Report for Final Project of DDBS Class

Haoshen Li, Wentao Guo

Jan. 4, 2021

1 Abstract

We have constructed two replicated DBMS on MongoDB and build the connection to HDFS and cache AID/UID and downloaded files with redis server. We implemented a MongoDBManager class for tracking server status, allow new server to join in and leave, connect with Hadoop Distributed File System for retrieving popular articles and with redis for caching fragmentation scheme. In total, we have simulated a sample database and its ecosystem for an online library in the distributed context.

2 Introduction

As illustrated in the project aim, the most crucial problem we try to resolve is related to big data management in a distributed environment. High volume, high speed, and wide variety characterize the term "big data" and for the management of such data, it involves a broad concept which includes the policies, procedures, and the technologies used for collection, storage, governance, organization, administration and delivery of large repositories of data.

It is common to encounter challenges in big data management and there are more than ninety percent participants said that experienced big data management challenges based on the survey of In the IDG 2016 Data and Analytics. Some of common issues include data silos, growing data stores, data and architectural complexity, and ensuring data quality. Take data silos as an example: it is common to have multiple databases that include similar information, yet the data isn't always consistent from one database to another.

For example, a retailer may store customer addresses in a marketing database, a customer service database, an accounting database and an e-commerce website database. If one of those databases has slightly different information for a particular customer — such as listing a customer's street address as an "avenue" when the other databases list it as a "street" — it could lead to problems like duplicate mailings, losing track of customer service records and many other problems [3]. In addition, storing the same piece of information in several different locations eats up storage space — particularly when the problem is multiplied across an entire customer base.

With many challenges as mentioned above, successful big data management can bring us great benefits. For instance, one of the most obvious benefits of good data management practices is that it increases the accuracy and reliability of big data analysis. Good data coming in to the analytic solution sets the organization up for quality business insights coming out of the solution. Furthermore, according to the Experian survey, 57 percent of those surveyed said maintaining high-quality contact data helped them increase efficiency. Undoubtedly, cost savings is closely related to those efficiency gains, as experienced by 38 percent of those who took part in the Experian study. Similarly, in the NewVantage survey, 49.2 percent of those surveyed said their big data efforts had helped them decrease expenses [3]. All those benefits give us motivations to master big data management knowledge and implementation.

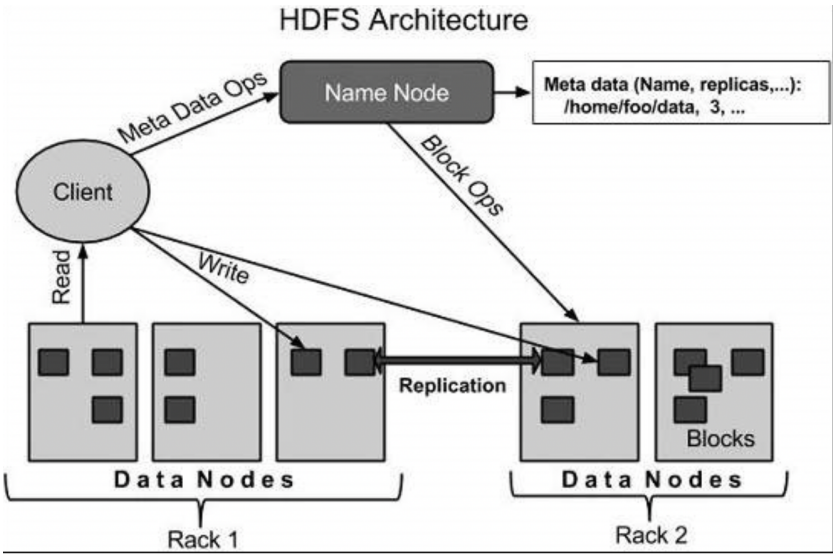
For our project specifically, we need to implement a distributed data center and achieve the following functionalities (1) efficient bulk data loading with data partitioning and replica consideration (2) efficient execution of data insert, update, and queries (3) online monitor the running status of DBMS servers, including its managed data (amount and location), workload, etc.

