 another on cluster-2. Because of this, the router will become neither bottleneck nor single-point-of-failure.
- Collection sharding is configured through the router; for more details, please refer to the configuration manual in the Appendix.

Our setup implements this design using 7 docker containers managed by docker-compose. Tools like Docker Swarm would make it fairly straightforward to start these nodes on either seven or two separate physical machines, which of course would be a more realistic deployment.

File storage

We decided to use Hadoop/HDFS for storing unstructured data. These include texts, images, and videos attached to the individual articles. We treat HDFS as a blob storage where we simply save files under specific paths and leave the actual storage decisions to the system.

HDFS offers great scalability and fault tolerance, even for very large files. While our deployment uses a single-node docker image to run all Hadoop components (name node, data nodes, etc.), the implemented solution would work the same way with an actual multi-node Hadoop deployment.

Data import

Having set up the database and file storage components, we needed to find a way to bulk import the example data sets and generate the other two collection.

First, we implemented a Python script (`import_collections.py`) that is responsible for importing the three pre-generated collections: user, article, and read. We rely on `pymongo`, a simple Python client library for mongoDB that enables use to execute mongoDB queries

remotely and automatically. Moreover, we implemented configurable batching for bulk import to reduce DB workload and make it easier to follow progress on the client side.

To generate the collections be-read and popular-rank, we implemented two other scripts (`generate_be_read.py`, `generate_popular_rank.py`). These scripts rely heavily on mongoDB's aggregation pipelines. As an example, generating the daily popular ranks involves the following steps:

1. **Projection**: Only keep the fields `aid`, `timestamp` and `readOrNot`. Convert timestamp into a formatted date string only containing year, month, and day, e.g. `1545969992` would become `2018-12-28`.
2. **Grouping**: We group the entries based on `date` **and** `aid`. For entries with the same `date` and `aid`, we sum up the `readOrNot` values, effectively counting the number of reads for the given article on the given day.
3. **Sort**: We sort the results increasing by `date`, decreasing by `count`, and increasing by `aid`. This means that, having many articles with the same read count on the same day, we simply follow alphabetical ordering. (This is quite relevant in our synthetic dataset; however, in real life scenarios, articles are likely to have different read counts.)
4. **Grouping**: We group again by the `date`, collecting `aid`'s and read counts from the same day into an array.
5. **Projection**: For each array (containing all articles from a given day sorted) we only keep the first 5, i.e. the top 5.
6. **Add fields**: We add the field `temporalGranularity` with the value **"`daily`"**.
7. **Output**: We save the results in a new collection.

While this seems complex at first, having grasped the basics of mongoDB aggregation, such complex queries become surprisingly easy. We have, however, run into some limitations of mongoDB, including:

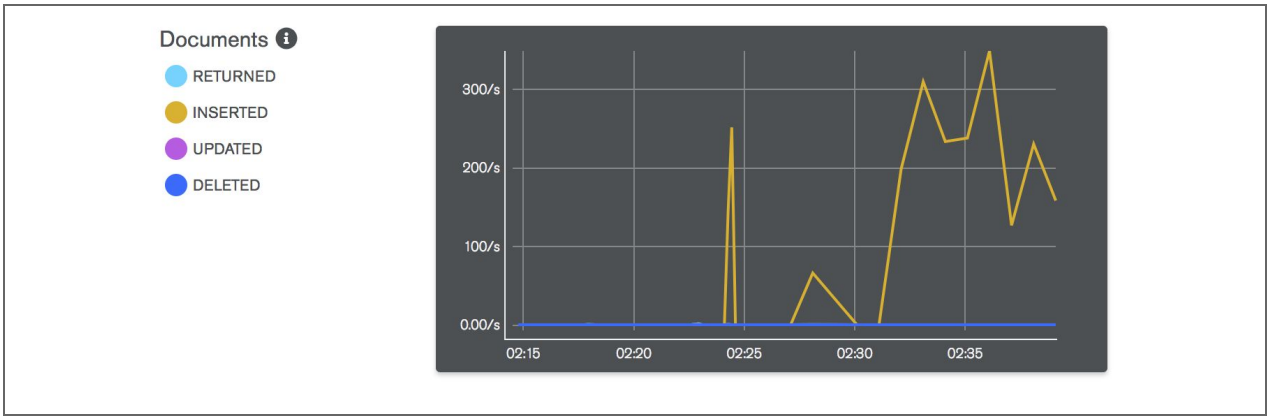
- The output of an aggregation cannot be sharded. This is relevant for be-read. Our less-than-satisfying workaround is to first output the results into a non-sharded collection, and then copy all entries into the final collection.