The technology stack we mainly use in this project includes Hadoop Distributed File System (HDFS) and MongoDB. Distributed file system design essentially contributes to the development of Hadoop File System. It is run on commodity hardware. Compared with other distributed systems, HDFS is highly fault-tolerant and designed using low-cost hardware.

HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure[7]. HDFS also makes applications available to parallel

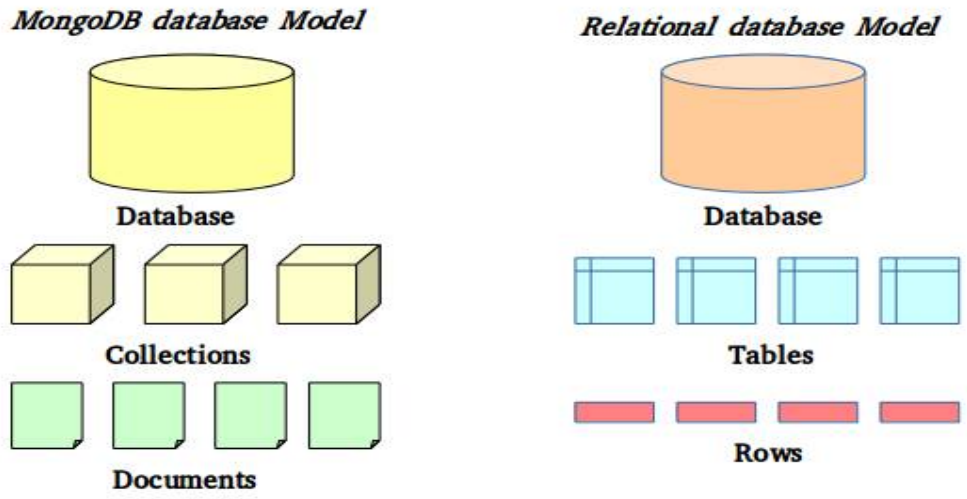
processing. Below is an overview of HDFS architecture[7]:

Figure 1: HDFS Architecture [7]



Scalable database is well deployed in enormous data centers at the field of online e-commence, libraries, etc. Bulk data will be inserted and updated frequently, and the magnitude of such data bulk often reaches hundreds of gigabytes or several terabytes per second, and such data also will to be frequently analyzed and queried. In the last century, relational database has been designed for satisfying ACID models but the rapid growth of Internet economy requires a faster and more available database models. People then turned to the idea of distributed databases, which features high availability and great scalability as it can take advantage of machine clusters and resilient to common network changes and hardware failures.

Figure 2: MongoDB vs. Relational Database [1]



The database we deploy in this project is MongoDB, which is a document-oriented NoSQL database used for high volume data storage. Instead of maintaining high-relational data as what the traditional relational databases do, MongoDB makes use of loosely-formatted documents to maintain data relations. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. In addition, MongoDB's primary-secondary data node model performs really well in distributed setting, and read from replica set can alleviate the high IO pressure on a single server, which in turn increase the data throughput. [2]

3 Problem Background and Motivation

For our project, we will construct a sample data center to simulate the workflow of an online library or other reading platform. There are 3 kinds of information we can gather: user, article, and how users react to the presented articles. We want to know the popular articles and group the article by users' read pattern (whether user share, whether user agree, etc). We also want to construct a distributed database with multiple data centers for performance, and we want to keep track of server status and accelerate the database query

if possible. This simulates how a real online library data-management works, and it will serve as a model for deploying and manipulating distributed database and its ecosystem, such as distributed file system.