- The output of an aggregation will overwrite the target collection instead of appending to it[2]. This is relevant for popular-rank, as we execute three separate aggregations for daily, weekly, and monthly ranks. The workaround, again, is to use temporary collections and then merge the results manually.

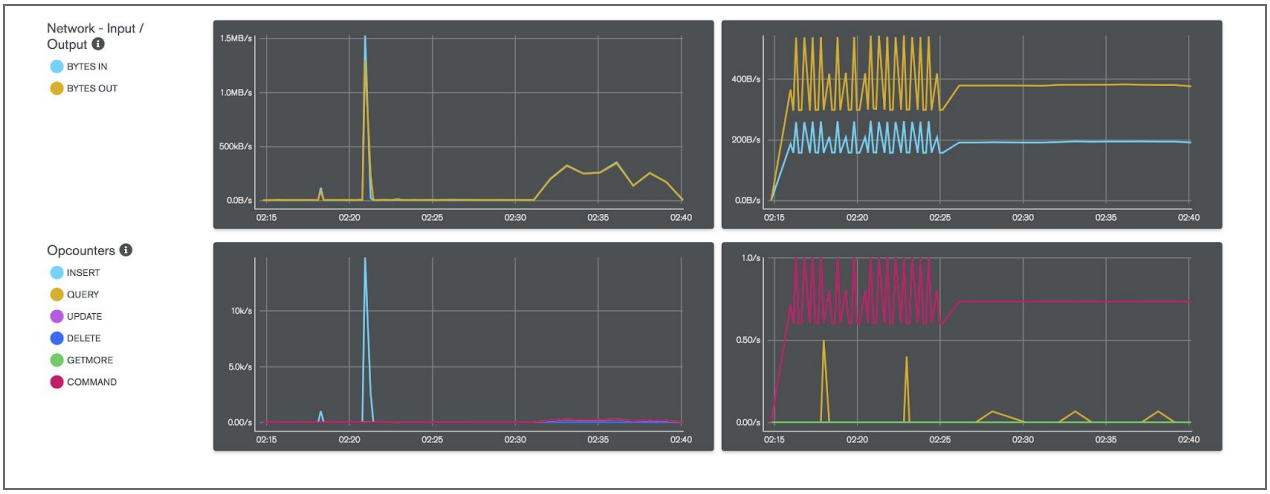
DB monitoring

For database monitoring, we simply use the free cloud monitoring service provided by mongoDB. By running the command `db.enableFreeMonitoring()` on the database nodes, we get a cloud URL with useful statistics for each instance. These include disk, network, and CPU usage, the number and type of operations executed, etc.

Below is an example output for inserted documents during a bulk insert session.



For multi-node replica sets, we can see each node's activity separately:



[2] https://jira.mongodb.org/browse/SERVER-12280

Web API

When developing database systems for frontend and/or mobile phone applications, it is common to hide the storage details and control access by providing a web API as a gateway to the databases. This way the clients simply need to make HTTP requests without knowing about the actual database design behind. This design is often called *thin client* and it corresponds to common software engineering best-practices such as decoupling and access control.

We use the popular Python web framework Flask for implementing our simple API. Every endpoint triggers a specific mongoDB query sent to the database router. The results are sent back over HTTP in JSON format.

The following table shows the implemented endpoints and some example responses.

Endpoint	Example response

`GET /user/<int:uid>`
`GET localhost:5000/user/1`
	`{`
	`  "dept": "dept17",`
	`  "email": "email1",`
	`  "gender": "male",`
	`  "grade": "grade4",`
	`  "language": "zh",`
	`  "name": "user1",`
	`  "obtainedCredits": "18",`
	`  "phone": "phone1",`
	`  "preferTags": "tags39",`
	`  "region": "Beijing",`
	`  "role": "role0",`
	`  "timestamp": "1506328859001",`
	`  "uid": "1"`
	`}`

`GET /user/<int:uid>/articles_read`
`GET localhost:5000/user/1/articles_read`
	`[`
	`  "199897",`
	`  "128632",`
	`  "109724",`
	`  ...`
	`  "175074",`
	`  "15445"`
	`]`

GET /article/<int:uid>
GET localhost:5000/article/1
	{
	"abstract": "abstract article1",
	"aid": "1",
	"articleTags": "tags33",
	"authors": "author471",
	"category": "science",
	"image": "77af778b51",
	"language": "zh",
	"text": "84393add8c",
	"timestamp": "1506000000001",
	"title": "title1",
	"video": "92a15e5a53"
	}

GET /article/<int:uid>/meta
GET localhost:5000/article/1/meta
	{
	"agreeNum": 1,
	"agreeUidList": [
	"1383"
	],
	"aid": "1",
	"category": "science",
	"commentNum": 1,
	"commentUidList": [
	"1383"
	],
	"readNum": 4,
	"readUidList": [
	"5134",
	"1383",
	"774",
	"3590"
	],
	"shareNum": 1,
	"shareUidList": [
	"5134"
	]
	}

GET /text/<string:id> GET localhost:5000/text/84393add8c	[text of article #id]
GET /image/<string:id> GET localhost:5000/image/77af778b51	[image of article #id]
GET /video/<string:id> GET localhost:5000/video/92a15e5a53	[video of article #id]

GET
/top/<string:date>/<string:granularity>
GET localhost:5000/top/2017-12-1/daily
	[
	  {
	  "aid": "106416",
	  "count": 3
	  },
	  {
	  "aid": "163877",
	  "count": 3
	  },
	  {
	  "aid": "31113",
	  "count": 3
	  },
	  {
	  "aid": "74362",
	  "count": 3
	  },
	  {
	  "aid": "78462",
	  "count": 3
	  }
	]

This API could easily be extended to offer REST-like CRUD support including updates and deletes. However, this would require stricter access control and as of now is not yet exposed on our server.

Redis cache

Some queries like loading images or performing complex operations like listing all articles a user has read may take a relatively long time. To solve this issue, we decided to use a local in-memory Redis cache on the web API node.

Redis is run as a docker container on the API host. We use the flash_caching Python package to connect Flash with Redis. Having configured Flask, adding caching ability to the individual endpoints is as easy as adding a single line of code.

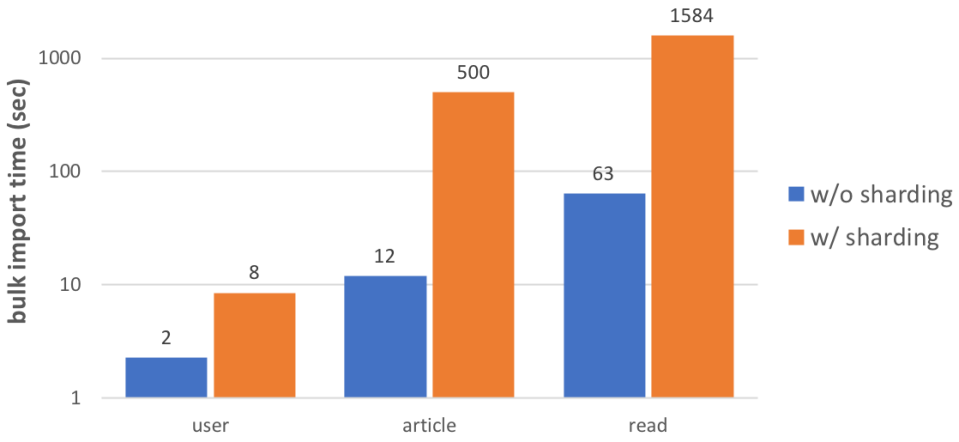
We configured flash_caching with a 60 second timeout. That is, after a query to a certain resource, subsequent queries within the same minute will retrieve the result from Redis instead of the database cluster. Queries after one minute will again access the database.