4 Problem Definition

There are 5 tables as a data source: User, Article, Read. We need to populate 'Be-Read' and 'Popular-Rank'. Table of Be-Read contains records for user read behavior (whether the user really read this article, whether user agreed, etc.) while table of Popular-rank contains Top5 mostly read articles for a temporal granularity of day, week and month. Moreover, we are also interested in fulfilling for the following functionalities:

- Bulk data loading with data partitioning and replica consideration
- Efficient execution of data insert, update, and queries
- Monitoring the running status of DBMS servers, including its managed data (amount and location), workload, etc.
- Dropping a DBMS server at will and adding a DBMS server to running server cluster during runtime
- Data migration from one data center to others

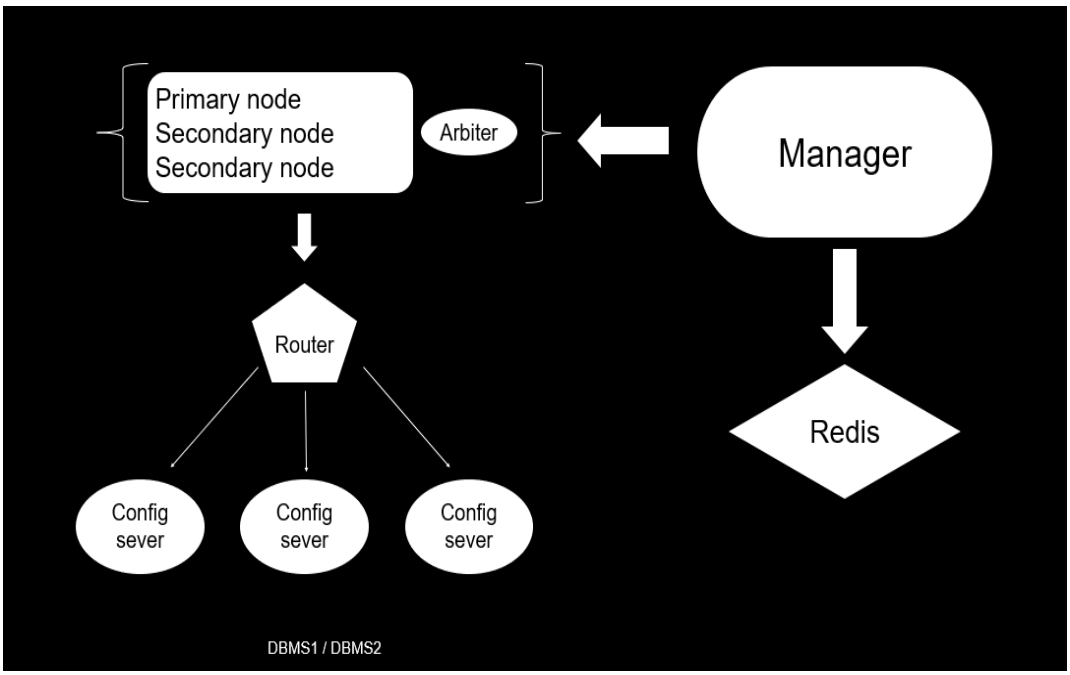
5 Existing Solutions

User, article, and read data is intrinsically relational and therefore deploying a relational database system, such as MySQL, is highly favorable to the industry. A typical solution should be constructing 3 relational table for User, Article and Read, together with a foreign key-primary key and some triggers. Then, we do the insertions and group the data for 'Be-Read' table and 'Popular-Rank' on a single server.

6 Solution

We first generate a 10,000 users, 50,000 articles and 100,000 reads. This creates 200,000 folders that store images, texts, and videos for articles. We construct

Figure 3: Overall Architecture



two clusters of MongoDB servers that represent DBMS1 and DBMS2. DBMS1 is the database cluster for storing 'User' in Beijing and part of science 'Article' while DBMS2 stores 'User' in Hong Kong and part of science 'Article' and all technology 'Article'. We notice that MongoDB has a primary-secondary read and write model for distributed data replication and synchronization. [6].

We apply such model, as we initiate a replica set for each cluster. For each cluster, we have 2 secondary nodes, 1 primary node, 1 arbiter, 3 config servers (as a config replica set), 1 router.

After deploying these 2 replica set (each corresponds with one DBMS), we create a MongoDBManager class that manages the connection to DBMS1 and DBMS2 and record the server logs and status info for each cluster whenever the user call this method. We have implemented a function in MongoDBManager to specialize in profiling server status. Note that MongoDBManager will also manages the redis server for quick AID and UID lookup for the partitioning scheme. Note that once a MongoDBManager instance is created, it will automatically retrieve all the AID and UID from DBMS1 and DBMS2 and

load the partitioning scheme into the redis server.

Figure 4: Server Status Log for DBMS1

We implement MongoDBInserter and MongoDBUpdater for managing bulk insert/update. For each collection, MongoDB supports a bulk insert/update operation in pymongo package [4] yet we need to control the size of such bulk since it cannot exceed the size of namespace, which has default of 16 MiB. We first take a small sample and measure the average size of document as S, and then we created a array with size $\frac{16 * 1024 * 1024}{S}$ and then we use the pymongo bulk_write API to write this array of insertion/update operation to the collection.

Figure 5: Bulk Write Implementation with Fixed-Size Array

```python
def testOneBulkWrite(self, col, arr, ptr, bulkSize):
    if arr[ptr] == bulkSize:
        ptr = 0
        col.bulk_write(arr, ordered=True)
    return ptr

def process_one_file_one_collection(self, file, col, bulk_size, pred):
    arr = [range(bulk_size)]
    ptr = 0
    with open(file, "r") as f:
        for line in f.readlines():
            if pred(line) == False: continue
            ptr = self.testOneBulkWrite(col, arr, ptr, bulk_size)
            arr[ptr] = InsertOne(json.loads(line))
            ptr += 1
    col.bulk_write(arr[:ptr], ordered=True)
```

We need to populate the popular-rank table, and we need to traverse read table for doing so. During this traversal, we will create 3 dictionaries for each DBMS to store the article popularity in the specified temporal granularity. We use 'aid' as the key, and another dictionary *timeKey_lookup* as the value. *timeKey_lookup* uses one time key as the key and maps to the read count on this time key. For example, we might have recorded three reads record on 'Oct, 2017' for one article, and *timeKey_lookup* will store the key 'Oct, 2017' and map to 3 (read count) in this case.

Daily/Weekly/Monthly Dictionary Structure

$$\{\,\text{aid}: \{\,\text{timestamp key [t] in desired granularity}: \text{ read count during } [t]\,\}\,\}$$

Figure 6: Daily/Weekly/Monthly Dictionary Core Logic

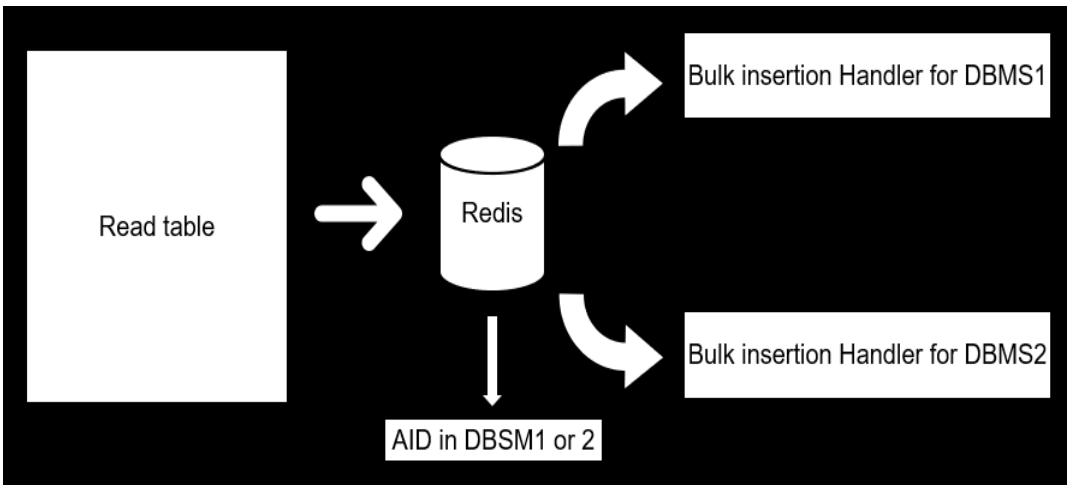
```
def process_one_entry(self, lookup, timeKey, aid):
    if aid in lookup:
        pair = lookup[aid]
        if timeKey in pair:
            pair[timeKey] = pair[timeKey] + 1
        else:
            pair[timeKey] = 1
    else:
        pair = dict()
        pair[timeKey] = 1
        lookup[aid] = pair
```

After this traversal, we sort the daily/weekly/monthly dictionary based on each item's value (*timeKey_lookup*)'s maximum value. We then collect the top 5 'aid' and this is the top5 read articles in daily/weekly/monthly temporal granularity for this DBMS.