This 60 second interval is an arbitrary choice and should be chosen based on user experience considerations.

Solution evaluation

DB access performance

Some simple benchmarks on our scripts revealed that sharding and replication have a significant negative impact on performance of bulk imports (logarithmic axis):



The impact on collection generation is also significant:

However, we do not consider this an issue, as data import and collection generation are both single-time operations. At the same time, query performance has remained the same or even improved in some cases.

Impact of caching

Given how simple adding Redis cache to our web API was, it gave us a surprisingly large performance boost, especially with complex queries. For retrieving things like the list of articles read by a user or images and videos, using Redis reduced access time from up to several hundreds of milliseconds to just around 15ms, which is a great improvement in user experience.

As mentioned before, for real-world deployments, more thought has to be put into finding the right cache timeout. If we are going to have multiple apps or web pages using the same database backend, cache invalidation has to be considered as well.

Tool evaluation

The challenges we faced when using mongoDB have been discussed in the previous section. In the next paragraphs we will elaborate on some other problems we had and evaluate whether our choice of tools was appropriate.

Docker and docker-compose have made setting up multiple database instances and HDFS much easier than it would have been otherwise. That said, we still ran into some limitations. We decided to set up a custom subnetwork for our containers so that we can assign static IPs to them. However, docker on Mac does not support IP-based host-container connections for custom networks; we needed to find some workarounds to solve this.

HDFS has proven to be a sufficient tool with a usable API. However, we will need to put more effort into the performance evaluation of this approach, especially on multi-node setups.

Overall, our choice of mongoDB, HDFS, and Docker made most of our features fairly straightforward to implement.

Solution verification

Below, we will go through the project requirements one-by-one and discuss how our solution handles them.

1. Bulk data loading with data partitioning and replica consideration ✓

   Our Python script discussed above take care of bulk data loading through the router node, while mongoDB's sharding and replication makes sure the data is saved at the correct database instances.

2. Efficient execution of data insert, update, and queries ✓

   Here, again, we rely on mongoDB's built-in functionalities. By executing queries on the router node, mongoDB will automatically distribute the corresponding sub-queries among database nodes and aggregate the results.

3. Monitoring the running status of DBMS servers, including its managed data (amount and location), workload, etc. ✓

   MongoDB's free cloud monitoring gives us real-time metrics like number of documents, number and type of operations being executed, etc. These give us sufficient information to gain a good understanding of the status of our system.

4. (Optional) advanced functions
   a. Hot / Cold Standby DBMSs for fault tolerance ✓

      A hot standby DBMS can easily be added to the system by simply adding another node to the corresponding replica sets. This way, all updates and inserts will be stored in the standby instance too. However, this might incur a performance impact.

   b. Expansion at the DBMS-level allowing a new DBMS server to join ✓

      Similarly to the previous point, it is very easy to add new instances either to existing replica sets of new shards. We just need to make sure that we update the replica- and shard configurations accordingly.

c. Dropping a DBMS server at will ✓

Dropping a server within a replica set does not lead to data loss (apart from unfinished writes). MongoDB handles failover automatically.

d. Data migration from one data center to others ✓

Having added a new data center, data can be moved through the router by calling the moveChunk method (see the Appendix for more details). By first replicating data to both data centers and later dropping the original one, we can avoid data loss.

Conclusion

Throughout the implementation of this project, we used some of the techniques that we have learned in the class as well as self-learn some new techniques along the way. For example, we decided to use sharding on MongoDB, yet the support for aggregating shards on MongoDB is very limited so we have to pre-process the data using the techniques that we have learned from the class. Starting from designing the whole system schema to implementing each layer of the schema in bottom-up order, we have gained NoSQL development experience throughout this project while, at the same time, developed a system that will hopefully help struggling teenagers to cope with their stress and anxiety.

References

(WHO, 2016)
https://www.who.int/mental_health/maternal-child/child_adolescent/en

(Upwork, n.d.)
https://www.upwork.com/hiring/development/what-is-containerization-and-is-it-the-right-solution-for-you

(Novoseltseva, 2017)
https://apiumhub.com/tech-blog-barcelona/top-benefits-using-docker

(mongoDB, n.d.)
https://docs.mongodb.com/manual

(Vimal, 2017)
http://www.ranjeetvimal.com/memcached-vs-redis-one-pick

Appendix I.: Manual

In this section, we are presenting step-by-step instructions for locally running our system.

Prerequisites

We assume the following applications are installed and available: docker, docker-compose, python[3].

- Docker is used for starting mongoDB, redis, and Hadoop/HDFS containers.
- Docker-compose simplifies the coordination and management of multi-container systems. Note: The containers can be started one-by-one; if you would like to do this, please refer to the corresponding docker-compose config files.
- Python is used for database imports and the web server.

Installing dependencies

First, make sure you have the necessary docker images by running:

```
$ docker pull mongo
$ docker pull redis
$ docker pull sequenceiq/hadoop-docker:2.7.1
```

Then, proceed to install the required python packages.

```
$ pip install pymongo hdfs flask flask_caching redis
```

Cluster startup

Start the database clusters by running the following commands:

```
$ mkdir db db/conf1 db/conf2 db/dbms11 db/dbms12 db/dbms21 db/dbms22
$ docker-compose -f config/cluster-1.yml -f config/cluster-2.yml \
                 -f config/hdfs.yml --project-name cluster up -d
```

[3] Our tests were done using Python version 3.6.

These commands will initialize the folders storing the database files, and then proceed to run eight containers.

- The first cluster contains: config1, dbms11, dbms12, router1.
- The second cluster contains: config2, dbms21, dbms22.
- Finally, the third configuration specifies the single-node HDFS container.

To stop the containers, simply run

```
$ docker-compose -f config/cluster-1.yml -f config/cluster-2.yml \
                 -f config/hdfs.yml --project-name cluster down
```

Replica set configuration

To initialize the configuration replica set (config1 and config2), run:

```
$ docker run -it --net cluster_mongonet mongo \
             mongo mongodb://172.18.0.30:27017
> rs.initiate({
    _id: "config", configsvr: true, members: [
      { _id : 0, host : "172.18.0.30:27017" },
      { _id : 1, host : "172.18.0.40:27017" }
    ]
})
```

What we do here is connect to config1 (at **172.18.0.30:27017**) and configure it as a config server (**configsvr:true**), letting it know how to reach config2 (**172.18.0.40:27017**). This command could be run on config2 as well (but not on both of them). After some seconds, the prompt should change to **config:PRIMARY>**.

To initialize the dbms1-only replica set, run:

```
$ docker run -it --net cluster_mongonet mongo \
             mongo mongodb://172.18.0.31:27017
> rs.initiate()
…
dbms1-only:PRIMARY> |
```

To initialize the dbms2-only replica set, run:

```
$ docker run -it --net cluster_mongonet mongo \
             mongo mongodb://172.18.0.41:27017
> rs.initiate()
…
dbms2-only:PRIMARY> |
```

To initialize the dbms12 replica set, run:

```
$ docker run -it --net cluster_mongonet mongo mongo
mongodb://172.18.0.32:27017
> rs.initiate({
    _id: "dbms12", members: [
      { _id : 0, host : "172.18.0.32:27017" },
      { _id : 1, host : "172.18.0.42:27017" }
    ]
})
…
dbms12:PRIMARY> |
```

Shard configuration

Next, we are going to initialize sharding. First, connect to the router and configure it with the shard addresses:

```
$ docker run -it --net cluster_mongonet mongo \
             mongo mongodb://172.18.0.33:27017
> sh.addShard("dbms1-only/172.18.0.31:27017")
> sh.addShard("dbms2-only/172.18.0.41:27017")
> sh.addShard("dbms12/172.18.0.32:27017")
> sh.enableSharding("db")
```

Let us configure sharding on the user collection.

```
> sh.shardCollection("db.user", { region : 1 })
> sh.disableBalancing("db.user")

> sh.splitAt("db.user", { region: "Beijing" })
> sh.splitAt("db.user", { region: "Hong Kong" })
> sh.moveChunk("db.user", { region: "Beijing" }, "dbms1-only")
> sh.moveChunk("db.user", { region: "Hong Kong" }, "dbms2-only")
```