We finally merge DBMS1 and DBMS2's corresponding top5 reads record for each pair (<daily, daily>, <weekly, weekly>, <monthly, monthly>). This yields the top5 read article for the total database.

We also need to populate the 'be-read' collection. We need to sort and traverse the 'read' collection and create 2 insertion classes for DBMS1 and DBMS2 respectively. For each document of 'read' collection, we use redis server to do a quick lookup on which DBMS this 'aid' belongs to, and then we put this document into the corresponding insertion class. Since read table is sorted, we can directly put all other aid that is equal and sum up 'readOrNot', 'agreeOrNot', 'commentOrNot', 'shareOrNot', as 'readUidList', 'agreeUidList', 'commentUidList', 'shareUidList' for the 'be-read' table. We also apply the bulk insertion model for 'be-read' table during our traversal.

Figure 7: Be-Read Insertion Model



We also implemented the functionality of adding a server at runtime. We need to first start the new server in the replica mode (with the same replica name as the existing server cluster). We then add the new server's configuration to the existing primary node, and the router will broadcast this reconfig to all existing secondary node. The new server then becomes a new secondary node in this cluster, and it will receive data synchronization from a running secondary node (or the primary node depending on the read preference config of this replica).

Figure 8: New-Server Addition Core Logic

```
def addServer(serverName, running_primary):
    client = MongoClient(serverName)
    conf = running_primary.admin.command({'replSetGetConfig': 1})
    conf['config']['members'].append({
        '_id': len(conf['config']['members']),
        'host': serverName,
        'priority': 0,
        'votes': 0
    })
    conf['config']['version'] += 1
    res = running_primary.admin.command({'replSetReconfig': conf['config']})
```

We can also migrate our data to a new data center during runtime. We will migrate the data first as bulk inserting all collections to the new site, and then switch the MongoDBManager's client to point to the new server. Note that since we do not use a config file for MongoDBManager, we cannot do a persistent switch to new data center (as if we reboot the program, the client of read/write will still be handled by the old data center).

Finally, we can also drop a server at will. Our database architecture composes of 2 secondary nodes and 1 primary node and 1 arbiter node. MongoDB's replica policy allows us to drop as many secondary nodes as we want. But once a primary node is dropped, there will be a reelection among the available secondary nodes. Since we have an arbiter, we do not need to be concern about the case that a secondary node cannot elect self [5].

In total, we can drop 2 secondary nodes, or 1 secondary node and 1 primary node, or 1 primary node and 1 arbiter node for each DBMS.

Finally, our MongoDBManager class is responsible for connecting with the HDFS system. We implemented a program and compiled a .jar file for large number of file uploads (multi-threaded uploading file with continuous progress logging). We also have a python helper class for efficient articles download by prefetching the entire directory and store the keys of downloaded files in redis server.

7 Solution Evaluations

To achieve high availability, we deploy multiple servers simultaneously for DBMS1 and DBMS2. For example, we deployed 4 data nodes in total (1 primary, 2 secondary, 1 arbiter) and 3 config servers for DBMS1, and we set the sync intervals from primary to secondary node as 2 seconds. This setting can allow up to 2 data-node server's failures and 1 config server failure.