Let us configure sharding on the article collection.

```
> sh.shardCollection("db.article", { category : 1 })
> sh.disableBalancing("db.article")

> sh.splitAt("db.article", { category: "science" })
> sh.splitAt("db.article", { category: "technology" })
> sh.moveChunk("db.article", { category: "science" }, "dbms12")
> sh.moveChunk("db.article", { category: "technology" }, "dbms2-only")
```

Let us configure sharding on the read collection.

```
> sh.shardCollection("db.read", { region : 1 })
> sh.disableBalancing("db.read")

> sh.splitAt("db.read", { region: "Beijing" })
> sh.splitAt("db.read", { region: "Hong Kong" })
> sh.moveChunk("db.read", { region: "Beijing" }, "dbms1-only")
> sh.moveChunk("db.read", { region: "Hong Kong" }, "dbms2-only")
```

Let us configure sharding on the be-read collection.

```
> sh.shardCollection("db.be-read", { "category" : 1 })
> sh.disableBalancing("db.be-read")

> sh.splitAt("db.be-read", { "category": "science" })
> sh.splitAt("db.be-read", { "category": "technology" })
> sh.moveChunk("db.be-read", { "category": "science" }, "dbms12")
> sh.moveChunk("db.be-read", { "category": "technology" }, "dbms2-only")
```

Bulk import data

Having set up our clusters, initialized our replica sets, and configured sharding, we are now ready to generate and import the data. To generate the test data, run:

```
$ cd data
data $ python ../scripts/genTable_433507293.py
```

Next, run the following four scripts:

```
$ python src/import_collections.py
$ python src/generate_be_read.py
$ python src/generate_popular_rank.py
$ python src/import_assets.py
```

As their names suggest, the first one imports the test data (user.dat, article.dat, read.dat); the second one generated the be-read collection; the third one generates the popular-rank collection; while the last one imports texts, images, and videos to HDFS.

Check and query

To can play around with the two mongoDB clusters directly by connecting to the router and issuing standard mongo commands:

```
$ docker run -it --net cluster_mongonet mongo \
             mongo mongodb://172.18.0.33:27017
> use db
> show collections
> db['be-read'].find()
...
```

Start server

To start the web server, first make sure you have a running redis instance listening on the standard redis port 6379. For example, you can use docker to run an instance like this:

```
$ docker run --name redis -d -p 6379:6379 redis
```

To run the server, simply run:

```
$ python src/server.py
 * Serving Flask app "server" (lazy loading)
   ...
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
   ...
```

The server now runs and is accessible at **`127.0.0.1:5000`**. You can now issue HTTP requests to the endpoints discussed above using tools like curl, Postman, or Google Chrome.

Troubleshooting

- Docker error: container name "/xxx" is already in use. Cannot recreate the container as the name is already used by an existing container. If you think you will not need that container instance, you can stop and remove it using

```
$ docker stop xxx && docker rm xxx
```

- Docker error: cannot create network as it overlaps with existing IP ranges. One of your existing docker network configurations conflicts with our system. Try:

```
$ docker network ls
NETWORK ID          NAME          DRIVER    SCOPE
…
4ed0a6b6a6a3    your_network    bridge    local
…
$ docker network inspect your_network
…
$ docker network rm your_network
```

- **Mongo connection refused**: If you fail to connect to mongoDB, make sure you are using the correct IP address and port. You might need to wait for a few minutes after starting the cluster.

- **Replica set errors**: Make sure your replica sets are initialized. If you connect to one of your node, the prompt should say PRIMARY (or SECONDARY). For debugging, run the following command on one of the database nodes (config1, config2, dbms11, dbms12, dbms21, dbms22):

```
> rs.status()
```

- **Sharding errors**: Make sure your sharding configuration is correct. For debugging, run the following command on the router:

```
> sh.status()
```

Appendix II.: Work allocation

	Peter (project lead)	**YanZhao**
DB design	✓	✓
mongoDB configuration	✓	
HDFS configuration		✓
Docker setup	✓	
Monitoring		✓
Web API, caching	✓	
Data import and generation	✓	✓
Report and presentation	✓	✓