Our bulk insert implementation is fast – the insertion of read table for roughly 50000 documents only take about 2 seconds, and most of time it spent is on read line and JSON str object parsing. However, our popular-rank computation is a little slow. It takes about 40 seconds to finish the entire popular-rank computation (including read-table traversal, timeKey mapping, and the final merge). The bottleneck should be on timeKey mapping since we have a data structure of timeKey dictionary inside a daily/weekly/monthly

dictionary, and we have 2 database to process. A potential better design is to allocate a dict for each AID for daily/weekly/monthly dictionary to avoid the costs of continuous dictionary expansion. Another possible enhancement direction will be editing the essential algorithm we are using: there should a dynamic programming solution by traversing from earliest read record to the latest and keep a DP table if possible. However, the time complexity of our algorithm is $O(n^2 \log n)$ and any great improvement of in time complexity might be unfeasible since the sorting part of this task has already induced the minimum time complexity as $O(n \log n)$. We think a fast $O(n^2)$ algorithm should be available, but that algorithm might not fit well in parallel computing (and thus not fit well in this distributed context).

Computing be-read table takes about 10-15 seconds. This time consumption mainly induced by traversing read-table and count the share/agree/comment occurrence for one aid. Some better techniques for improvement shall exist but such improvement should be trivial since such counting is inevitable.

Computing query is generally fast (as it only take less than 2 seconds), but the nature of NoSQL is against the efficiency of a complex join. Although we have manually implemented a left-join operation, our pipe is somewhat inflexible and only support simple operations such as single-table filter, 2 tables left join, sorting. In addition, our pipe do not leverage the collection statistics and its performance might encounter a bottleneck when we do a complex query.

For advanced functions, we have implemented server status log, data migration, dropping server at will, runtime server's join. Our server status log, although not comprehensive for printing all server info, is enough for collecting the essential info for data center's administrator to determine server's status since we have printed the primary-secondary node status, collections index/size in the desired database, and available keys in one collection. More detailed info is also available in Mongo Shell but we do not print it out. Our implementation of data migration is also primitive since we only collect documents and direct the output to the new server. We do not consider the workload on the existing cluster but in the real world, we need to implement a scheduled pipe since traversing all tables will interfere the working cluster's flows as it will block all write (insert/update) operations.

8 Conclusion and Future Perspective

We have constructed a small-size distributed database on MongoDB and build the connection to HDFS and cache some results with redis server. However, our database is still primitive, as we do not take much advantage of this distributed context to alleviate our workload (such as use collection statistics to determine query scheme, deploying some triggers between DBMS1 and DBMS2). We also do not take much advantage of combining native Mongo query commands with computing query in python. We also do not take much advantage of HDFS's distributed context since we only run one datanode in HDFS. Another subtle improvement is that we have only tried running Mongo on Python, which the performance might quite vary depending on the data structure we use (we mainly use Python built-in str, dict, list and occasionally we use NumPy array), implementing the exactly same operations in a compiled language like Java or C++ is expected to boost the speed and have a stable performance. These are all potential improvement directions we can do.

9 Work Contributions

Wentao started this project early and was mainly responsible developing basic tools and scripts for the MongoDB/HDFS/Redis while Haoshen mainly maintained and tested these scripts. Wentao and Haoshen together finished this report (Wentao was responsible for the second half while Haoshen finished the first half). Wentao recorded the demo presentation of the program on his laptop. Meanwhile, Haoshen finished the presentation slides and delivered the speech together with Wentao.

References

[1] Manoj Debnath. Getting started with mongodb as a java nosql solution. `https://www.developer.com/java/data/getting-started-with-mongodb-as-a-java-nosql-solution.html`.

[2] Guru99. What is mongodb. `https://www.guru99.com/what-is-mongodb.html`.

[3] Cynthia Harvey. Big data management. https://www.datamation.com/big-data/big-data-management.html.

[4] MongoDB Inc. Bulk write operations. https://pymongo.readthedocs.io/en/stable/examples/bulk.html.

[5] MongoDB Inc. Replica set elections. https://docs.mongodb.com/manual/core/replica-set-elections.

[6] MongoDB Inc. Replica set primary. https://docs.mongodb.com/manual/core/replica-set-primary/.

[7] tutorialspoint. hdfs-intro. https://www.tutorialspoint.com/hadoop/hadoop_hdfs_overview.htm.